

z/OS Communications Server
3.2

*IP Sockets Application Programming
Interface Guide and Reference*



Note:

Before using this information and the product it supports, be sure to read the general information under [“Notices” on page 801](#).

This edition applies to 3.1 of z/OS® (5655-ZOS), and to subsequent releases and modifications until otherwise indicated in new editions.

Last updated: 2025-09-20

© **Copyright International Business Machines Corporation 2000, 2025.**

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

Contents

Figures.....	xiii
Tables.....	xxiii
About this document.....	XXV
Who should read this document.....	xxv
How this document is organized.....	xxv
How to use this document.....	xxvi
How to provide feedback to IBM.....	xxvi
Conventions and terminology that are used in this information.....	xxvi
How to read a syntax diagram.....	xxviii
Prerequisite and related information.....	xxx
Summary of changes for IP Sockets Application Programming Interface Guide and Reference.....	xxxv
Summary of changes for z/OS 3.2.....	xxxv
Changes made in z/OS Communications Server 3.1.....	xxxv
Part 1. Overview.....	1
Chapter 1. Introducing TCP/IP concepts.....	3
TCP/IP concepts.....	3
Understanding sockets concepts.....	4
Introducing TCP/IP concepts: Programming with sockets.....	5
Introducing TCP/IP concepts: Selecting sockets	5
Introducing TCP/IP concepts: Socket libraries.....	6
Introducing TCP/IP concepts: Address families.....	8
Introducing TCP/IP concepts: Addressing sockets in an Internet domain.....	8
Chapter 2. Organizing a TCP/IP application program.....	13
Client and server socket programs.....	13
Client/server socket programs: Iterative server socket programs.....	13
Client/server socket programs: Concurrent server socket programs.....	14
Client/server socket programs: Call sequence in socket programs.....	15
Call sequence in stream socket sessions.....	15
Call sequence in datagram socket sessions.....	16
Client/server socket programs: Blocking, nonblocking, and asynchronous socket calls.....	17
Client/server socket programs: Testing a program using a miscellaneous server.....	19
Client/server socket programs: Testing a local machine using a loopback address.....	19
Client/server socket programs: Accessing required data sets.....	19
Part 2. Designing programs.....	21
Chapter 3. Designing an iterative server program.....	23
Allocating sockets in an iterative server program.....	23
Binding sockets in an iterative server program.....	25
Binding with a known port number.....	25
Binding using socket call gethostbyname.....	25
Binding a socket to a specific port number.....	26

Listening for client connection requests in an iterative server program.....	27
Accepting client connection requests in an iterative server program.....	28
Transferring data between sockets in an iterative server program.....	30
Closing a connection in an iterative server program.....	30
Active and passive closing in an iterative server program.....	30
Shutdown call in an iterative server program.....	31
Linger option in an iterative server program.....	32
Chapter 4. Designing a concurrent server program.....	33
Concurrent servers in native MVS environment.....	33
MVS subtasking considerations in a concurrent server program.....	34
Access to shared storage areas in a concurrent server program.....	34
Data set access in MVS.....	35
Task and workload management in a concurrent server program.....	36
Security considerations in a concurrent server program.....	36
Reentrant code in a concurrent server program.....	37
Understanding the structure of a concurrent server program.....	37
Selecting requests in a concurrent server program.....	37
Client connection requests in a concurrent server program.....	41
Passing sockets in a concurrent server program.....	42
Transferring data between sockets in a concurrent server program.....	45
Closing a concurrent server program.....	45
Chapter 5. Designing a client program.....	47
Allocating a socket in a client program.....	47
Connecting to a server in a client program.....	47
Transferring data between sockets in a client program.....	48
Closing a client program.....	48
Chapter 6. Designing a program to use datagram sockets.....	49
Datagram socket characteristics.....	49
Understanding datagram socket program structure.....	49
Allocating a datagram socket.....	50
Binding datagram sockets to port numbers.....	50
Streamline data transfer using connect call.....	50
Transferring data between datagram sockets.....	50
Chapter 7. Transferring data between sockets.....	51
Transferring data between sockets: Streams and messages.....	51
Transferring data between sockets: Data representation.....	54
Using send() and recv() calls.....	55
The send() and recv() call conversation.....	55
Using socket calls in a network application.....	56
Reading and writing data from and to a socket.....	57
Using sendto() and recvfrom() calls.....	58
Chapter 8. Designing IPv6 programs	61
Chapter 9. Designing multicast programs	63
Designing multicast programs: Multicast source filters.....	63
Designing multicast programs: IPv4 multicast options	65
IP_ADD_MEMBERSHIP and IP_DROP_MEMBERSHIP.....	65
IP_ADD_SOURCE_MEMBERSHIP and IP_DROP_SOURCE_MEMBERSHIP.....	68
IP_BLOCK_SOURCE and IP_UNBLOCK_SOURCE.....	69
IP_MULTICAST_IF.....	70
IP_MULTICAST_LOOP.....	71
IP_MULTICAST_TTL.....	72
Designing multicast programs: IPv6 multicast options.....	73

IPV6_JOIN_GROUP and IPV6_LEAVE_GROUP.....	73
IPV6_MULTICAST_IF.....	76
IPV6_MULTICAST_LOOP.....	76
IPV6_MULTICAST_HOPS.....	77
Designing multicast programs: Protocol-independent multicast options.....	78
MCAST_JOIN_GROUP and MCAST_LEAVE_GROUP.....	78
MCAST_JOIN_SOURCE_GROUP and MCAST_LEAVE_SOURCE_GROUP.....	80
MCAST_BLOCK_SOURCE and MCAST_UNBLOCK_SOURCE.....	80
Designing multicast programs: IOCTL multicast commands.....	81
SIOCGIPMSFILTER.....	81
SIOCGMSFILTER.....	82
SIOCSIPMSFILTER.....	82
SIOCSMSFILTER.....	82

Part 3. Application program interfaces..... 83

Chapter 10. C Socket application programming interface.....	85
Compiler restrictions with C applications.....	85
Compiling and linking C applications.....	86
C application compatibility considerations.....	86
C socket API: Non-reentrant modules.....	86
C socket API: Reentrant modules.....	88
C program compiler messages.....	91
C program abends.....	91
C socket implementation.....	91
C socket header files.....	92
Manifest.h header file.....	92
Prototyping.....	93
C structures.....	93
C socket API error messages and return codes.....	94
C socket calls.....	94
accept()	94
bind()	95
close()	98
connect()	99
endhostent()	102
endnetent()	102
endprotoent()	102
endservent()	103
fcntl()	103
getclientid()	104
getdtablesize()	105
gethostbyaddr()	105
gethostbyname()	106
gethostent()	107
gethostid()	108
gethostname()	108
getibmopt()	109
getibmsockopt()	110
getnetbyaddr()	111
getnetbyname()	112
getnetent()	113
getpeername()	113
getprotobyname()	114
getprotobynumber()	115
getprotoent()	115
getservbyname()	116

getservbyport()	117
getservent()	117
getsockname()	118
getsockopt()	119
givesocket()	123
htonl()	125
htons()	125
inet_addr()	126
inet_lnaof()	127
inet_makeaddr()	127
inet_netof()	128
inet_network()	128
inet_ntoa()	128
ioctl()	129
listen()	131
maxdesc()	132
ntohl()	133
ntohs()	133
read()	134
readv()	135
recv()	136
recvfrom()	137
recvmsg()	138
select()	140
selectex()	143
send()	144
sendmsg()	145
sendto()	147
sethostent()	148
setibmopt()	149
setibmsockopt()	150
setnetent()	151
setprotoent()	152
setservent()	152
setsockopt()	152
shutdown()	156
sock_debug()	157
sock_do_teststor()	157
socket()	158
takesocket()	160
tcperror()	161
write()	162
writew()	163
Sample C socket programs	164
Executing TCPS and TCPC modules	164
Executing UDPS and UDPC modules	164
C socket TCP client	165
C socket TCP server	166
C socket UDP server	168
C socket UDP client	170
Chapter 11. X/Open Transport Interface	173
XTI software requirements	173
What is provided with XTI	173
How XTI works in the z/OS environment	173
Creating an application using the XTI protocol	174
Coding XTI calls	174
Coding XTI calls: Initializing a transport endpoint	174

Coding XTI calls: Establishing a connection.....	175
Coding XTI calls: Transferring data.....	175
Coding XTI calls: Releasing a connection.....	175
Coding XTI calls: Disabling a connection.....	175
Coding XTI calls: Managing events.....	175
Coding XTI calls: Using utility calls.....	176
Coding XTI calls: Using system calls.....	176
Compiling and linking XTI applications using cataloged procedures.....	176
XTICL.....	177
XTIC.....	179
XTIS.....	180
Understanding XTI sample programs.....	180
XTI socket client sample program.....	181
XTI socket server sample program.....	186
Chapter 12. Macro application programming interface.....	193
Sockets API environmental restrictions and programming requirements.....	193
Sockets API input register information.....	194
Sockets API output register information.....	195
Sockets API compatibility considerations.....	195
Defining storage for the macro API	195
Understanding common parameter descriptions.....	197
Sockets API error messages and return codes.....	197
Characteristics of sockets.....	197
Task management and asynchronous function processing.....	198
Macro API asynchronous function processing: How it works.....	199
Asynchronous exit environmental and programming considerations.....	201
Using an unsolicited event-exit routine.....	201
Diagnosing problems in applications using the macro API.....	202
Macros for assembler programs.....	202
ACCEPT.....	202
BIND.....	205
BIND2ADDRSEL.....	208
CANCEL.....	211
CLOSE.....	213
CONNECT.....	215
FCNTL.....	218
FREEADDRINFO.....	220
GETADDRINFO.....	221
GETCLIENTID.....	228
GETHOSTBYADDR.....	230
GETHOSTBYNAME.....	232
GETHOSTID.....	234
GETHOSTNAME.....	236
GETIBMOPT.....	238
GETNAMEINFO.....	241
GETPEERNAME.....	245
GETSOCKNAME.....	248
GETSOCKOPT.....	251
GIVESOCKET.....	268
GLOBAL.....	270
INET6_IS_SRCADDR.....	271
INITAPI.....	274
IOCTL.....	278
LISTEN.....	287
NTOP.....	289
PTON.....	291
READ.....	293

READV.....	295
RECV.....	297
RECVFROM.....	301
RECVMSG.....	305
SELECT.....	309
SELECTEX.....	313
SEND.....	318
SENDMSG.....	320
SENDTO.....	324
SETSOCKOPT.....	327
SHUTDOWN.....	345
SOCKET.....	347
TAKESOCKET.....	350
TASK.....	352
TERMAPI.....	353
WRITE.....	354
WRITEV.....	356
Macro interface assembler language sample programs.....	358
EZASOKAS sample server program for IPv4.....	358
EZASOKAC sample client program for IPv4.....	365
EZASO6AS sample server program for IPv6.....	371
EZASO6AC sample client program for IPv6.....	381
Chapter 13. CALL instruction application programming interface.....	389
CALL instruction API environmental restrictions and programming requirements.....	389
CALL instruction API output register information.....	390
CALL instruction API compatibility considerations.....	390
CALL instruction application programming interface (API).....	391
Understanding COBOL, Assembler, and PL/I call formats.....	391
COBOL language call format.....	391
Assembler language call format.....	391
PL/I language call format.....	391
Converting parameter descriptions.....	392
Diagnosing problems in applications using the CALL instruction API.....	392
CALL instruction API error messages and return codes.....	393
Code CALL instructions.....	393
ACCEPT.....	393
BIND.....	395
BIND2ADDRSEL.....	397
CLOSE.....	400
CONNECT.....	401
FCNTL.....	404
FREEADDRINFO.....	406
GETADDRINFO.....	407
GETCLIENTID.....	414
GETHOSTBYADDR.....	415
GETHOSTBYNAME.....	418
GETHOSTID.....	420
GETHOSTNAME.....	421
GETIBMOPT.....	422
GETNAMEINFO.....	424
GETPEERNAME.....	428
GETSOCKNAME.....	430
GETSOCKOPT.....	432
GIVESOCKET.....	448
INET6_IS_SRCADDR.....	450
INITAPI.....	453
IOCTL.....	455

LISTEN.....	464
NTOP.....	465
PTON.....	467
READ.....	469
READV.....	470
RECV.....	472
RECVFROM.....	474
RECVMSG.....	477
SELECT.....	481
SELECTEX.....	484
SEND.....	489
SENDMSG.....	491
SENDTO.....	495
SETSOCKOPT.....	497
SHUTDOWN.....	513
SOCKET.....	515
TAKESOCKET.....	517
TERMAPI.....	519
WRITE.....	519
WRITEV.....	521
Using data translation programs for socket call interface.....	522
Assembler language utility programs call format.....	522
Data translation.....	522
Bit-string processing.....	523
Call interface sample programs.....	534
Sample code for IPv4 server program.....	534
Sample program for IPv4 client program.....	537
Sample code for IPv6 server program.....	539
Sample program for IPv6 client program.....	543
Common variables used in PL/I sample programs.....	546
Common variables used in COBOL sample programs.....	554
COBOL call interface sample IPv6 server program.....	559
COBOL call interface sample IPv6 client program.....	568
Chapter 14. REXX socket application programming interface.....	577
Overview.....	577
Supported REXX APIs.....	577
Prerequisites for using REXX sockets.....	577
Format of the REXX socket function and return values.....	577
REXX programming hints and tips.....	578
How structures are represented.....	580
Runtime functions.....	583
ACCEPT.....	586
BIND.....	588
BIND2ADDRSEL.....	590
CLOSE.....	592
CONNECT.....	593
FCNTL.....	596
GETADDRINFO.....	597
GETCLIENTID.....	605
GETDOMAINNAME.....	607
GETHOSTBYADDR.....	608
GETHOSTBYNAME.....	609
GETHOSTID.....	610
GETHOSTNAME.....	611
GETNAMEINFO.....	612
GETPEERNAME.....	614
GETPROTOBYNAME.....	616

GETPROTOBYNUMBER.....	616
GETSERVBYNAME.....	617
GETSERVBYPORT.....	618
GETSOCKNAME.....	619
GETSOCKOPT.....	621
GIVESOCKET.....	629
INET6ISSRCADDR.....	631
INITIALIZE.....	633
IOCTL.....	634
LISTEN.....	644
READ.....	645
RECV.....	647
RECVFROM.....	649
RESOLVE.....	652
SELECT.....	653
SEND.....	655
SENDTO.....	657
SETSOCKOPT.....	659
SHUTDOWN.....	669
SOCKET.....	670
SOCKETSET.....	672
SOCKETSETLIST.....	673
SOCKETSETSTATUS.....	674
TAKESOCKET.....	675
TERMINATE.....	677
VERSION.....	678
WRITE.....	678
Sample programs.....	680
Overview of REXX sample programs.....	680
The REXX-EXEC RSCLIENT sample program for IPv4.....	683
The REXX-EXEC RSSERVER sample program for IPv4.....	683
The REXX-EXEC R6CLIENT sample program for IPv6.....	683
The REXX-EXEC R6SERVER sample program for IPv6.....	684
AT-TLS security definitions for REXX samples.....	684
Running the REXX sample programs.....	684
Testing the GIVESOCKET and TAKESOCKET commands.....	684
Chapter 15. Pascal application programming interface.....	687
Steps for Pascal language API procedure calls.....	687
Pascal language API software requirements.....	688
Pascal API header files.....	688
Pascal language API compatibility considerations.....	688
Pascal language API data structures.....	689
Connection state.....	689
Connection information record.....	690
Notification record.....	691
File specification record.....	695
Pascal language API: using procedure calls.....	696
Notifications.....	696
TCP initialization procedures.....	696
TCP termination procedure.....	696
TCP communication procedures.....	696
PING interface.....	696
Monitor procedures.....	696
UDP communication procedures.....	696
Raw IP interface.....	696
Timer routines.....	697
Host lookup routines.....	697

Assembler calls.....	697
Other routines.....	697
Pascal return codes.....	697
Pascal language API procedure calls.....	699
AddUserNote.....	699
BeginTcpIp.....	700
ClearTimer.....	700
CreateTimer.....	701
DestroyTimer.....	701
EndTcpIp.....	701
GetHostNumber.....	702
GetHostResol.....	702
GetHostString.....	703
GetIdentity.....	703
GetNextNote.....	704
GetSmsg.....	704
Handle.....	705
IsLocalAddress.....	705
IsLocalHost.....	706
MonQuery.....	706
PingRequest.....	707
RawIpClose.....	708
RawIpOpen.....	708
RawIpReceive.....	709
RawIpSend.....	710
ReadXlateTable.....	711
SayCalRe.....	712
SayConSt.....	712
SayIntAd.....	712
SayIntNum.....	713
SayNotEn.....	713
SayPorTy.....	713
SayProTy.....	714
SetTimer.....	714
TcpAbort.....	714
TcpClose.....	715
TcpFReceive, TcpReceive, and TcpWaitReceive.....	716
TcpFSend, TcpSend, and TcpWaitSend.....	718
TcpNameChange.....	720
TcpOpen and TcpWaitOpen.....	720
TcpOption.....	722
TcpStatus.....	723
UdpClose.....	724
UdpNReceive.....	724
UdpOpen.....	725
UdpReceive.....	726
UdpSend.....	726
Unhandle.....	727
Sample Pascal program.....	728
Building the sample Pascal API module.....	728
Running the sample module.....	728
Sample Pascal application program.....	729
Appendix A. Multitasking C socket sample program.....	733
Server sample program in C.....	733
The subtask sample program in C.....	738
The client sample program in C.....	740

Appendix B. Socket call error return codes.....	743
System error codes for socket calls.....	743
Sockets return codes (ERRNOs).....	743
z/OS UNIX return codes.....	754
Additional return codes.....	754
Sockets extended ERRNOs.....	755
User abend U4093.....	759
Appendix C. Address family cross reference.....	761
Appendix D. GETSOCKOPT/SETSOCKOPT command values.....	767
Appendix E. Abbreviations and acronyms.....	771
Appendix F. GETNAMEINFO flags and returned information examples.....	777
Appendix G. Related protocol specifications.....	779
Appendix H. Accessibility.....	799
Notices.....	801
Terms and conditions for product documentation.....	802
IBM Online Privacy Statement.....	803
Policy for unsupported hardware.....	803
Minimum supported hardware.....	803
Programming interface information.....	804
Policy for unsupported hardware.....	804
Trademarks.....	804
Bibliography.....	805
Index.....	809

Figures

1. The TCP/IP protocol stack.....	3
2. Socket concept.....	4
3. TCP/IP networking API relationship on z/OS.....	6
4. The port concept.....	9
5. Port number assignments.....	9
6. Iterative server main logic.....	14
7. A typical stream socket session.....	16
8. A typical datagram socket session.....	17
9. Socket call variables.....	24
10. MVS TCP/IP socket descriptor table.....	24
11. An application using the bind() call.....	25
12. A bind() call using gethostbyname().....	26
13. Variables used for the BIND call.....	26
14. Variables used by the listen call.....	27
15. Variables used by the ACCEPT call.....	29
16. Socket states.....	30
17. Closing sockets.....	31
18. Serialized access to a shared storage area.....	34
19. Synchronized use of a common service task.....	35
20. Concurrent server in an MVS address space.....	37
21. To set/test bits for SELECT calls.....	40
22. An application using the select() call.....	41
23. Accepting a client connection.....	42

24. Giving a socket to a subtask.....	44
25. Taking sockets from the main process.....	45
26. Finding the IP address of a server host using gethostbyname().....	47
27. Layout of a message between a TPI client and a TPI server.....	52
28. Transaction request message segment.....	52
29. The TCP buffer flush technique.....	53
30. Big or little endian byte order for a 2-byte integer.....	54
31. An application using the send() and recv() calls.....	56
32. An application using the sendto() and recvfrom() Calls.....	59
33. IP_ADD_MEMBERSHIP and IP_DROP_MEMBERSHIP.....	68
34. IP_ADD_SOURCE_MEMBERSHIP and IP_DROP_SOURCE_MEMBERSHIP.....	69
35. IP_BLOCK_SOURCE and IP_UNBLOCK_SOURCE.....	70
36. IP_MULTICAST_IF	71
37. IP_MULTICAST_LOOP.....	72
38. IP_MULTICAST_TTL.....	73
39. IPV6_JOIN_GROUP and IPV6_LEAVE_GROUP	76
40. IPV6_MULTICAST_IF.....	76
41. IPV6_MULTICAST_LOOP	77
42. IPV6_MULTICAST_HOPS.....	78
43. MCAST_JOIN_GROUP and MCAST_LEAVE_GROUP.....	79
44. MCAST_JOIN_SOURCE_GROUP and MCAST_LEAVE_SOURCE_GROUP.....	80
45. MCAST_BLOCK_SOURCE and MCAST_UNBLOCK_SOURCE.....	81
46. Sample JCL for compiling non-reentrant modules.....	87
47. Sample JCL for linking non-reentrant modules.....	88
48. Sample JCL for running non-reentrant modules.....	88

49. Sample JCL for compiling reentrant modules.....	89
50. Sample JCL for prelinking and linking reentrant modules.....	90
51. Sample JCL for running the reentrant program.....	90
52. C socket TCP client sample.....	166
53. C socket TCP server sample.....	168
54. C socket UDP server sample.....	170
55. C socket UDP client sample.....	171
56. Using XTI with TCP/IP.....	174
57. Sample compile and link-edit job control procedure.....	178
58. Sample client execution job control procedure.....	179
59. Sample server execution job control procedure.....	180
60. Sample client code for XTI.....	186
61. Sample server code for XTI.....	192
62. ECB input parameter.....	199
63. User token setting.....	199
64. HOSTENT structure returned by the GETHOSTBYADDR macro.....	231
65. HOSTENT structure returned by the GETHOSTBYNAME macro.....	234
66. NUM_IMAGES field settings.....	240
67. EZASOKAS sample server program for IPv4.....	365
68. EZASOKAC sample client program for IPv4.....	371
69. EZASO6AS sample server program for IPv6.....	380
70. EZASO6AC sample client program for IPv6.....	388
71. Storage definition statement examples.....	392
72. ACCEPT call instructions example.....	394
73. BIND call instruction example.....	396

74. BIND2ADDRSEL call instruction example.....	399
75. CLOSE call instruction example.....	401
76. CONNECT call instruction example.....	403
77. FCNTL call instruction example.....	405
78. FREEADDRINFO call instruction example.....	406
79. GETADDRINFO call instruction example.....	408
80. GETCLIENTID call instruction example.....	415
81. GETHOSTBYADDR call instruction example.....	416
82. HOSTENT structure that is returned by the GETHOSTBYADDR call.....	417
83. GETHOSTBYNAME call instruction example.....	418
84. HOSTENT structure returned by the GETHOSTBYNAME call.....	419
85. GETHOSTID call instruction example.....	420
86. GETHOSTNAME call instruction example.....	421
87. GETIBMOPT call instruction example.....	423
88. Example of name field	424
89. GETNAMEINFO call instruction example.....	425
90. GETPEERNAME call instruction example.....	429
91. GETSOCKNAME call instruction example.....	431
92. GETSOCKOPT call instruction example.....	433
93. GIVESOCKET call instruction example.....	449
94. INET6_IS_SRCADDR call instruction example.....	451
95. INITAPI call instruction example.....	454
96. IOCTL call instruction example.....	456
97. COBOL language example for SIOCGHOMEIF6.....	457
98. COBOL language example for SIOCGIFNAMEINDEX.....	459

99. COBOL II example for SIOCGIFCONF.....	464
100. LISTEN call instruction example.....	464
101. NTOP call instruction example.....	466
102. PTON call instruction example.....	468
103. READ call instruction example.....	470
104. READV call instruction example.....	471
105. RECV call instruction example.....	473
106. RECVFROM call instruction example.....	475
107. RECVMSG call instruction example.....	478
108. SELECT call instruction example.....	483
109. SELECTEX call instruction example.....	486
110. SEND call instruction example.....	490
111. SENDMSG call instruction example.....	493
112. SENDTO call instruction example.....	496
113. SETSOCKOPT call instruction example.....	498
114. SHUTDOWN call instruction example.....	514
115. SOCKET call instruction example.....	515
116. TAKESOCKET call instruction example.....	518
117. TERMAPI call instruction example.....	519
118. WRITE call instruction example.....	520
119. WRITEV call instruction example.....	521
120. EZACIC04 EBCDIC-to-ASCII table.....	523
121. EZACIC04 call instruction example.....	524
122. EZACIC05 ASCII-to-EBCDIC table.....	524
123. EZACIC05 call instruction example.....	525

124. EZACIC06 call instruction example.....	525
125. EZAZIC08 call instruction example.....	528
126. EZACIC09 call instruction example.....	531
127. EZACIC14 EBCDIC-to-ASCII table.....	532
128. EZACIC14 call instruction example.....	532
129. EZACIC15 ASCII-to-EBCDIC table.....	533
130. EZACIC15 call instruction example.....	533
131. EZASOKPS PL/1 sample server program for IPv4.....	537
132. EZASOKPC PL/1 sample client program for IPv4.....	539
133. EZASO6PS PL/1 sample server program for IPv6.....	543
134. EZASO6PC PL/1 sample client program for IPv6.....	546
135. CBLOCK PL/1 common variables.....	554
136. EZACOBOL COBOL common variables.....	559
137. EZASO6CS COBOL call interface sample IPv6 server program.....	568
138. EZASO6CC COBOL call interface sample IPv6 client program.....	575
139. ACCEPT command example.....	588
140. CLOSE command example.....	593
141. CONNECT command example.....	595
142. FCNTL command example.....	597
143. GETCLIENTID command example.....	607
144. GETDOMAINNAME command example.....	608
145. GETHOSTBYADDR command example.....	609
146. GETHOSTBYNAME command example.....	610
147. GETHOSTID command example.....	611
148. GETHOSTNAME command example.....	612

149. GETNAMEINFO command example.....	614
150. GETPEERNAME command example.....	615
151. GETPROTOBYNAME command example.....	616
152. GETPROTOBYNUMBER command example.....	617
153. GETSERVBYNAME command example.....	618
154. GETSERVBYPOR command example.....	619
155. GETSOCKNAME command example.....	621
156. ASCII to EBCDIC.....	624
157. EBCDIC to ASCII.....	624
158. GETSOCKOPT command example.....	629
159. READ command example.....	647
160. READ command example.....	652
161. RESOLVE command example.....	653
162. SELECT command example.....	655
163. ASCII to EBCDIC.....	665
164. EBCDIC to ASCII.....	665
165. SOCKET command example.....	672
166. SOCKETSET command example.....	673
167. SOCKETSETLIST command example.....	674
168. SOCKETSETSTATUS command example.....	675
169. TERMINATE command example.....	678
170. VERSION command example.....	678
171. Pascal declaration of connection state type.....	689
172. Pascal declaration of connection information record.....	690
173. Pascal declaration of socket type.....	691

174. Notification record.....	692
175. Pascal declaration of file specification record.....	695
176. Sample calling sequence.....	700
177. BeginTcpIp example.....	700
178. ClearTimer example.....	701
179. Create timer example.....	701
180. Destroy timer example.....	701
181. EndTcpIp example.....	701
182. GetHostNumber example.....	702
183. GetHostResol example.....	702
184. GetHostString example.....	703
185. GetIdentity example.....	703
186. GetNextNote example.....	704
187. GetSmsg example.....	704
188. Handle example.....	705
189. IsLocalAddress example.....	705
190. IsLocalHost example.....	706
191. MonQuery example.....	706
192. Monitor query record.....	707
193. PingRequest example.....	707
194. RawIpClose example.....	708
195. RawIpOpen example.....	709
196. RawIpReceive example.....	709
197. RawIpSend example.....	710
198. ReadXlateTable example.....	711

199. SayCalRe example.....	712
200. SayConSt example.....	712
201. SayIntAd example.....	712
202. SayIntNum example.....	713
203. SayNotEn example.....	713
204. SayPorTy example.....	713
205. SayProTy example.....	714
206. SetTimer example.....	714
207. TcpAbort example.....	714
208. TcpClose example.....	715
209. TcpFReceive example.....	716
210. TcpReceive example.....	716
211. TcpWaitReceive example.....	716
212. TcpFSend example.....	718
213. TcpSend example.....	718
214. TcpWaitSend example.....	719
215. TcpNameChange example.....	720
216. TcpOpen example.....	720
217. TcpWaitOpen example.....	720
218. TcpOption example.....	722
219. TcpStatus example.....	723
220. UdpClose example.....	724
221. UdpNReceive example.....	724
222. UdpOpen example.....	725
223. UdpReceive example.....	726

224. UdpSend example.....	727
225. Unhandle example.....	727
226. Sample Pascal API with receive option.....	728
227. Sample Pascal API with send option.....	729
228. Sample Pascal application program.....	732
229. MTCCSUB C socket server program sample.....	740
230. MTCCLNT C socket server program sample.....	742
231. Example of abend U4093.....	760

Tables

1. Socket programming interface actions.....	18
2. TCP/IP data sets and applications.....	20
3. Effect of shutdown socket call.....	32
4. First fullword passed in a bit string select().....	38
5. Second fullword passed in a bit string using select().....	39
6. C structures in assembler language format.....	93
7. Initializing a call.....	174
8. Establishing a connection.....	175
9. Transferring data.....	175
10. Releasing a connection.....	175
11. Disabling a connection.....	175
12. Managing events.....	176
13. Using utilities.....	176
14. System function calls.....	176
15. OPTNAME options for GETSOCKOPT and SETSOCKOPT.....	254
16. IOCTL macro arguments.....	284
17. OPTNAME options for GETSOCKOPT and SETSOCKOPT.....	331
18. OPTNAME options for GETSOCKOPT and SETSOCKOPT.....	434
19. IOCTL call arguments.....	462
20. OPTNAME options for GETSOCKOPT and SETSOCKOPT.....	499
21. REXX socket sample programs.....	680
22. TCP connection states.....	690
23. Pascal language return codes.....	697

24. Sockets ERRNOs.....	743
25. Sockets extended ERRNOs.....	755
26. C socket address families cross reference	761
27. MACRO, CALL, REXX, socket address families cross reference	764
28. MACRO, CALL, REXX, exceptions.....	765
29. GETSOCKOPT/SETSOCKOPT command values for Macro, Assembler, COBOL and PL/I.....	767
30. GETSOCKOPT/SETSOCKOPT optname value for C programs.....	768

About this document

This document describes the syntax of the TCP/IP application programming interface (API). The APIs described in this document can be used to create TCP/IP client and server applications or modify existing applications to communicate using TCP/IP. The information in this document supports both IPv6 and IPv4. Unless explicitly noted, information describes IPv4 networking protocol. IPv6 support is qualified within the text.

This document refers to Communications Server data sets by their default SMP/E distribution library name. Your installation might, however, have different names for these data sets where allowed by SMP/E, your installation personnel, or administration staff. For instance, this document refers to samples in SEZAINST library as simply in SEZAINST. Your installation might choose a data set name of SYS1.SEZAINST, CS390.SEZAINST or other high-level qualifiers for the data set name.

To provide flexibility in writing new applications and adapting existing applications, the following programming languages and interfaces are described:

- C sockets
- X/Open Transport Interface (XTI)
- Assembler, PL/I, and COBOL sockets
- REXX sockets
- Pascal language

Who should read this document

This document is intended for experienced programmers familiar with MVS, the IBM® multiple virtual storage operating system, TCP/IP protocols, UNIX sockets, and data networks.

To use this document, you should be familiar with MVS and the IBM timesharing option (TSO).

You should also be familiar with z/OS Communications Server and installing and customizing any required programming products for your network.

Depending on the design and function of your application, you should be familiar with one or more of the following programming languages:

- Assembler
- C
- COBOL
- Pascal
- PL/I
- REXX

How this document is organized

This document is organized into the following parts:

- [Part 1, “Overview,” on page 1](#) presents an overview of TCP/IP concepts and organizing a TCP/IP application program.
- [Part 2, “Designing programs,” on page 21](#) describes ways to design various types of programs.
- [Part 3, “Application program interfaces,” on page 83](#) describes the following socket application program interfaces (APIs):
 - C Socket application programming interface (API)
 - X/Open Transport Interface (XTI)

- Macro application programming interface (API)
- CALL instruction application programming interface (API)
- REXX socket application programming interface (API)
- Pascal application programming interface (API)
- Appendixes provide additional information for this document.
- “Notices” on page 801 contains notices and trademarks used in this document.
- “Bibliography” on page 805 contains descriptions of the documents in the z/OS Communications Server library.

How to use this document

To use this document, you should be familiar with z/OS TCP/IP Services and the TCP/IP suite of protocols.

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.

Conventions and terminology that are used in this information

Commands in this information that can be used in both TSO and z/OS UNIX environments use the following conventions:

- When describing how to use the command in a TSO environment, the command is presented in uppercase (for example, NETSTAT).
- When describing how to use the command in a z/OS UNIX environment, the command is presented in bold lowercase (for example, **netstat**).
- When referring to the command in a general way in text, the command is presented with an initial capital letter (for example, Netstat).

All the exit routines described in this information are *installation-wide exit routines*. The installation-wide exit routines also called installation-wide exits, exit routines, and exits throughout this information.

The TPF logon manager, although included with VTAM®, is an application program; therefore, the logon manager is documented separately from VTAM.

Samples used in this information might not be updated for each release. Evaluate a sample carefully before applying it to your system.

z/OS no longer supports mounting HFS data sets (The POSIX style file system). Instead, a z/OS File System (zFS) can be implemented. The term hierarchical file system, abbreviated as HFS, is defined as a data structure that has a hierarchical nature with directories and files. References to hierarchical file systems or HFS might still be in use in z/OS Communications Server publications.

Network Express and Open Systems Adapter-Express (OSA-Express) terminology:

- The Network Express feature is introduced with the IBM z17 processor family. The Network Express feature is the next generation of Open Systems Adapter (OSA) technology. The term OSA (Open Systems Adapter) is carried forward with Network Express. The IBM z17 processor supports both the Network Express and the OSA-Express7S features. In this information, when a general reference is made to OSA that applies to all these features, then the term OSA is used, and the acronym will appear in italics. This formatting style and guideline for usage for the term OSA is used throughout this document. When a distinction is necessary, then the specific feature name is used such as the Network Express feature
- The Network Express feature is defined as channel (CHPID) type OSH (Open System Adapter for Hybrid networks) that might operate in either 10 GbE or 25 GbE link speed. When this term is used in this information, the processing being described applies to either link speed. If processing is applicable to only one link speed, the full terminology, for instance, IBM 25 GbE Network Express will be used.

- Network Express is defined with new system architecture called Enhanced Queued Direct I/O (EQDIO). In this information there are many references to QDIO or OSA/QDIO. When the reference applies to both QDIO and EQDIO the reference just indicates OSA. When the reference is specific to the QDIO or EQDIO architecture, then the specific architecture is referenced, for example, OSA/QDIO or OSA/EQDIO. Some OSA references also use or include the channel type for OSA such as OSD (QDIO). When the reference applies to both features, then the term OSA is used. When a distinction is necessary then the specific channel or architecture type is used, OSD/QDIO or OSH/EQDIO.

Shared Memory Communications over Remote Direct Memory Access (SMC-R) terminology

- *RoCE*, which is a generic term representing IBM® 10 GbE RoCE Express, IBM 10 GbE RoCE Express2, IBM 25 GbE RoCE Express2, IBM 10 GbE RoCE Express3, IBM 25 GbE RoCE Express3, IBM 10 GbE Network Express and IBM 25 GbE Network Express feature capabilities. When this term is used in this information, the processing being described applies to all of these features. If processing is applicable to only one feature, the full terminology, for instance, Network Express will be used.
- RoCE Express2, which is a generic term representing an IBM RoCE Express2 feature that might operate in either 10 GbE or 25 GbE link speed. When this term is used in this information, the processing being described applies to either link speed. If processing applies to only one link speed, the full terminology, for instance, IBM 25 GbE RoCE Express2 will be used.
- RoCE Express3, which is a generic term representing an IBM RoCE Express3 feature that might operate in either 10 GbE or 25 GbE link speed. When this term is used in this information, the processing being described applies to either link speed. If processing applies to only one link speed, the full terminology, for instance, IBM 25 GbE RoCE Express3 will be used.
- Network Express, which is a generic term representing an Network Express feature that might operate in either 10 GbE or 25 GbE link speed. When this term is used in this information, the processing being described applies to either link speed. If processing is applicable to only one link speed, the full terminology, for instance, IBM 25 GbE Network Express will be used. When configured with a CHPID type of NETH, the Network Express feature may operate as an RDMA network interface card.
- RDMA network interface card (RNIC), which is used to refer to the IBM 10 GbE RoCE Express, IBM 10 GbE RoCE Express2, IBM 25 GbE RoCE Express2, IBM 10 GbE RoCE Express3, or IBM 25 GbE RoCE Express3, IBM 10 GbE Network Express or IBM 25 GbE Network Express feature.
- Shared RoCE environment, which means that the *RoCE* feature can be used concurrently, or shared, by multiple operating system instances. The feature is considered to operate in a shared RoCE environment even if you use it with a single operating system instance.

Clarification of notes

Information traditionally qualified as Notes is further qualified as follows:

Attention

Indicate the possibility of damage

Guideline

Customary way to perform a procedure

Note

Supplemental detail

Rule

Something you must do; limitations on your actions

Restriction

Indicates certain conditions are not supported; limitations on a product or facility

Requirement

Dependencies, prerequisites

Result

Indicates the outcome

Tip

Offers shortcuts or alternative ways of performing an action; a hint

How to read a syntax diagram

This syntax information applies to all commands and statements that do not have their own syntax described elsewhere.

The syntax diagram shows you how to specify a command so that the operating system can correctly interpret what you type. Read the syntax diagram from left to right and from top to bottom, following the horizontal line (the main path).

Symbols and punctuation

The following symbols are used in syntax diagrams:

Symbol	Description
➤➤	Marks the beginning of the command syntax.
➤	Indicates that the command syntax is continued.
	Marks the beginning and end of a fragment or part of the command syntax.
➤➤	Marks the end of the command syntax.

You must include all punctuation such as colons, semicolons, commas, quotation marks, and minus signs that are shown in the syntax diagram.

Commands

Commands that can be used in both TSO and z/OS UNIX environments use the following conventions in syntax diagrams:

- When describing how to use the command in a TSO environment, the command is presented in uppercase (for example, NETSTAT).
- When describing how to use the command in a z/OS UNIX environment, the command is presented in bold lowercase (for example, netstat).

Parameters

The following types of parameters are used in syntax diagrams.

Required

Required parameters are displayed on the main path.

Optional

Optional parameters are displayed below the main path.

Default

Default parameters are displayed above the main path.

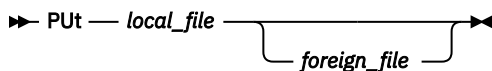
Parameters are classified as keywords or variables. For the TSO and MVS console commands, the keywords are not case sensitive. You can code them in uppercase or lowercase. If the keyword appears in the syntax diagram in both uppercase and lowercase, the uppercase portion is the abbreviation for the keyword (for example, OPERand).

For the z/OS UNIX commands, the keywords must be entered in the case indicated in the syntax diagram.

Variables are italicized, appear in lowercase letters, and represent names or values you supply. For example, a data set is a variable.

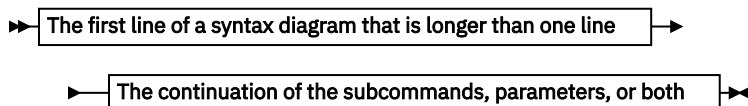
Syntax examples

In the following example, the P`Ut` subcommand is a keyword. The required variable parameter is *local_file*, and the optional variable parameter is *foreign_file*. Replace the variable parameters with your own values.



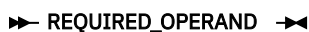
Longer than one line

If a diagram is longer than one line, the first line ends with a single arrowhead and the second line begins with a single arrowhead.



Required operands

Required operands and values appear on the main path line. You must code required operands and values.



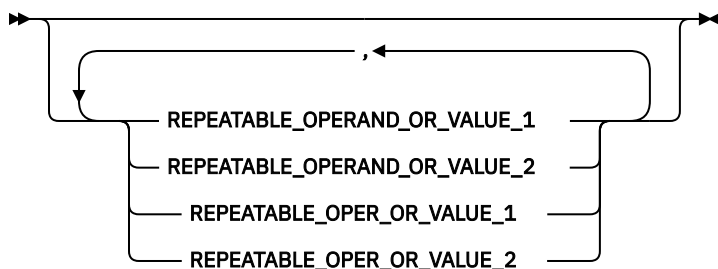
Optional values

Optional operands and values appear below the main path line. You do not have to code optional operands and values.



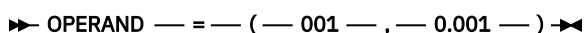
Selecting more than one operand

An arrow returning to the left above a group of operands or values means more than one can be selected, or a single one can be repeated.



Nonalphanumeric characters

If a diagram shows a character that is not alphanumeric (such as parentheses, periods, commas, and equal signs), you must code the character as part of the syntax. In this example, you must code `OPERAND=(001,0.001)`.



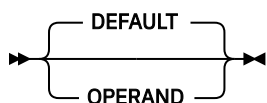
Blank spaces in syntax diagrams

If a diagram shows a blank space, you must code the blank space as part of the syntax. In this example, you must code OPERAND=(001 FIXED).

➤ OPERAND — = — (— 001 — — FIXED —) ➤

Default operands

Default operands and values appear above the main path line. TCP/IP uses the default if you omit the operand entirely.



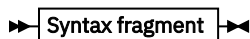
Variables

A word in all lowercase italics is a *variable*. Where you see a variable in the syntax, you must replace it with one of its allowable names or values, as defined in the text.

➤ *variable* ➤

Syntax fragments

Some diagrams contain syntax fragments, which serve to break up diagrams that are too long, too complex, or too repetitious. Syntax fragment names are in mixed case and are shown in the diagram and in the heading of the fragment. The fragment is placed below the main diagram.



Syntax fragment

➤ 1ST_OPERAND — , — 2ND_OPERAND — , — 3RD_OPERAND ➤

Prerequisite and related information

z/OS Communications Server function is described in the z/OS Communications Server library. Descriptions of those documents are listed in [“Bibliography”](#) on page 805, in the back of this document.

Required information

Before using this product, you should be familiar with TCP/IP, VTAM, MVS, and UNIX System Services.

Softcopy information

Softcopy publications are available in the following collection.

Titles	Description
<i>IBM Z Redbooks</i>	The IBM Z® subject areas range from e-business application development and enablement to hardware, networking, Linux®, solutions, security, parallel sysplex, and many others. For more information about the Redbooks® publications, see http://www.redbooks.ibm.com/ and http://www.ibm.com/systems/z/os/zos/zfavorites/ .

Other documents

This information explains how z/OS references information in other documents.

When possible, this information uses cross-document links that go directly to the topic in reference using shortened versions of the document title. For complete titles and order numbers of the documents for all products that are part of z/OS, see [z/OS Information Roadmap \(SA23-2299\)](#). The Roadmap describes what level of documents are supplied with each release of z/OS Communications Server, and also describes each z/OS publication.

To find the complete z/OS library, visit the [z/OS library](#) in [IBM Documentation](#) (<https://www.ibm.com/docs/en/zos>).

Relevant RFCs are listed in an appendix of the IP documents. Architectural specifications for the SNA protocol are listed in an appendix of the SNA documents.

The following table lists documents that might be helpful to readers.

Title	Number
<i>DNS and BIND</i> , Fifth Edition, O'Reilly Media, 2006	ISBN 13: 978-0596100575
<i>Routing in the Internet</i> , Second Edition, Christian Huitema (Prentice Hall 1999)	ISBN 13: 978-0130226471
<i>sendmail</i> , Fourth Edition, Bryan Costales, Claus Assmann, George Jansen, and Gregory Shapiro, O'Reilly Media, 2007	ISBN 13: 978-0596510299
<i>SNA Formats</i>	GA27-3136
<i>TCP/IP Illustrated, Volume 1: The Protocols</i> , W. Richard Stevens, Addison-Wesley Professional, 1994	ISBN 13: 978-0201633467
<i>TCP/IP Illustrated, Volume 2: The Implementation</i> , Gary R. Wright and W. Richard Stevens, Addison-Wesley Professional, 1995	ISBN 13: 978-0201633542
<i>TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols</i> , W. Richard Stevens, Addison-Wesley Professional, 1996	ISBN 13: 978-0201634952
<i>TCP/IP Tutorial and Technical Overview</i>	GG24-3376
<i>Understanding LDAP</i>	SG24-4986
z/OS Cryptographic Services System SSL Programming	SC14-7495
z/OS IBM Tivoli Directory Server Administration and Use for z/OS	SC23-6788
z/OS JES2 Initialization and Tuning Guide	SA32-0991
z/OS Problem Management	SC23-6844
z/OS MVS Diagnosis: Reference	GA32-0904
z/OS MVS Diagnosis: Tools and Service Aids	GA32-0905
z/OS MVS Using the Subsystem Interface	SA38-0679
z/OS Program Directory	GI11-9848
z/OS UNIX System Services Command Reference	SA23-2280
z/OS UNIX System Services Planning	GA32-0884
z/OS UNIX System Services Programming: Assembler Callable Services Reference	SA23-2281
z/OS UNIX System Services User's Guide	SA23-2279
z/OS C/C++ Runtime Library Reference	SC14-7314

Title	Number
OSA-Express Customer's Guide and Reference	SA22-7935

Redbooks publications

The following Redbooks publications might help you as you implement z/OS Communications Server.

Title	Number
<i>IBM z/OS Communications Server TCP/IP Implementation, Volume 1: Base Functions, Connectivity, and Routing</i>	SG24-8096
<i>IBM z/OS Communications Server TCP/IP Implementation, Volume 2: Standard Applications</i>	SG24-8097
<i>IBM z/OS Communications Server TCP/IP Implementation, Volume 3: High Availability, Scalability, and Performance</i>	SG24-8098
<i>IBM z/OS Communications Server TCP/IP Implementation, Volume 4: Security and Policy-Based Networking</i>	SG24-8099
<i>IBM Communication Controller Migration Guide</i>	SG24-6298
<i>IP Network Design Guide</i>	SG24-2580
<i>Managing OS/390 TCP/IP with SNMP</i>	SG24-5866
<i>Migrating Subarea Networks to an IP Infrastructure Using Enterprise Extender</i>	SG24-5957
<i>SecureWay Communications Server for OS/390 V2R8 TCP/IP: Guide to Enhancements</i>	SG24-5631
<i>SNA and TCP/IP Integration</i>	SG24-5291
<i>TCP/IP in a Sysplex</i>	SG24-5235
<i>TCP/IP Tutorial and Technical Overview</i>	GG24-3376
<i>Threadsafe Considerations for CICS</i>	SG24-6351

Where to find related information on the Internet

z/OS

This site provides information about z/OS Communications Server release availability, migration information, downloads, and links to information about z/OS technology

<http://www.ibm.com/systems/z/os/zos/>

z/OS Internet Library

Use this site to view and download z/OS Communications Server documentation

<http://www.ibm.com/systems/z/os/zos/library/bkserv/>

z/OS Communications Server product

The page contains z/OS Communications Server product introduction

<https://www.ibm.com/products/zos-communications-server>

IBM Communications Server product support

Use this site to submit and track problems and search the z/OS Communications Server knowledge base for Technotes, FAQs, white papers, and other z/OS Communications Server information

<https://www.ibm.com/mysupport>

IBM Communications Server performance information

This site contains links to the most recent Communications Server performance reports

<http://www.ibm.com/support/docview.wss?uid=swg27005524>

IBM Systems Center publications

Use this site to view and order Redbooks publications, Redpapers, and Technotes

<http://www.redbooks.ibm.com/>

z/OS Support Community

Search the z/OS Support Community Library for Techdocs (including Flashes, presentations, Technotes, FAQs, white papers, Customer Support Plans, and Skills Transfer information)

[z/OS Support Community](#)

Tivoli® NetView for z/OS

Use this site to view and download product documentation about Tivoli NetView for z/OS

<http://www.ibm.com/support/knowledgecenter/SSZJDU/welcome>

RFCs

Search for and view Request for Comments documents in this section of the Internet Engineering Task Force website, with links to the RFC repository and the IETF Working Groups web page

<http://www.ietf.org/rfc.html>

Internet drafts

View Internet-Drafts, which are working documents of the Internet Engineering Task Force (IETF) and other groups, in this section of the Internet Engineering Task Force website

<http://www.ietf.org/ID.html>

Information about web addresses can also be found in information APAR II11334.

Note: Any pointers in this publication to websites are provided for convenience only and do not serve as an endorsement of these websites.

DNS websites

For more information about DNS, see the following USENET news groups and mailing addresses:

USENET news groups

comp.protocols.dns.bind

BIND mailing lists

<https://lists.isc.org/mailman/listinfo>

BIND Users

- Subscribe by sending mail to bind-users-request@isc.org.
- Submit questions or answers to this forum by sending mail to bind-users@isc.org.

BIND 9 Users (This list might not be maintained indefinitely.)

- Subscribe by sending mail to bind9-users-request@isc.org.
- Submit questions or answers to this forum by sending mail to bind9-users@isc.org.

The z/OS Basic Skills Information Center

The z/OS Basic Skills Information Center is a web-based information resource intended to help users learn the basic concepts of z/OS, the operating system that runs most of the IBM mainframe computers

in use today. The Information Center is designed to introduce a new generation of Information Technology professionals to basic concepts and help them prepare for a career as a z/OS professional, such as a z/OS systems programmer.

Specifically, the z/OS Basic Skills Information Center is intended to achieve the following objectives:

- Provide basic education and information about z/OS without charge
- Shorten the time it takes for people to become productive on the mainframe
- Make it easier for new people to learn z/OS

To access the z/OS Basic Skills Information Center, open your web browser to the following website, which is available to all users (no login required): <https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zbasics/homepage.html?cp=zosbasics>

Summary of changes for IP Sockets Application Programming Interface Guide and Reference

This document contains terminology, maintenance, and editorial changes, including changes to improve consistency and retrievability. 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.

Summary of changes for z/OS 3.2

The following content is new, changed, or no longer included in z/OS 3.2.

New

The following content is new.

September 2025 release

- None.

Changed

The following content is changed.

September 2025 release

- None.

Deleted

The following content is deleted.

September 2025 release

- None.

Changes made in z/OS Communications Server 3.1

This information contains no technical changes for this release.

Part 1. Overview

For native IPv4 addresses, the application must create an AF_INET address family socket. For native IPv6 addresses and IPv4-mapped IPv6 addresses, the application must create an AF_INET6 address family socket. See [z/OS Communications Server: IPv6 Network and Appl Design Guide](#) and the SOCKET command under the APIs that support IPv6 for details.

For details on which TCP/IP APIs and commands support the AF_INET6 (IPv6) address family, see [Appendix C, “Address family cross reference,” on page 761](#).

Chapter 1. Introducing TCP/IP concepts

This information explains basic TCP/IP concepts and sockets programming.

TCP/IP concepts

Conceptually, the TCP/IP protocol stack consists of four layers, each layer consisting of one or more protocols. A protocol is a set of rules or standards that two entities must follow so as to allow each other to receive and interpret messages sent to them. The entities could, for example, be two application programs in an application protocol, or the entities might be two TCP protocol layers in two different IP hosts (the TCP protocol).

Figure 1 on page 3 illustrates the TCP/IP protocol stack.

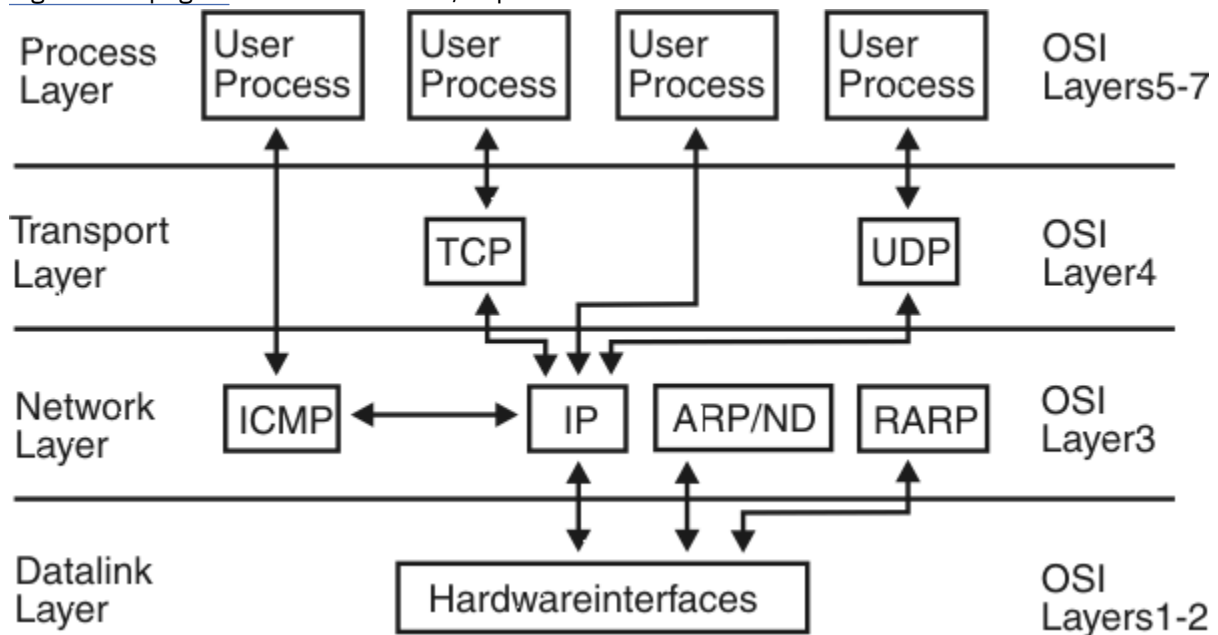


Figure 1. The TCP/IP protocol stack

Programs are located at the process layer; here they can interface with the two transport layer protocols (TCP and UDP), or directly with the network layer protocols (ICMP and IP).

TCP

Transmission Control Protocol is a transport protocol providing a reliable, full-duplex byte stream. Most TCP/IP applications use the TCP transport protocol.

UDP

User Datagram Protocol is a connectionless protocol providing datagram services. UDP is less reliable because there is no guarantee that a UDP datagram ever reaches its intended destination, or that it reaches its destination only once and in the same condition as it was passed to the sending UDP layer by a UDP application.

ICMP

Internet Control Message Protocol is used to handle error and control information at the IP layer. The ICMP is most often used by network control applications that are part of the TCP/IP software product itself, but ICMP can be used by authorized user processes as well. PING and TRACEROUTE are examples of network control applications that use the ICMP protocol.

IP

Internet Protocol provides the packet delivery services for TCP, UDP, and ICMP. The IP layer protocol is unreliable (called a best-effort protocol). There is no guarantee that IP packets arrive, or that they

arrive only once and are error-free. Such reliability is built into the TCP protocol, but not into the UDP protocol. If you need reliable transport between two UDP applications, you must ensure that reliability is built into the UDP applications.

ARP/ND

The IPv4 networking layer uses the Address Resolution Protocol (ARP) to map an IP address into a hardware address. In the IPv6 networking layer, this mapping is performed by the Neighbor Discovery (ND function). On local area networks (LANs), such an address would be called a media access control (MAC) address.

RARP

Reverse Address Resolution Protocol is used to reverse the operation of the ARP protocol. It maps a hardware address into an IPv4 address. Note that both ARP packets and RARP packets are not forwarded in IP packets, but are themselves media level packets. ARP and RARP are not used on all network types, as some networks do not need these protocols.

Understanding sockets concepts

A socket uniquely identifies the endpoint of a communication link between two application ports.

A port represents an application process on a TCP/IP host, but the port number itself does not indicate the protocol being used: TCP, UDP, or IP. The application process might use the same port number for TCP or UDP protocols. To uniquely identify the destination of an IP packet arriving over the network, you have to extend the port principle with information about the protocol used and the IP address of the network interface; this information is called a socket. A socket has three parts: protocol, local-address, local-port.

Figure 2 on page 4 illustrates the concept of a socket.

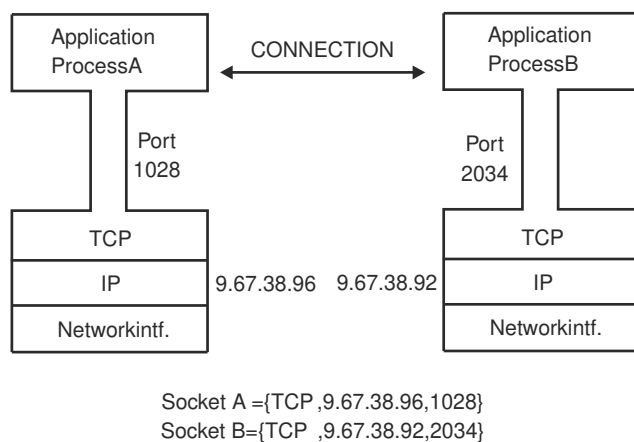


Figure 2. Socket concept

The term *association* is used to specify completely the two processes that comprise a connection: (protocol,local-address,local-port,foreign-address,foreign-port).

The terms *socket* and *port* are sometimes used as synonyms, but note that the terms *port number* and *socket address* are not like one another. A port number is one of the three parts of a socket address, and can be represented by a single number (for example, 1028) while a socket address can be represented by (tcp,myhostname,1028).

A socket descriptor (sometimes referred to as a socket number) is a binary integer that acts as an index to a table of sockets; the sockets are currently allocated to a given process. A socket descriptor represents the socket, but is not the socket itself.

Introducing TCP/IP concepts: Programming with sockets

A socket is an endpoint for communication able to be named and addressed in a network. From the perspective of the application program, it is a resource allocated by the address space; it is represented by an integer called the socket descriptor.

The socket interface was designed to provide applications a network interface that hides the details of the physical network. The interface is differentiated by the different services provided: Stream, datagram, and raw sockets. Each interface defines a separate service available to applications.

The MVS socket APIs provide a standard interface using the transport and internetwork layer interfaces of TCP/IP. These APIs support three socket types: stream, datagram, and raw. Stream and datagram socket types interface with the transport layer protocols; raw socket types interface with the network layer protocols. Choose the most appropriate interface for your application.

Introducing TCP/IP concepts: Selecting sockets

You can choose among the following types of sockets:

- Stream
- Datagram
- Raw

Stream sockets perform like streams of information. There are no record lengths or character boundaries between data, so communicating processes must agree on their own mechanisms for distinguishing information. Usually, the process sending information sends the length of the data, followed by the data itself. The process receiving information reads the length and then loops, accepting data until all of it has been transferred. Because there are no boundaries in the data, multiple concurrent read or write socket calls of the same type, on the same stream socket, will yield unpredictable results. For example, if two concurrent read socket calls are issued on the same stream socket, there is no guarantee of the order or amount of data that each instance will receive. Stream sockets guarantee to deliver data in the order sent and without duplication. The stream socket defines a reliable connection service. Data is sent without error or duplication and is received in the order sent. Flow control is built in to avoid data overruns. No boundaries are imposed on the data; the data is treated as a stream of bytes.

Stream sockets are most common because the burden of transferring the data reliably is handled by TCP/IP, rather than by the application.

The datagram socket is a connectionless service. Datagrams are sent as independent packets. The service provides no guarantees. Data can be lost or duplicated, and datagrams can arrive out of order. The size of a datagram is limited to the size able to be sent in a single transaction. Currently, the default value is 8192 bytes, and the maximum value is 65535. The maximum size of a datagram is 65535 for UDP and 65535 bytes for raw.

The raw socket allows direct access to lower layer protocols, such as IP and the ICMP. This interface is often used to test new protocol implementation, because the socket interface can be extended and new socket types defined to provide additional services. For example, the transaction type sockets can be defined for interfacing to the Versatile Message Transfer Protocol (VMTP). ¹Transaction-type sockets are not supported by TCP/IP. Because socket interfaces isolate you from the communication function of the different protocol layers, the interfaces are largely independent of the underlying network. In the MVS implementation of sockets, stream sockets interface with TCP, datagram sockets interface with UDP, and raw sockets interface with ICMP and IP.

Notes:

1. The TCP and UDP protocols cannot be used with raw sockets.

¹ David R. Cheriton and Carey L. Williamson, "MVSTP as the Transport Layer for High-Performance Distributed Systems," *IEEE Communications*, June 1989, Vol. 27, No. 6.

2. If you are communicating with an existing application, you must use the same protocols used by the existing application. For example, if you communicate with an application that uses TCP, you must use stream sockets.

You should consider the following factors for these applications:

- Reliability

Stream sockets provide the most reliable connection. Datagrams and raw sockets are unreliable because packets can be discarded or duplicated during transmission. This characteristic might be acceptable if the application does not require reliability or if the application implements reliability beyond the socket interface.

- Performance

Overhead associated with reliability, flow control, and connection maintenance degrades the performance of stream sockets so that they do not perform as well as datagram sockets.

- Data Transfer

Datagram sockets limit the amount of data moved in a single transaction. If you send fewer than 2048 bytes of data at one time, use datagram sockets. When the amount of data in a single transaction is greater, use stream sockets.

If you are writing a new protocol to use on top of IP, or if you want to use the ICMP protocol, you must choose raw sockets; but to use raw sockets, you must be authorized by way of RACF® or APF.

Introducing TCP/IP concepts: Socket libraries

Figure 3 on page 6 illustrates the TCP/IP networking API relationship on z/OS.

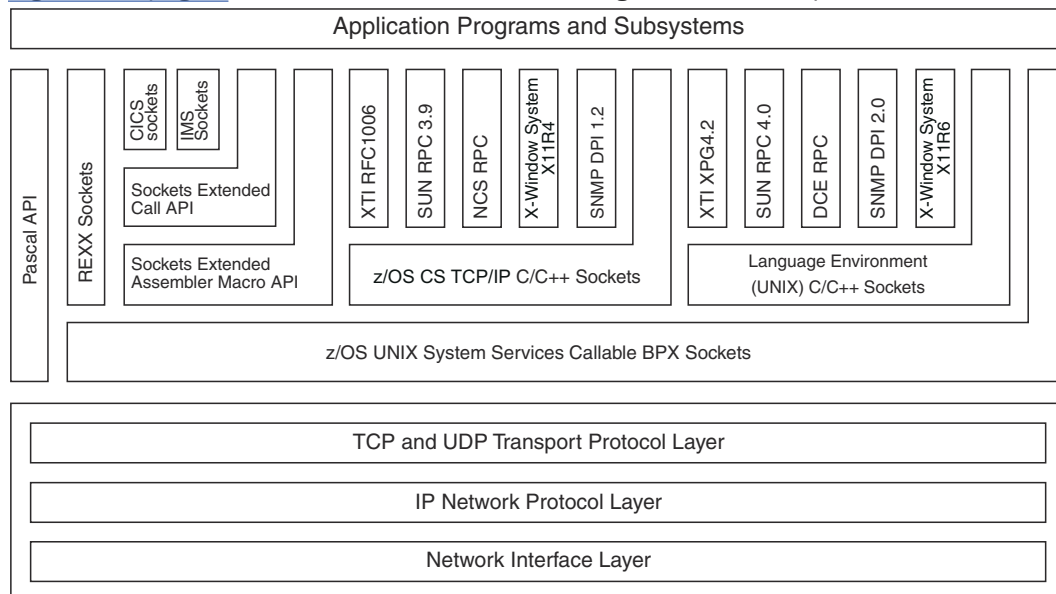


Figure 3. TCP/IP networking API relationship on z/OS

When we create a sockets program, we use something that generally is called a sockets library. A sockets library consists of both compile-time structures, statically linked support modules, and run-time support modules.

There are two main sockets execution environments in z/OS with available libraries:

- Native TCP/IP (implemented by TCP/IP in z/OS Communications Server)
- UNIX (implemented by z/OS UNIX System Services [Language Environment®])

Native TCP/IP

A non-UNIX socket program can use only one TCP/IP protocol stack at a time. The native TCP/IP C socket library is not POSIX compliant and it should not be used for new C socket program development. The non-C native TCP/IP socket libraries (sockets extended: call and assembler macro, REXX sockets, CICS® sockets, and IMS sockets) are available for development of new socket application programs. The following TCP/IP Services APIs are included in this library:

Pascal API

The Pascal IPv4 socket application programming interface enables you to develop TCP/IP applications in the Pascal language. Supported environments are normal MVS address spaces. The Pascal programming interface is based on Pascal procedures and functions that implement conceptually the same functions as the C socket interface. The Pascal routines, however, have different names than the C socket calls. Unlike the other APIs, the Pascal API does not interface with z/OS UNIX System Services; it uses an internal interface to communicate with the TCP/IP protocol stack.

IMS sockets

The Information Management System (IMS) IPv4 socket interface supports development of client/server applications in which one part of the application executes on a TCP/IP-connected host and the other part executes as an IMS application program. The programming interface used by both application parts is the socket programming interface, and the communication protocols are either TCP, UDP, or RAW. For more information, see [z/OS Communications Server: IP IMS Sockets Guide](#).

CICS sockets

The CICS socket interface enables you to write CICS applications that act as IPv4 or IPv6 clients or servers in a TCP/IP-based network. Applications can be written in C language, using the C sockets programming, or they can be written in COBOL, PL/I, or assembler, using the Extended Sockets programming interface. For more information, see [z/OS Communications Server: IP CICS Sockets Guide](#).

z/OS CS TCP/IP C/C++ Sockets

The C/C++ Sockets interface supports IPv4 socket function calls that can be invoked from C/C++ programs.

Note: Use of the UNIX C socket library is encouraged.

Sockets Extended macro API

The Sockets Extended macro API is a generalized assembler macro-based interface to IPv4 and IPv6 socket programming. It includes extensions to the socket programming interface, such as support for asynchronous processing on most sockets function calls.

Sockets Extended Call Instruction API

The Sockets Extended Call Instruction API is a generalized call-based interface to IPv4 and IPv6 sockets programming. The functions implemented in this call interface resemble the C-sockets implementation, with some extensions similar to the sockets extended macro interface.

REXX sockets

The REXX sockets programming interface implements facilities for IPv4 and IPv6 socket communication directly from REXX programs by way of an address rxsocket function. REXX socket programs can execute in TSO, online, or batch.

UNIX

A UNIX socket program can use up to eight TCP/IP protocol stacks at once. The stacks may be a combination of any TCP/IP protocol stack that is supported by z/OS UNIX System Services. The following APIs are provided by the UNIX element of z/OS and are not addressed in detail in this publication:

z/OS C sockets

z/OS UNIX C sockets are used in the z/OS UNIX environment. Programmers use this API to create IPv4 and IPv6 applications that conform to the POSIX or XPG4 standard (a UNIX specification). Applications built with z/OS UNIX C sockets can be ported to and from platforms that support these standards. For more information, see [z/OS C/C++ Runtime Library Reference](#).

z/OS UNIX Assembler Callable Services

z/OS UNIX Assembler Callable Services is a generalized call-based interface to z/OS UNIX IPv4 and IPv6 sockets programming. The functions implemented in this call interface resemble the z/OS UNIX C sockets implementation, with some extensions similar to the sockets extended macro interface. For more information, see [z/OS UNIX System Services Programming: Assembler Callable Services Reference](#).

Introducing TCP/IP concepts: Address families

Address families define different styles of addressing. All hosts in a given address family understand and use the same scheme for addressing socket endpoints. TCP/IP supports addressing families AF_INET and AF_INET6. See [“Introducing TCP/IP concepts: Socket libraries” on page 6](#) to determine which APIs support the AF_INET or both the AF_INET and AF_INET6 address families. The AF_INET domain defines addressing for the IPv4 internet domain. The AF_INET6 domain defines addressing for the IPv6 internet domain.

Introducing TCP/IP concepts: Addressing sockets in an Internet domain

This topic describes how to address sockets in an Internet domain.

Internet addresses

Internet addresses are 32-bit quantities (AF_INET) or 128-bit quantities (AF_INET6) that represent a network interface. Every Internet address within an administered AF_INET domain must be unique. Every Internet address within a *scope* for AF_INET6 domain must be unique. An internet host can also have multiple Internet addresses. In fact, a host has at least as many Internet addresses as it has network interfaces. For IPv4 interfaces, there must be one unique address per interface. However, the same is not true for IPv6 interfaces. See [z/OS Communications Server: IPv6 Network and Appl Design Guide](#) for more information.

Ports

A port is used to differentiate among different applications using the same network interface. It is an additional qualifier used by the system software to get data to the correct application. Physically, a port is a 16-bit integer. Some ports are reserved for particular applications; they are labeled as well-known ports.

In the client/server model, the server provides a resource by listening for clients on a particular port. Some applications, such as FTP and Telnet, are standardized protocols and listen on a well-known port. Such standardized applications use the same port number on all TCP/IP hosts. For your client/server applications, however, you need a way to assign port numbers to represent the services you intend to provide. An easy way to define services and their ports is to enter them into data set *hlq.ETC.SERVICES*. In C, the programmer uses the `getservbyname()` function to determine the port for a particular service. Should the port number for a particular service change, only the *hlq.ETC.SERVICES* data set needs to be modified.

Note: Note that *hlq* is the high-level qualifier. z/OS Communications Server ships with a default *hlq* of *TCPIP*. Use this default or override it using the *DATASETPREFIX* statement in the *PROFILE.TCPIP* and *TCPIP.DATA* configuration files. TCP/IP is shipped with data set *hlq.ETC.SERVICES* that contains the well-known services of FTP and Telnet. Data set *hlq.ETC.SERVICES* is described in the [z/OS Communications Server: IP Configuration Reference](#).

A socket program in an IP host identifies itself to the underlying TCP/IP protocol layers by port number.

A port number is a 16-bit integer ranging from 0 to 65535. A port number uniquely identifies this application to the protocol underlying this TCP/IP host (TCP, UDP, or IP). Other applications in the TCP/IP network can contact this application by way of reference to the port number on this specific IP host.

[Figure 4 on page 9](#) shows the port concept.

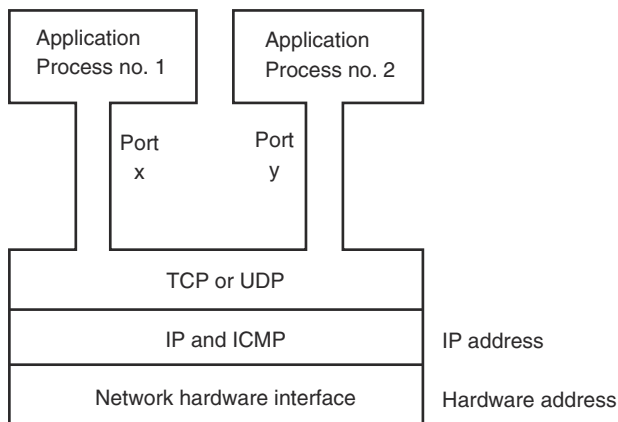


Figure 4. The port concept

Both server applications and client applications have port numbers. A server application uses a specific port number to uniquely identify this server application. The port number can be reserved to a particular server, so no other process ever uses it. In an IBM TCP/IP Services environment, you can do this using the PORT statement in the *hlq.PROFILE.TCP/IP* configuration data set. When the server application initializes, it uses the bind() socket call to identify its port number. A client application must know the port number of a server application in order to contact it.

Because advance knowledge of the client's port number is not needed, a client often leaves it to TCP/IP to assign a free port number when the client issues the connect() socket call to connect to a server. Such a port number is called an ephemeral port number; this means it is a port number with a short life. The selected port number is assigned to the client for the duration of the connection, and is then made available to other processes. It is the responsibility of the TCP/IP software to ensure that a port number is assigned to only one process at a time.

Well-known official Internet port numbers are in the range of 0 - 255. See <http://www.iana.org/assignments/multicast-addresses> for details. In addition, port numbers in the range of 256 - 1023 are reserved for other well-known services. Port numbers in the range of 1024 - 65535, which are neither in-use nor reserved, are used when TCP/IP automatically assigns port numbers to client programs that do not use specific port numbers.

Figure 5 on page 9 shows port number assignments.



Figure 5. Port number assignments

Before you select a port number for your server application, consult the *hlq.ETC.SERVICES* data set. This data set is used to assign port numbers to server applications. The server application can use socket call getservbyname() to retrieve the port number assigned to a given server name. Add the names of your server applications to this data set and use socket call getservbyname(). With this technique, you avoid hard coding the port number into your server program. The client program must know the port number of the server on the server host. There is no socket call to obtain that information from the server host. To compensate, synchronize the contents of data sets ETC.SERVICES on all TCP/IP hosts in your network. Client application can then use the getservbyname() socket call to query its local ETC.SERVICES data set for the port number of the server. Use this technique to develop your own local well-known services.

Network byte order

Ports and addresses are usually specified by calls using the network byte ordering convention. Network byte order is also known as big endian byte ordering, where the high order byte defines significance. Network byte ordering allows hosts using different architectures to exchange address information. See

[“accept\(\) ” on page 94](#), [“bind\(\) ” on page 95](#), [“htonl\(\) ” on page 125](#), [“htons\(\) ” on page 125](#), [“ntohl\(\) ” on page 133](#), and [“ntohs\(\) ” on page 133](#) for more information about network byte order.

Notes:

1. The socket interface does not handle application data byte ordering differences. Application writers must handle byte order differences themselves, or use higher level interfaces such as remote procedure calls (RPC). For description of the RPC calls, see [z/OS Communications Server: IP Programmer's Guide and Reference](#).
2. If you use the socket API, your application must handle the issues related to different data representations on different hardware platforms. For character based data, some hosts use ASCII, while other hosts use EBCDIC. Your application must handle translation between the two representations.

Maximum number of sockets

For most socket interfaces, the maximum number of sockets allowed per each connection between an application and the TCP/IP sockets interface is 65535. The exceptions to this rule are the C sockets interface and the C sockets interface for CICS, where the maximum allowed for both of these interfaces is 2000.

Programmers need to be aware that for an application using a sockets interface that uses Sockets Transform (for example, the EZASMI macro API, the callable EZASOKET API, CICS Sockets, or IMS Sockets) approximately 68 bytes of storage per socket in the application's address space is allocated when the application connects to the TCP/IP sockets interface. Each time a REXX client opens a socket, approximately 208 bytes of storage is allocated. If an application using a sockets interface that uses sockets transform requests 65 535 sockets, then approximately 4.25 MB (65535*68 bytes) of storage in the application's address space is allocated just for the socket array. If a REXX client opens 65535 sockets, then approximately 13 MB (65535*208 bytes) of storage is allocated for the socket chain. The monitoring and processing of this many sockets is also costly in terms of performance and CPU utilization.

The number of sockets that an application can open is also limited by the MAXFILEPROC UNIX System Services parameter in the BPXPRMxx parmlib member. This parameter determines the number of sockets each z/OS UNIX System Services process can have open. Each address space is usually a z/OS UNIX System Services process. Thus, in most cases the combined number of sockets opened by all the applications within an address space is limited to the MAXFILEPROC parameter. If MAXFILEPROC is 65535 and two different applications within the same address space both request 65535 sockets, then the two applications will not be able to concurrently have 65535 sockets open. If one of the applications has 65000 sockets open, then the other application will not be able to have more than 535 sockets open even though it has allocated 65535 sockets.

The number of sockets that an application can open in a particular addressing family is also limited by the MAXSOCKETS parameter in BPXPRMxx parmlib member's NETWORK statement that corresponds to the addressing family. This value determines how many sockets for a particular addressing family can be opened in the entire system. If MAXSOCKETS for the AF_INET addressing family is set to 60000 and there are already 50000 AF_INET sockets open in the system, then a new application will not be able to open more than 10000 AF_INET sockets even if it requests a higher number when it connects to the TCP/IP sockets interface.

For details on the BPXPRMxx member, see the following publications:

- [z/OS UNIX System Services Planning](#)
- [z/OS MVS Initialization and Tuning Reference](#)
- [z/OS UNIX System Services File System Interface Reference](#)

AF_INET socket addresses in an Internet domain

A socket address in an Internet addressing family is comprised of four fields:

- The address family (AF_INET)
- The Internet address

- A port
- A character array

The structure of an Internet socket address is defined by the following *sockaddr_in* structure, which is found in header file IN.H:

```
struct in_addr
{
    u_long s_addr;
};
struct sockaddr_in
{
    short    sin_family;
    u_short  sin_port;
    struct in_addr sin_addr;
    char     sin_zero[8];
};
```

The *sin_family* field is set to AF_INET. The *sin_port* field is the port used by the application, in network byte order. *sin_addr* field specifies a 32-bit Internet address. The *sin_addr* field is the Internet address of the network interface used by the application; it is also in network byte order. The *sin_zero* field should be set to zeros.

AF_INET6 socket addresses in an Internet domain

See [z/OS Communications Server: IPv6 Network and Appl Design Guide](#) for parts of the AF_INET6 family. The structure of an Internet socket address is defined by the following *sockaddr_in6* structure, which is found in header file IN.H:

```
struct in6_addr
{
    union
    {
        uint8_t      S6_u8[16];
        uint32_t     S6_u32[4];
    }
    _S6_un;
};

#define s6_addr_S6_un._S6_u8

#define SIN6_LEN

struct sockaddr_in6
{
    uint8_t      sin6_len;
    sa_family_t  sin6_family;
    in_port_t    sin6_port;
    uint32_t     sin6_flowinfo;
    struct in6_addr sin6_addr;
    uint32_t     sin6_scope_id;
};
```

The *sin6_family* field is set to AF_INET6. The *sin6_port* field is a halfword binary field that is the port used by the application, in network byte order. The *sin6_addr* field specifies a 128-bit Internet address. The *sin6_addr* field is the Internet address of the network interface used by the application; it is also in network byte order. The *sin6_flowinfo* field is a fullword binary field specifying the traffic class and flow label. This field is currently not implemented. The *sin6_scope_id* field identifies a set of interfaces as appropriate for the scope of the address carried in the *sin6_addr* field.

Chapter 2. Organizing a TCP/IP application program

This information explains how to organize a TCP/IP application program. All examples are shown using an address family of AF_INET (IPv4). All concepts described below can also be applied to an address family of AF_INET6 (IPv6).

- Client and server socket programs
- Call sequence in socket programs
- Blocking, nonblocking, and asynchronous socket calls
- Testing a program using a miscellaneous server
- Testing a local machine using a loopback address
- Accessing required data sets

Client and server socket programs

The terms *client* and *server* are common within the TCP/IP community, and many definitions exist. In the TCP/IP context, these terms are defined as follows:

Server

A process that waits passively for requests from clients, processes the work specified, and returns the result to the client that originated the request.

Client

A process that initiates a service request.

No matter which socket programming interface you select, function is identical. The syntax might vary, but the underlying concept is the same.

While clients communicate with one server at a time, servers can serve multiple clients. When you design a server program, plan for multiple concurrent processes. Special socket calls are available for that purpose; they are called *concurrent servers*, as opposed to the more simple type of *iterative server*.

To distinguish between these generic socket program categories, the following terms are used:

- **Client program** identifies a socket program that acts as a client.
- **Iterative server program** identifies a socket program that acts as a server, and processes fully one client request before accepting another client request.
- **Concurrent server main program** identifies that part of a concurrent server that manages child processes, accepts client connections, and schedules client connections to child processes.
- **Concurrent server child program** identifies that part of a concurrent server that processes the client requests.

In a concurrent server main program, the child program might be active in many parallel child processes, each processing a client request. In an MVS environment, a process is either an MVS task, a CICS transaction, or an IMS transaction.

Client/server socket programs: Iterative server socket programs

An iterative server processes requests from clients in a serial manner; one connection is served and responded to before the server accepts a new client connection.

[Figure 6 on page 14](#) shows the iterative server main logic.

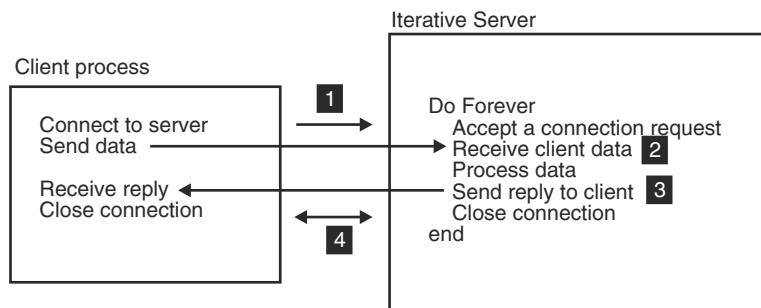


Figure 6. Iterative server main logic

The following list describes the iterative server socket process.

1. When a connection request arrives, it accepts the connection and receives the client data.
2. The iterative server processes the received data and does whatever has to be done to build a reply.
3. The server sends the data back to the client.
4. The iterative server closes the socket and waits for the next connection request from the network.

An MVS iterative server can be implemented as follows:

- As a batch job or MVS task started manually, or by automation software. The job remains active until it is closed by operator intervention.
- As a TSO transaction. For a production implementation, submit a job that executes a batch terminal monitor program (TMP).
- As a long-running CICS task. The task normally begins during CICS startup, but it can be started by an authorized CICS operator entering the appropriate CICS transaction code.
- As a batch message program (BMP) in IMS.

From a socket programming perspective, there is no difference between an iterative server that runs in a native MVS environment (batch job, started task, or TSO) and one that runs as a CICS task, or as a BMP under IMS.

You can terminate the server process in various ways. For jobs that execute in traditional MVS address spaces (batch job, started task, TSO, IMS BMP), you can implement functions in the server to enable an operator to use the MVS MODIFY command to signal stop; for example F SERVER,STOP. (This technique cannot be used for CICS tasks.) Alternatively, you can include a shutdown message in the application protocol. By doing so, you can develop a shutdown client program that connects to the server and sends a shutdown message. When the server receives a shutdown message from a socket client, it terminates itself.

Client/server socket programs: Concurrent server socket programs

A concurrent server accepts a client connection, delegates the connection to a child process of some kind, and immediately signals its availability to receive the next client connection.

The following list describes the concurrent server process.

1. When a connection request arrives in the main process of a concurrent server, it schedules a child process and forwards the connection to the child process.
2. The child process takes the connection from the main process.
3. The child process receives the client request, processes it, and returns a reply to the client.
4. The connection is closed, and the child process terminates or signals to the main process that it is available for a new connection.

You can implement a concurrent server in the following MVS environments:

- Native MVS (batch job, started task, or TSO). In this environment you implement concurrency by using traditional MVS subtasking facilities. These facilities are available from assembler language programs or from high-level languages that support multitasking or multithreading; for example, C/370.
- CICS. The concurrent main process is started as a long-running CICS task that accepts connection requests from clients, and initiates child processes by way of the EXEC CICS START command. CICS sockets include a generic concurrent server main program called the CICS LISTENER.
- IMS. The concurrent main process is started as a BMP that accepts connection requests from clients and initiates child processes by way of the IMS message switch facilities. The child processes execute as IMS message processing programs (MPP). IMS sockets include a generic concurrent server main program called the IMS LISTENER.

In the iterative and concurrent server scenarios described above, client and server processes could have exchanged a series of request and reply sequences before closing the connection.

Client/server socket programs: Call sequence in socket programs

The following topics describe call sequence concepts for different types of socket sessions.

Call sequence in stream socket sessions

This topic describes a typical stream socket session.

Use stream sockets for both passive (server) and active (client) processes. While some calls are necessary for both types, others are role specific. See [“Sample C socket programs” on page 164](#) for sample socket communication client and server programs. All connections exist until closed by the socket. During the connection, data is delivered, or an error code is returned by TCP/IP.

[Figure 7 on page 16](#) shows the general sequence of calls for most socket routines using stream sockets.

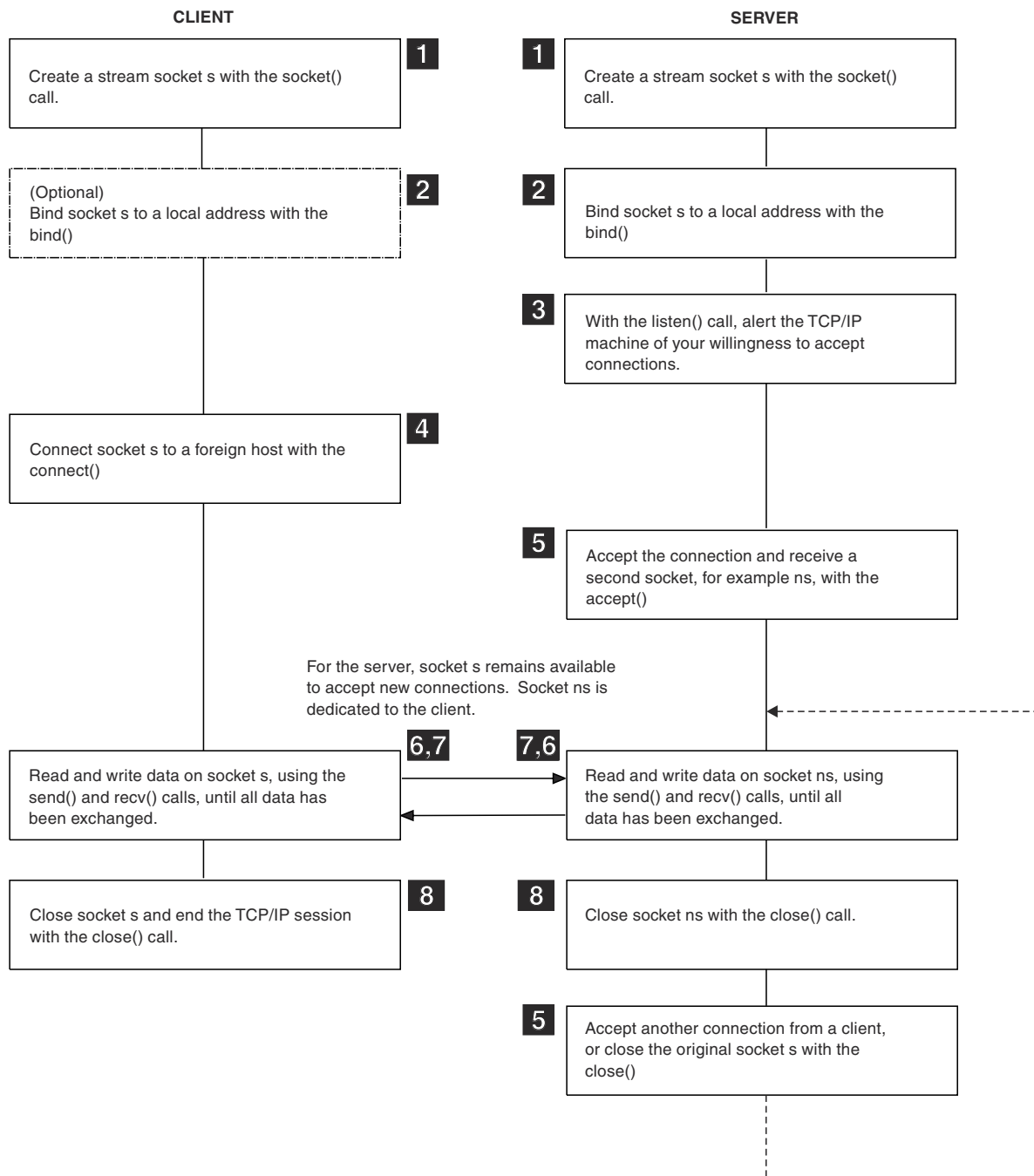


Figure 7. A typical stream socket session

Call sequence in datagram socket sessions

Datagram socket processes, unlike stream socket processes, are not clearly distinguished by server and client roles. The distinction lies in connected and unconnected sockets. An unconnected socket can be used to communicate with any host, but a connected socket can send data to and receive data from one host only.

Both connected and unconnected sockets transmit data without verification. After a packet has been accepted by the datagram interface, neither its integrity nor its delivery can be assured.

Figure 8 on page 17 shows the general sequence of calls for socket routines using datagram sockets.

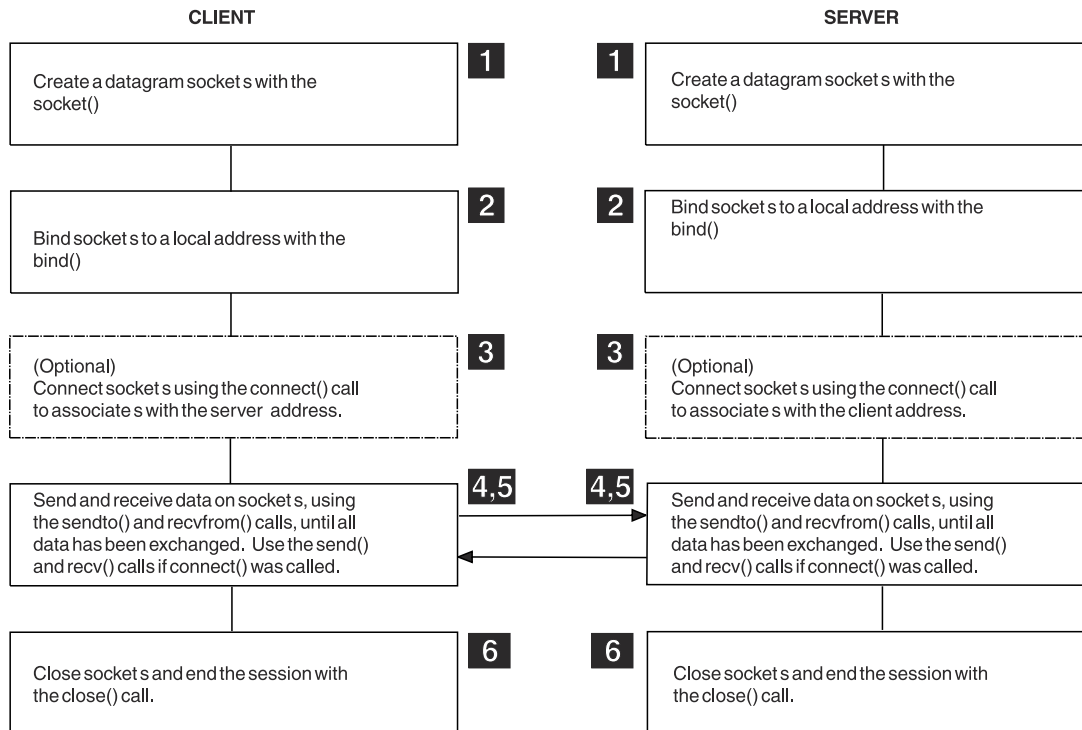


Figure 8. A typical datagram socket session

Client/server socket programs: Blocking, nonblocking, and asynchronous socket calls

A socket is in blocking mode when an I/O call waits for an event to complete. If the blocking mode is set for a socket, the calling program is suspended until the expected event completes.

If nonblocking is set by the `FCNTL()` or `IOCTL()` calls, the calling program continues even though the I/O call might not have completed. If the I/O call could not be completed, it returns with `ERRNO EWOULDBLOCK`. (The calling program should use `SELECT()` to test for completion of any socket call returning an `EWOULDBLOCK`.)

Note: The default mode is blocking.

If data is not available to the socket, and the socket is in blocking and synchronous modes, the `READ` call blocks the caller until data arrives.

All IBM TCP/IP Services socket APIs support nonblocking socket calls. Some APIs, in addition to nonblocking calls, support asynchronous socket calls.

Blocking

The default mode of socket calls is blocking. A blocking call does not return to your program until the event you requested has been completed. For example, if you issue a blocking `recvfrom()` call, the call does not return to your program until data is available from the other socket application. A blocking `accept()` call does not return to your program until a client connects to your socket program.

Nonblocking

Change a socket to nonblocking mode using the `ioctl()` call that specifies command `FIONBIO` and a fullword (four byte) argument with a nonzero binary value. Any succeeding socket calls against the involved socket descriptor are nonblocking calls.

Alternatively, use the `fcntl()` call using the `F_SETFL` command and `FNDELAY` as an argument.

Nonblocking calls return to your program immediately to reveal whether the requested service was completed. An error number may mean that your call would have blocked had it been a blocking call.

If the call was, for example, a `recv()` call, your program might have implemented its own wait logic and reissued the nonblocking `recv()` call at a later time. By using this technique, your program might have implemented its own timeout rules and closed the socket, failing receipt of data from the partner program, within an application-determined period of time.

A new `ioctl()` call can be used to change the socket from nonblocking to blocking mode using command `FIONBIO` and a fullword argument of value 0 ('0').

Asynchronous

Like nonblocking calls, asynchronous calls return control to your program immediately. But in this case, there is no need to reissue the call. Asynchronous calls are available with the macro API. For more information, see [“Task management and asynchronous function processing”](#) on page 198.

Table 1 on page 18 lists the actions taken by the socket programming interface.

<i>Table 1. Socket programming interface actions</i>			
Call type	Socket state	blocking	Nonblocking
Types of <code>read()</code> calls	Input is available	Immediate return	Immediate return
	No input is available	Wait for input	Immediate return with <code>EWOULDBLOCK</code> error number (select() exception: <code>READ</code>)
Types of <code>write()</code> calls	Output buffers available	Immediate return	Immediate return
	No output buffers available	Wait for output buffers	Immediate return with <code>EWOULDBLOCK</code> error number (select() exception: <code>WRITE</code>)
<code>accept()</code> call	New connection	Immediate return	Immediate return
	No connections queued	Wait for new connection	Immediate return with <code>EWOULDBLOCK</code> error number (select() exception: <code>READ</code>)
<code>connect()</code> call		Wait	Immediate return with <code>EINPROGRESS</code> error number (select() exception: <code>WRITE</code>)

Test pending activity on a number of sockets in a synchronous program by using the `select()` call. Pass the list of socket descriptors that you want to test for activity to the `select()` call; specify by socket descriptor the following type of activity you want test to find:

- Pending data to read
- Ready for new write
- Any exception condition

When you use `select()` call logic, you do not issue any socket call on a given socket until the `select()` call tells you that something has happened on that socket; for example, data has arrived and is ready to be read by a `read()` call. By using the `select()` call, you do not issue a blocking call until you know that the call cannot block.

The `select()` call can itself be blocking, nonblocking, or, for the macro API, asynchronous. If the call is blocking and none of the socket descriptors included in the list passed to the `select()` call have had any activity, the call does not return to your program until one of them has activity, or until the timer value passed on the `select()` call expires.

The `select()` call and `selectex()` call are available. The difference between `select()` and `selectex()` calls is that `selectex()` call allows you to include nonsocket related events in the list of events that can trigger the `selectex()` call to complete. You do so by passing one or more MVS event control blocks (ECBs) on the `selectex()` call. If there is activity on any of the sockets included in the select list, if the specified timer expires, or if one of the external events completes, the `selectex()` call returns to your program.

Typically, a server program waits for socket activity or an operator command to shut it down. By using the `selectex()` call, a shutdown ECB can be included in the list of events to be monitored for activity.

Client/server socket programs: Testing a program using a miscellaneous server

To test your program using either a stream or a datagram socket session, you can use the MISCSERV server. You must start MISCSERV before a client application can connect to it. If Ports 7, 9, or 19 are used by another application, or using another copy of MISCSERV, this MISCSERV command cannot operate properly. Available MISCSERV servers are:

Tool

Server description

Echo

Specify Port 7 when you want MISCSERV to return data exactly as it is received (stream and datagram sessions).

Discard

Specify Port 9 when you want MISCSERV to discard the data.

Character Generator

Specify Port 19 when you want MISCSERV to return random data regardless of the data it receives. For a stream session, data is returned continuously until you end the session; the received data stream is discarded. For a datagram session, random data is returned for each datagram received; the received datagram is discarded.

Note: The server uses MAXSOC=50. This value limits the sockets available to the server.

For more information, see RFC 862, RFC 863, RFC 864, and the [z/OS Communications Server: IP Configuration Reference](#).

Client/server socket programs: Testing a local machine using a loopback address

You can use a local loopback address to test your local TCP/IP host without accessing the network. For the AF_INET family, the class A network address 127.0.0.1 is the default loopback address. For AF_INET6, the network address ::1 is the default loopback address. Depending on the address family, you can specify 127.0.0.1 (AF_INET) or ::1 (AF_INET6). Additional loopback addresses can be configured by your TCP/IP administrator.

You can use the loopback address with any TCP/IP command that accepts IP addresses, although you might find it particularly useful in conjunction with FTP and PING commands. When you issue a command with a loopback address, the command is sent from your local host client to the local TCP/IP host where it is recognized as a loopback address and is sent to your local host server.

Using a loopback address on commands allows you to test client and server functions on the same host for proper operation.

Note: Any command or data that you send using the loopback address never actually leaves your local TCP/IP host.

The information you receive reflects the state of your system and tests the client and server code for proper operation. If the client or server code is not operating properly, a command message is returned.

Client/server socket programs: Accessing required data sets

Table 2 on page 20 lists the data sets and applications to which TCP/IP applications must have access to compile and link-edit.

Table 2. TCP/IP data sets and applications

Data set	Application
SEZACMAC	Client Pascal macros, C headers, and assembler macros
SEZACMTX	Sockets and Pascal API
SEZADPIL	SNMP DPI
SEZALIBN	NCS
SEZAOLDX	X Release 10 compatibility routines
SEZANMAC	C headers and assembler macros for z/OS UNIX and TCP/IP Services APIs
SEZARNT1	Sockets, X11, and PEXlib (reentrant)
SEZARNT2	Athena widget (reentrant)
SEZARNT3	Motif widget (reentrant)
SEZARPCL	Remote procedure calls
SEZAXAWL	Athena widget set
SEZAXMLB	OSF/Motif-based widget set
SEZAXTLB	Xt Intrinsics
SEZAX11L	Xlib, Xmu, Xext, and Xau routines

Part 2. Designing programs

Describes ways to design various types of programs including the following information:

- [Chapter 3, “Designing an iterative server program,” on page 23](#)
- [Chapter 4, “Designing a concurrent server program,” on page 33](#)
- [Chapter 5, “Designing a client program,” on page 47](#)
- [Chapter 6, “Designing a program to use datagram sockets,” on page 49](#)
- [Chapter 7, “Transferring data between sockets,” on page 51](#)
- [Chapter 8, “Designing IPv6 programs ,” on page 61](#)
- [Chapter 9, “Designing multicast programs ,” on page 63](#)

Chapter 3. Designing an iterative server program

This information uses an address family of AF_INET (IPv4). All of the concepts described can also be applied to an address family of AF_INET6 (IPv6).

- [“Allocating sockets in an iterative server program” on page 23](#)
- [“Binding sockets in an iterative server program” on page 25](#)
- [“Listening for client connection requests in an iterative server program” on page 27](#)
- [“Accepting client connection requests in an iterative server program” on page 28](#)
- [“Transferring data between sockets in an iterative server program” on page 30](#)
- [“Closing a connection in an iterative server program” on page 30](#)

Allocating sockets in an iterative server program

The server must allocate a socket to provide an endpoint to which clients connect. All commands that pass a socket address must be consistent with the address family specified when the socket was opened.

- If the socket was opened with an address family of AF_INET, then any command for that socket that includes a socket address must use an AF_INET socket address.
- If the socket was opened with AF_INET6, then any command for that socket that includes a socket address must use an AF_INET6 socket address.

A socket is actually an index into a table of connections to the TCP/IP address space, so socket numbers are usually assigned in ascending order. In C, the programmer issues the socket() call to allocate a new socket, as shown in the following example:

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

The socket function requires specification of the address family (AF_INET), the type of socket (SOCK_STREAM), and the particular networking protocol to be used. When 0 is specified, the TCP/IP address space automatically uses the protocol appropriate to the socket type specified. A new socket is allocated and returned.

An application must first get a socket descriptor using the socket() call, as seen in the following example. For a complete description, see [“socket\(\) ” on page 158](#).

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

The code fragment allocates socket descriptor s in the internet addressing family. The domain parameter is a constant that specifies the domain in which the communication is taking place. A domain is a collection of applications using a single addressing convention. The type parameter is a constant that specifies the type of socket; it can be SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW. The protocol parameter is a constant that specifies the protocol to be used. This parameter is ignored, unless type is set to SOCK_RAW. Passing 0 chooses the default protocol. If successful, the socket() call returns a positive integer socket descriptor.

The server obtains a socket by way of the socket call. You must specify the domain to which the socket belongs, and the type of socket you want.

[Figure 9 on page 24](#) lists the socket call() variables using the CALL API.

```

*-----*
* Variables used for the SOCKET call                                     *
*-----*
01  afinet                      pic 9(8) binary value 2.
01  soctype-stream              pic 9(8) binary value 1.
01  proto                      pic 9(8) binary value 0.
01  socket-descriptor          pic 9(4) binary value 0.
*-----*
* Get us a socket descriptor                                           *
*-----*
      call 'EZASOCKET' using socket-socket
      afinet
      soctype-stream
      proto
      errno
      retcode.
      if retcode < 0 then
        - process error -
      else
        Move retcode to socket-descriptor.

```

Figure 9. Socket call variables

The internet domain has a value of 2. A stream socket is requested by passing a type value of 1. The proto field is normally 0, which means that the socket API should choose the protocol to be used for the domain and socket type requested. In this example, the socket uses TCP protocols.

A socket descriptor representing an unnamed socket is returned from the socket() call. An unnamed socket has no port and no IP address information associated with it; only protocol information is available. The socket descriptor is a 2-byte binary field and must be passed on subsequent socket calls as such.

A socket is an inconvenient concept for a program because it consists of three different items: a protocol specification, a port number, and an IP address. To represent the socket conveniently, we use the socket descriptor.

The socket descriptor is not in itself a socket, but represents a socket and is used by the socket library routines as an index into the table of sockets owned by a given MVS TCP/IP client. On all socket calls that reference a specific socket, you must pass the socket descriptor that represents the socket with which you want to work.

Figure 10 on page 24 lists the MVS TCP/IP socket descriptors.

Socket Descriptor	Socket
0	Our listen socket
1	Our connected socket

Figure 10. MVS TCP/IP socket descriptor table

The first socket descriptor assigned to your program is 0 (for a sockets extended program). If your program is written in C, socket descriptors 0, 1, and 2 are reserved for std.in, std.out and std.err, and the first socket descriptor assigned for your AF_INET sockets is numeral 3 or higher.

When a socket is closed, the socket descriptor becomes available; it is returned as a new socket descriptor representing a new socket in response to a succeeding request for a socket.

Note: In reference documentation, the socket descriptor is normally represented by a single letter: S, or by two letters: SD.

When you possess the socket descriptor, you can request the socket address structure from the socket programming interface by way of call getsockname(). A socket does not include both port and IP addresses until after a successful bind(), connect(), or accept() call has been issued.

If your socket program is capable of handling sockets simultaneously, you must keep track of your socket descriptors. Build a socket descriptor table inside of your program to store information related to the socket and the status of the socket. This information is sometimes needed, and can help in debug situations.

Binding sockets in an iterative server program

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

- The socket just given to the server.
- The number of the port to which the server is to provide service.
- The IP address of the network connection from which the server is to accept connection. If this address is 0, the server accepts connection requests from any address.

Binding with a known port number

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

```
bind(s, (struct sockaddr *)&x, sizeof(struct sockaddr));
```

After an application possesses a socket descriptor, it can explicitly `bind()` a unique address to that socket, as in the example listed in [Figure 11 on page 25](#). For more information about binding, see [“bind\(\)” on page 95](#).

```
int bind(int s, struct sockaddr *name, int namelen);
.
.
.
int rc;
int s;
struct sockaddr_in myname;

/* clear the structure to clear the sin_zero field */
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr = inet_addr("129.5.24.1"); /* specific interface */
myname.sin_port = htons(1024);
:
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
```

Figure 11. An application using the `bind()` call

This example binds socket descriptor `s` to the address `129.5.24.1`, and port `1024` to the internet domain. Servers must bind to an address and port to be accessible to the network. The example in [Figure 11 on page 25](#) lists two utility routines:

- Socket call `inet_addr()` takes an internet address in dotted decimal form and returns it in network byte order. For a description, see [“inet_addr\(\)” on page 126](#).
- Socket call `htons()` takes a port number in host byte order and returns the port number in network byte order. For a description, see [“htons\(\)” on page 125](#).

Binding using socket call `gethostbyname`

[Figure 12 on page 26](#) shows another example of socket call `bind()`. It uses the utility routine `gethostbyname()` to find the internet address of the host, rather than using socket call `inet_addr` with a specific address.

```

int bind(int s, struct sockaddr *name, int namelen);
.
.
.
int rc;
int s;
char *hostname = "myhost";
struct sockaddr_in myname;
struct hostent *hp;

    hp = gethostbyname(hostname);

    /* clear the structure to clear the sin_zero field */
    memset(&myname,0,sizeof(myname));
    myname.sin_family = AF_INET;
    myname.sin_addr.s_addr = *((unsigned long *)hp->h_addr);
    myname.sin_port = htons(1024);
.
rc = bind(s,(struct sockaddr *) &myname, sizeof(myname));

```

Figure 12. A `bind()` call using `gethostbyname()`

Binding a socket to a specific port number

By binding the socket to a specific port number, you avoid having an ephemeral port number assigned to the socket.

Servers find it inconvenient to have an ephemeral port number assigned, because clients have to connect to a different port number for every instance of the server. By using a predefined port number, clients can be developed to always connect to a given port number.

Client programs can use the socket call `bind()`, but client programs rarely benefit from using the same port number every time they execute.

Figure 13 on page 26 shows a list of BIND call variables.

```

*-----*
* Variables used for the BIND Call                                     *
*-----*
01  server-socket-address.
05  server-afinet              pic 9(4) binary value 2.
05  server-port               pic 9(4) binary value 9998.
05  server-ipaddr             pic 9(8) binary value 0.
05  filler                    pic x(8) value low-value.
01  socket-descriptor         pic 9(4) binary.
*-----*
* Bind socket to our server port number                               *
*-----*
    call 'EZASOKET' using socket-bind
        socket-descriptor
        server-socket-address
        errno
        retcode.
    if retcode < 0 then
        - process error -

```

Figure 13. Variables used for the BIND call

Before you issue this call, you must build a socket address structure for your own socket using the following information:

- The address family is two, indicating `AF_INET`. See [z/OS Communications Server: IPv6 Network and Appl Design Guide](#) for a description of binding to an `AF_INET6` socket.
- Port number for your server application. For a sockets extended program, you have to create a predefined port number; this is either a constant in your program, or a variable passed to your program as an initialization parameter. If you develop your socket program in C, you can issue a `getservbyname()` call to locate the port number reserved for your server application in data set `hlq.ETC.SERVICES`.
- IP address on which your server application is to accept incoming requests. If your application is executing on a multihomed host, and you want to accept incoming requests over all available network interfaces, you must set this field to binary zeros.

- For TCP connections, 0 allows a server to accept incoming connections to the specified port regardless of which destination IP address for this host is used.
- For UDP, 0 allows a server to receive all datagrams destined for the specified port and any destination address for this host.
- For TCP and UDP client applications, specifying a 0 address for the `BIND()` indicates that TCP/IP will select the source IP address to be used.

Normally, the IP address is set to `INADDR_ANY`, but there are situations in which you might want to use a specific IP address. Consider the case of a TCP/IP system address space having been configured with two virtual IP addresses (VIPA). One VIPA address is returned by the named server when clients resolve one host name, and the other VIPA address is returned by the name server when clients resolve the other host name. In fact, both host names represent the same TCP/IP system address space, but the host names can be used to represent two different major socket applications on that MVS host. If your Server A and your Server B can generate a very high amount of network traffic, your network administrator might want to implement what is known as session traffic splitting. This means that IP traffic for one server comes in on one network adapter while traffic for the other server comes in on another adapter. To facilitate such a setup, you must be able to bind the server listener socket to one of the two VIPA addresses.

At this point in the process, you have not told TCP/IP anything about the purpose of the socket you obtained. You are free to use it as a client to issue connect requests to servers in the IP network, or use it to become a server yourself. In terms of the socket, it is, at the moment, active; this is the default status for a newly created socket.

Listening for client connection requests in an iterative server program

After the bind is issued, the server has been specified a particular IP address and port. It now must notify the TCP/IP address space 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 for pending connections. In passive open mode, the socket is open to client contact. For example:

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

The server gives to the socket on which it will be listening 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.

When you call `listen`, you inform TCP/IP that you intend to be a server and accept incoming requests from the IP network. By doing so, socket status is changed from active status to passive.

A passive socket does not initiate a connection; it waits for clients to connect to it.

The `listen()` call variables are shown in [Figure 14 on page 27](#).

```
*-----*
* Variables used by the Listen Call                               *
*-----*
01 backlog-queue          pic 9(8) binary value 10.
01 socket-descriptor      pic 9(4) binary.
*-----*
* Issue passive open via Listen call                               *
*-----*
call 'EZASOCKET' using socket-listen
socket-descriptor
backlog-queue
errno
retcode.
if retcode < 0 then
- process error -
```

Figure 14. Variables used by the listen call

The backlog queue value is used by the TCP/IP system address space when a connect request arrives and your server program is busy processing the previous client request. TCP/IP queues new connection

requests to the number you specify in the backlog queue parameter. If additional connection requests arrive, they are silently ignored by TCP/IP, since there is a limit to the size of the backlog queue parameter.

The system-wide limit is set in the TCP/IP system address space PROFILE.TCP/IP configuration data set by parameter SOMAXCONN. The default value of SOMAXCONN is ten, but you can configure it higher as follows:

```
;
; *****
; * Set the listen queue to a maximum of 100 *
; *****
;
SOMAXCONN 100
```

The value you specify on the listen() call in the backlog parameter cannot exceed the value set for SOMAXCONN in TCPIP.PROFILE. If you specify a backlog parameter of 200 and SOMAXCONN is set to 20, no error is returned, but your backlog queue size will be set to 20 instead of the 200 you requested.

There is a C header file called SOCKET.H (datasetprefix.SEZACMAC member SOCKET) in which there is a variable called SOMAXCONN. The shipped value of this variable is 10, as illustrated below:

```
/*
 *Maximum queue length specifiable by listen
 */
#define SOMAXCONN 10
```

The listen () call does not establish connections; it merely changes the socket to a passive state, so it is prepared to receive connection requests coming from the IP network. If a connection request for this server arrives between the time of the listen() call and the succeeding accept() call, it is queued according to the backlog value passed on the listen() call.

Accepting client connection requests in an iterative server program

To this point in the process, 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 connect with a client. The accept() call blocks the server until a connection request arrives; if there are connection requests in the backlog queue, a connection is established with the first client in the queue. The following example shows the accept() call:

```
client_sock = accept(s);
```

The server passes its socket to the accept call. When the connection is established, the accept call returns a new socket representing the connection with the client. When the server wants to communicate with the client, or to end the connection, it uses this new socket, client_sock. The original socket s is now ready to accept connection to other clients. The original socket is still allocated, bound, and passively opened. To accept another connection, the server calls accept() again. By repeatedly calling accept(), the server can establish simultaneous sessions with multiple clients.

The accept() call dequeues the first queued connection request or blocks the caller until a connection request arrives over the IP network.

The accept() call uses the variables listed in [Figure 15 on page 29](#).


```

*-----*
* Variables used by the ACCEPT Call *
*-----*
01  client-socket-address.
    05  client-afinet          pic 9(4) binary value 0.
    05  client-port           pic 9(4) binary value 0.
    05  client-ipaddr         pic 9(8) binary value 0.
    05  filler                 pic x(8) value low-value.
01  accepted-socket-descriptor pic 9(4) binary value 0.
01  socket-descriptor         pic 9(4) binary.
*-----*
* Start iterative server loop with a blocking Accept Call *
*-----*
call 'EZASOKET' using socket-accept
    socket-descriptor
    client-socket-address
    errno
    retcode.
if retcode < 0 then
    - process error -
else
    Move retcode to accepted-socket-descriptor.

```

Figure 15. Variables used by the ACCEPT call

This call works with the following socket descriptors:

- The first socket descriptor represents the socket that was obtained, bound to the server port and (optionally) the IP address, and changed to the passive state using the listen() call.
- The accept() call returns a new socket descriptor, to represent a complete association:

```

Accepted_socket_descriptor represents:
{TCP, server IP address, server port, client IP address, client port}

```

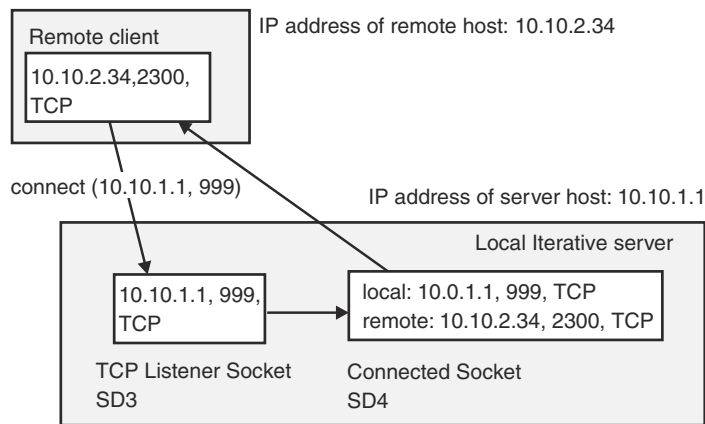
The original socket, which was passed to the accept() call, is unchanged and is still representing our server half association only:

```

Original_socket_descriptor represents:
{TCP, server IP address, server port}

```

When control returns to your program, the socket address structure passed on the call has been filled with the socket address information of the connecting client. [Figure 16 on page 30](#) illustrates the socket states.



Socket Descriptor Table for the local Iterative server

Descriptor#	Local part (IP addr, port, protocol)	Remote part (IP addr, port, protocol)
SD3	10.0.1.1, 999, TCP	
SD4	10.0.1.1, 999, TCP	10.10.2.34, 2300, TCP

Figure 16. Socket states

When a socket is created, we know the protocol that we are going to use with this socket, but nothing else. When a server calls the `bind()` function, a local address is assigned to the socket, but the socket still only represents a half-association; the remote address is still empty. When the client connects to the listener socket and a new socket is created, this new socket represents a fully bound socket possessing both a local address (that of the listener socket) and a remote address (that of the client socket). [Figure 16 on page 30](#) illustrates a fully bound socket.

Subsequent socket calls for the exchange of data between the client and the server use the new socket descriptor. The original socket descriptor remains unused until the iterative server has finished processing the client request and closed the new socket. The iterative server then reissues the `accept()` call using the original socket descriptor and waits for a new connection.

Transferring data between sockets in an iterative server program

See [Chapter 7, “Transferring data between sockets,” on page 51](#).

Closing a connection in an iterative server program

Closing a socket imposes some problems because the TCP protocol layer must ensure that all data has been successfully transmitted and received before the socket resources can be safely freed at both ends.

The following topics describe various ways to close a connection.

Active and passive closing in an iterative server program

The program that initiates the closedown process by issuing the first `close()` call is said to initiate an active close. The program that closes in response to the initiation is said to initiate a passive close.

[Figure 17 on page 31](#) illustrates socket closing.

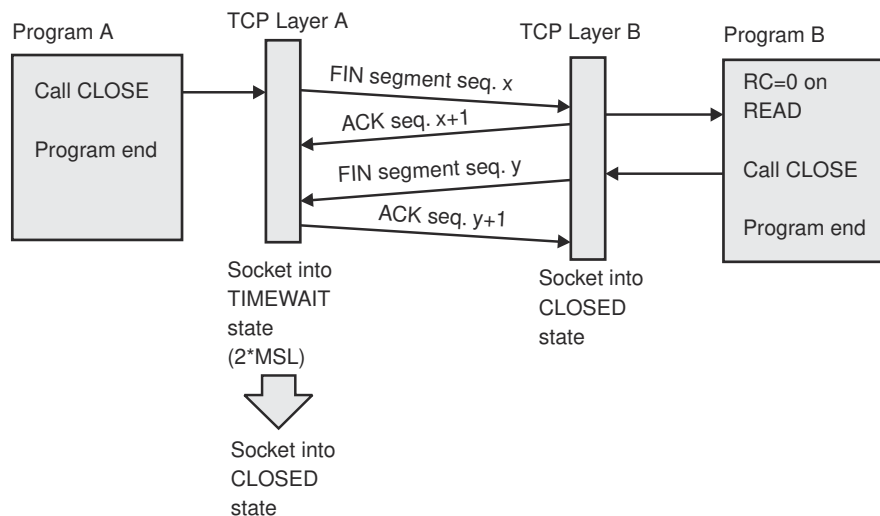


Figure 17. Closing sockets

In Figure 17 on page 31, Program A initiates the active close, while Program B initiates the passive close. When a program calls the close socket function, the TCP protocol layer sends a segment known as FIN (FINish). When Program B receives the final acknowledgment segment, it knows that all data has been successfully transferred and that Program A has received and processed the FIN segment. The TCP protocol layer for Program B can then safely remove the resources that were occupied by the Program socket. The TCP protocol layer for Program A sends an acknowledgment to the FIN segment it received from Program B, but the Program A TCP protocol layer does not know whether that ACK segment arrived at the Program B TCP protocol layer. It must wait a reasonable amount of time to see whether the FIN segment from Program B is retransmitted, indicating that Program B never received the final ACK segment from Program A. In that case, Program A must be able to retransmit the final ACK segment. The Program A socket cannot be freed until this time period has elapsed. The time period is defined as twice the maximum segment life time, normally in the range of 1 to 4 minutes, depending on the TCP implementation.

If Program A is the client in a TCP connection, this TIMEWAIT state does not create any major problems. A client normally uses an ephemeral port number; if the client restarts before the TIMEWAIT period has elapsed, it is merely assigned another ephemeral port number. If Program A, on the other hand, is the server in a TCP connection, this TIMEWAIT state does create a problem. A server binds its socket to a predefined port number; if the server tries to restart and bind the same port number before the TIMEWAIT period has elapsed, it receives an EADDRINUSE error code on the bind() call. This situation could arise when a server crashes and you try to restart it before the TIMEWAIT period has elapsed. You must wait to restart your server.

If the server cannot wait for one to four minutes, you can use the setsockopt() call in the server to specify SO_REUSEADDR before it issues the bind() call. In that case, the server will be able to bind its socket to the same port number it was using before, even if the TIMEWAIT period has not elapsed. However, the TCP protocol layer still prevents it from establishing a connection to the same partner socket address. As clients normally initiate connections and clients use ephemeral port numbers, the likelihood of this is low.

Shutdown call in an iterative server program

If you want to close the stream in one direction only, use the shutdown socket call instead of the close() call. On the shutdown() call, you can specify the direction in which the stream is to be closed.

When a shutdown() call is issued for receive and there is unread data queued to the socket, the connection is aborted. If data arrives inbound on a connection that has been shut down for receive, the connection is aborted. When the connection is aborted, all outstanding socket calls on the socket will be posted with an ECONNABORTED error. The abort discards all unsent and unreceived data on the local

and remote end of the connection, and the connection is destroyed. The application should issue a `close()` on the socket.

See Table 3 on page 32 for a list of the effect on read and write calls when the stream is shut down in one or both directions.

Table 3. Effect of shutdown socket call				
Socket calls in local program	Local program		Remote program	
	Shutdown SEND	Shutdown RECEIVE	Shutdown RECEIVE	Shutdown SEND
Write calls	Error number EPIPE on first call		Error number EPIPE on second call*	
Read calls		Zero length return code		Zero length return code
* If you issue two write calls immediately, both might be successful, and an EPIPE error number might not be returned until a third write call is issued.				

Linger option in an iterative server program

By default, a `close` socket call returns control to your program immediately, even where there is unsent data on the socket. This data will be transmitted by the TCP protocol layer, but your program is not notified of any error. This is true of both blocking and nonblocking sockets.

You can request that no control be returned to your program before unsent data has been transmitted and acknowledged by the receiver. To do so, issue the `SO_LINGER` option on call `setsockopt`. Before you issue the actual `close()` call, pass the following option value fields:

ONOFF

This fullword is used to enable or disable the `SO_LINGER` option. Any nonzero value enables the option; a 0 value disables it.

LINGER

This is the linger time, in seconds; this is the maximum delay the `close` call observes. If data is successfully transmitted before this time expires, control is returned to your program. If this time interval expires before data has been successfully transmitted, control is returned to your program also. You cannot distinguish between the two return events.

Note: If you set a 0 linger time, the connection cannot close in an orderly manner, but stops, resulting in a RESET segment being sent to the connection partner. Also, if the aborting socket is in nonblocking mode, the `close` call is treated as though no linger option had been set.

Chapter 4. Designing a concurrent server program

This information describes concurrent server programs. All examples are shown using an address family of AF_INET (IPv4). All concepts also can be applied to an address family of AF_INET6 (IPv6). These programs include:

- [“Concurrent servers in native MVS environment” on page 33](#)
- [“MVS subtasking considerations in a concurrent server program” on page 34](#)
- [“Understanding the structure of a concurrent server program” on page 37](#)
- [“Selecting requests in a concurrent server program” on page 37](#)
- [“Client connection requests in a concurrent server program” on page 41](#)
- [“Transferring data between sockets in a concurrent server program” on page 45](#)
- [“Closing a concurrent server program” on page 45](#)

For the MVS address space examples presented in this topic, the more traditional MVS subtasking facilities are used; the main process and the child process operate as tasks within the same address space.

You can implement your concurrent server in an IMS, a CICS, or a traditional MVS address space environment, but unlike the implementation of an iterative server, the implementation of a concurrent server is unique to its environment. In this topic, the implementation of a concurrent server in an MVS address space is used as an example.

A server handling more than one client simultaneously acts like a dispatcher. The server receives client requests and then creates and dispatches tasks to handle each client.

In the UNIX operating system, a new process is dispatched using the `fork()` system call after the server has established the connection; this new process automatically inherits the socket attached to the client. In MVS, an independent task is started using the supervisor call instruction ATTACH. A server can complete the call after each connection is established (similar to the UNIX operating system), or it can repeatedly request an ATTACH when it begins execution, and pass clients to tasks that exist. In either case, the server must manually give the new socket to the subtask. Because each task has its own socket table, it is not sufficient to pass only the socket number to the subtask; for example, socket Number 4 for Task A is not the same as socket Number 4 for Task B. You must specify the task as well as the socket number.

Concurrent servers in native MVS environment

The concurrent server is complicated to implement. Logic must be split into a main program and a child program. In addition, you have to manage all processes within your application.

In the MVS environment, you implement such logic by means of the UNIX `fork()` call. Because this call is not available in a traditional MVS environment, you must improvise.

In the UNIX environment, the `fork` function is implemented using APPC/MVS to schedule and initiate a child process in an MVS address space other than the address space of the original process.

Note: For simplicity, the scope of our applications is limited to the AF_INET addressing family and stream sockets.

If you want to implement a high-performance server application that creates or accesses MVS resource of various kinds (especially MVS data sets), you will probably implement your server as a concurrent server in an MVS address space. This address space can be TSO, batch, or started task.

To implement concurrence in an MVS address space, use MVS multitasking facilities. This limits available programming interfaces to the sockets extended assembler macro programming interface or to C sockets.

For the sockets extended assembler macro interface, use standard MVS subtasking facilities: ATTACH and DETACH assembler macros.

For C sockets, use the subtasking facilities that are part of the IBM implementation of C in an MVS environment.

The following topics show sockets extended assembler macro examples to illustrate the implementation of a concurrent server in an MVS address space environment.

MVS subtasking considerations in a concurrent server program

Using multiple tasks in a single address space brings unique challenges that apply equally to assembler programming and to high-level languages that support subtasking.

For example, tasks might be concurrently dispatched on different processors, for example, running your application on an *n*-way system. Two or more tasks might execute in parallel, one perhaps passing the other.

Access to shared storage areas in a concurrent server program

If two tasks access the same storage area, you need full control over the use of the storage area unless the storage is read-only. If the storage area is used to pass parameters between the tasks, you must serialize access to the shared resource (the storage area).

In an MVS environment, you can use MVS latching services or traditional enqueue and dequeue system calls to access the shared resource. For MVS latching services, use the ISGLOBT and ISGLREL callable services. In assembler, use the ENQ and DEQ macros for enqueue and dequeue functions.

Figure 18 on page 34 illustrates access to a shared storage area.

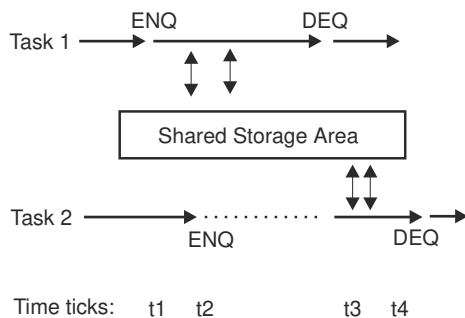


Figure 18. Serialized access to a shared storage area

The following steps describe this process.

1. At time t1, Task 1 issues a serialize request by means of an enqueue call. On the enqueue() call it passes two character fields to uniquely identify the resource in question. The literal value of these two fields does not matter; the other tasks must use these same values when they access this storage area. As no other task has issued an enqueue for the resource in question, Task 1 gets access to it and continues to modify the storage area.
2. At time t2, Task 2 needs to access the same storage area, and issues an enqueue() call using the same resource names used by Task 1. Because Task 1 has enqueued, Task 2 is placed in a wait and stays there until Task 1 releases the resource.
3. At time t3, Task 1 releases the resource with a dequeue system() call, and Task 2 is immediately taken out of its wait and begins to modify the shared storage area.
4. At time t4, Task 2 has finished updating the shared storage area and releases the resource with a dequeue system() call. (In this example, we assumed we need serialized access only when the tasks need to update information in the shared storage area.)

There are situations in which this assumption does not suffice. If you use a storage area to pass parameters to some kind of service task inside your address space, you must ensure that the service task has read the information and acted on it before another task in your address space tries to pass information to the service task using the same storage area, like running a log or trace. This is illustrated in Figure 19 on page 35.

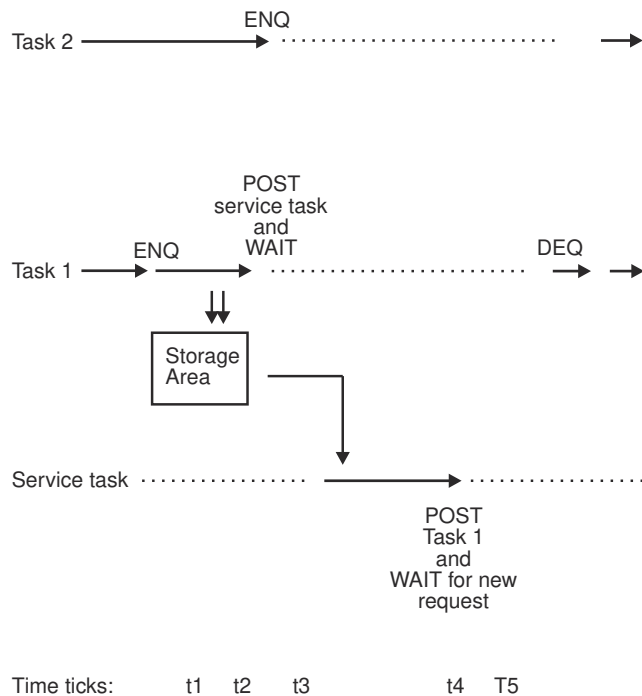


Figure 19. Synchronized use of a common service task

Follow these steps to synchronize a common service task:

1. At time t1, Task 1 gains access to the common storage area to implicitly use the service task in question.
2. At time t2, Task 2 also needs to use the service task services, but it is placed into a wait, because Task 1 already has the resource.
3. At time t3, Task 1 has finished placing values into the common storage area, and signals the service task to start processing it. This is done with a POST system call. Immediately following this call, Task 1 enters a wait, where it stays until the service task has completed its processing. The service task starts, processes the data in the common storage area, and prints.
4. At time t4, the service task has finished its work and signals to Task 1 that Task 1 can continue, while the service task enters a new wait and waits for a new work request.
5. At time t5, Task 1 releases the lock it obtained at time t1, and Task 2 is immediately taken out of its wait and starts filling its values into the common storage area before posting the same service task to process a new request.

This technique is relatively simple. It can be made more complicated, and more efficient, by using internal request queues so the requesting task does not need to wait for the service task to complete the active request.

When you use the enqueue system call, you have the option to test whether a resource is available. In some situations, you might choose this to avoid the wait at a particular point in your processing, so you can divert to some other actions when the resource is not available.

Data set access in MVS

When you access MVS data sets in a multitasking environment, observe these general rules:

- A given DD-name can be used by only one open data control block (DCB) at a time. If you need to have more DCBs open for the same data set, you must use different DD names. This strategy works best for read access only.
- Only the task that opens a DCB can issue read and write requests using that DCB. You cannot let your main task open a DCB, and then have your subtasks issue read or write requests to that DCB. You can deal with this by using the technique described, but include a special services task that opens a DCB to a particular data set. Other tasks then issue requests to this service task for access to the data set. Such a service task is generally called a data services task (DST). One very common implementation of a DST is the example used above: print log and trace information to a sysout file.
- Authorization checking for access to a data set is done when the data set is opened, not for every read or write request. If you develop a multitasking server where you establish task level security environments for each transaction entering your server, you must plan to authorize access to the information in a data set owned by a DST. You can, of course, open and close the data set for each transaction, but that might degrade performance.

Task and workload management in a concurrent server program

When a program is started by MVS, it runs as the main task of the address space in which it was started. In the examples in this topic, the main task is used as the main process of our concurrent server implementation. The child processes are then started as subtasks to the main.

Generally, there are two ways to manage your processes:

- Each time a connection request arrives, a new subtask is started. The subtask makes one connection and then terminates.
- During initialization, the main task starts a number of subtasks. Each subtask initializes and enters wait-for-work status. When a connection request arrives, the main process selects the first subtask waiting for work and schedules the connection to that subtask. The subtask processes the connection and, when complete, reenters wait-for-work status.

The second process is most efficient because it limits the overhead of creating new tasks to one time during server startup. But, it is also more complicated to implement than the other process because:

- You must decide on the number of server subtasks to be started during initialization. If more connection requests arrive than you have server subtasks available, you must include code to deal with that situation. (Reject the connection or dynamically change the number of subtasks in your concurrent server address space. This is called workload management.)
- The subtasks must be reusable and include logic to enter wait-for-work status; they must be able to process connection requests serially.
- The main process must be able to manage situations in which a server subtask abends or terminates.
- To achieve a graceful shutdown, you must implement a technique to terminate subtasks in an orderly manner. A simple technique is to post the subtask from the main process with a return code. For example, use a return code of 0 for work and some other value for termination.

In the concurrent MVS server example ([Figure 20 on page 37](#)), the technique using a pool of subtasks that waited for work was presented. We did not implement a dynamic increase of subtasks, but sent a negative reply back to the requester when no server subtasks were available.

Security considerations in a concurrent server program

When you start your server address space in MVS, a security environment is established for that address space. This environment is based on the user ID of your batch job or TSO user, or based on the started task user ID associated with the started task procedure named in the RACF started task table (ICHRIN03).

Unless you specify otherwise, all tasks in your address space execute under the security environment of the address space. MVS resources access authorization is based on the MVS address space security environment.

If this setup does not meet your needs, MVS allows you to build and delete task-level security environments using the RACROUTE REQUEST=VERIFY function in MVS. The task must run in an authorized state.

Reentrant code in a concurrent server program

Reentrant code is not required but is efficient. Non-reentrant code is loaded into virtual storage as many times as subtasks requiring it are started. Reentrant code is loaded once.

High-level languages usually make reentrancy a compile option. In assembler language, it might be more complicated; however, good use of macros for program initiation and termination can simplify the process.

Understanding the structure of a concurrent server program

Figure 20 on page 37 shows the basic logic in a multitasking concurrent server.

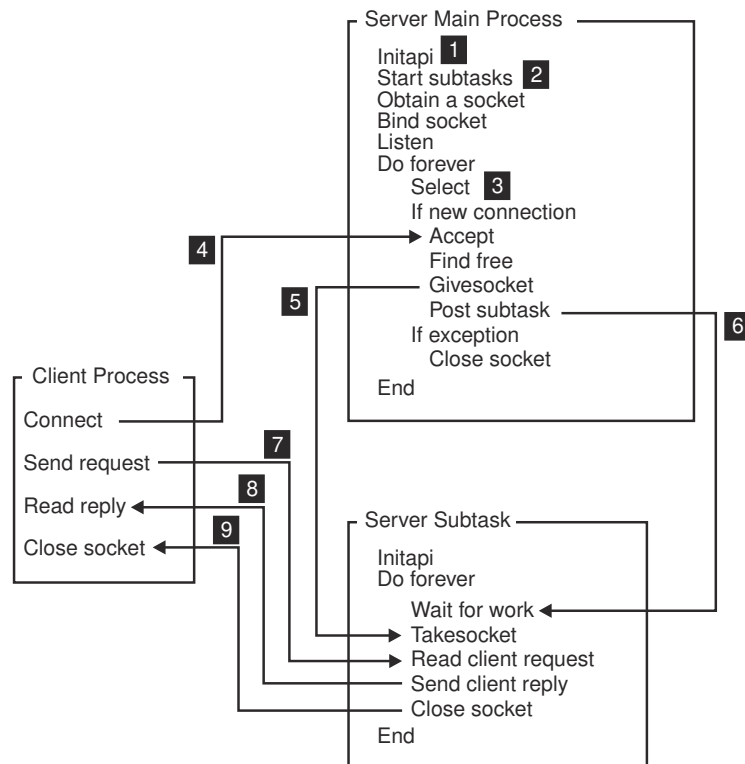


Figure 20. Concurrent server in an MVS address space

Selecting requests in a concurrent server program

At this point in the process, the server is ready to handle requests on this port from any client on a network from which the server is accepting connections. Until this point however, it had been assumed that the server was handling one socket only. Now, an application is not limited to one socket. Typically, a server listens for clients on a particular socket, but it allocates a new socket for each client it handles. For maximum performance, a server should operate only on those sockets ready for communication. The `select()` call allows an application to test for activity on a group of sockets.

To test any number of sockets with one call to `select()`, place 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 member of the set is represented by 0 or 1. If the members bit is 0, the member is not in the set; if the members bit is 1, the member is in the set. For example, if socket 3 is a member of a bit set, then bit 3 is set; otherwise, bit 3 is cleared.

In C language, the following functions are used to manipulate the bit sets:

FD_SET

Sets the bit corresponding to a socket.

FD_ISSET

Tests whether the bit corresponding to a socket is set or cleared.

FD_ZERO

Clears the entire bit set.

If a socket is active, it is ready for read or write data. If the socket is not active, an exception condition might have occurred. Therefore, the server specifies three bit sets of sockets in its call to the select() call as follows:

- One bit set for sockets on which to receive data
- One bit set for sockets on which to write data
- Any sockets with exception conditions

The select() call tests each socket in each bit set for activity and returns only those sockets that are active.

A server that processes many clients at once can be written to process only those clients that are ready for activity.

When all initialization is complete, and the server main process is ready to enter normal work, it builds a bit mask for a select() call. The select() call is used to test pending activity on a list of socket descriptors owned by this process. Before issuing the select() call, construct three bit strings representing the sockets you want to test, as follows:

- Pending read activity
- Pending write activity
- Pending exceptional activity

The length of a bit string must be expressed as a number of fullwords. If the highest socket descriptor you want to test is socket descriptor number 3, you must pass a 4-byte bit string, because this is the minimum length. If the highest number is 32, you must pass 8 bytes (2 fullwords).

The number of fullwords in each select mask can be calculated as follows:

```
INT(highest socket descriptor / 32) + 1
```

Table 4 on page 38 shows the first fullword passed using a bit string.

Table 4. First fullword passed in a bit string select()								
Socket descriptor numbers represented by byte	Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
Byte 0	31	30	29	28	27	26	25	24
Byte 1	23	22	21	20	19	18	17	16
Byte 2	15	14	13	12	11	10	9	8
Byte 3	7	6	5	4	3	2	1	0

Using standard assembler numbering notation, the leftmost bit or byte is relative to 0.

If you want to test socket descriptor number 5 for pending read activity, you raise bit 2 in byte 3 of the first fullword (X'00000020'). To test both socket descriptors 4 and 5, raise both bit 2 and bit 3 in byte 3 of the first fullword (X'00000030'). To test socket descriptor Number 32, pass 2 fullwords, where the numbering scheme for the second fullword resembles that of the first. Socket descriptor Number 32 is bit

7 in byte 3 of the second fullword. To test socket descriptors Number 5 and Number 32, pass 2 fullwords with the following content: X'0000002000000001'. The bits in the second fullword represent the socket descriptor numbers shown in Table 5 on page 39.

Table 5. Second fullword passed in a bit string using select()

Socket descriptor numbers represented by byte	Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
Byte 4	63	62	61	60	59	58	57	56
Byte 5	55	54	53	52	51	50	49	48
Byte 6	47	46	45	44	43	42	41	40
Byte 7	39	38	37	36	35	34	33	32

Subsequent mask words continue this pattern; word 3 for sockets 64-95, word 4 for sockets 96-127, and so on.

To set and test these bits in another way, use the following assembler macro, found in file SEZACMAC:

```

*****
.*
.* Part Name:          TPIMASK
.*
.* SMP/E Distribution Name: EZABCTPI
.*
.* Component Name:     SOK
.*
.* Copyright:          Licensed Materials - Property of IBM
.*                      This product contains "Restricted Materials of IBM"
.*                      5645-001 5655-HAL (C) Copyright IBM Corp. 1996.
.*                      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.
.*
.* Status:             TCP/IP for MVS
.*
.* Function:           Macro used to set or test bits in the
.*                      read, write and exception masks used
.*                      in the SELECT/SELECTEX macro or calls.
.*
.* Part Type:          MACRO - assembler
.*
.* Usage:
.*                      TPIMASK SET,MASK=READMASK,SD=SOCKDESC
.*                      or TEST, or WRITEMASK,
.*                      or EXCEPTMASK,
.*
.*                      SET - Set the SD bit on in MASK
.*                      TEST - Test SD bit in MASK for on/off
.*                      Follow the macro invocation with:
.*                      BE (Branch Equal) - Bit was on
.*                      BNE (Branch Not Equal) - Bit was off
.*
.* Change Activity:
.* CFD List:
.*
.* $xn= workitem  release  date  pgmr:  description
.*
.* End CFD List:
.*
*****
MACRO
TPIMASK &TYPE,          SET or TEST bit setting
                        &MASK=,      Read, Write or Except array
                        &SD=         Socket descriptor TOR PARAMETER
SR 14,14               Clear Reg14
AIF ('&SD'(1,1) EQ '().SDREG
X
X

```

	LH	15,&SD	Get Socket Descriptor
	AGO	.SDOK	
.SDREG	ANOP		
	LR	15,&SD	Get Socket Descriptor
.SDOK	ANOP		
	D	14,=A(32)	Divide by 32, R15 = word bit is in
	SLL	15,2	Multiply word by word length: 4
	AIF	('&MASK'(1,1) EQ '().MASKREG	
	LA	1,&MASK	Mask starts here
	AGO	.MASKOK	
.MASKREG	ANOP		
	LR	1,&MASK	Mask starts here
.MASKOK	ANOP		
	AR	15,1	Increment to word bit is in
	LA	1,1	Set rightmost bit on
	SLL	1,0(14)	Shift left remainder from division
	O	1,0(15)	Or with word from mask
	AIF	('&TYPE' EQ 'SET').DOSET	
	C	1,0(15)	If equal, bit was set on
	MEXIT		
.DOSET	ANOP		
	ST	1,0(15)	Update new mask after SET
	MEND		

Figure 21. To set/test bits for SELECT calls

If you develop your program using another programming language, you might be able to benefit from the EZACIC06 routine, which is provided as part of TCP/IP Services. This routine translates between a character string mask (1 byte per flag) and a bit string mask (1 bit per flag). If you use the select() call in COBOL, EZACIC06 can be very useful.

Build the three bit strings for the socket descriptors you want to test, and the select() call passes back three corresponding bit strings with bits raised for those of the tested socket descriptors with activity pending. Test the socket descriptors using the following sample:

* Test for socket descriptor activity with the SELECT call			*

	EZASMI	TYPE=SELECT,	*Select call C
		MAXSOC=TPIMMAXD,	*Max. this many descr. to test C
		TIMEOUT=SELTIMEO,	*One hour timeout value C
		RSNDMSK=RSNDMASK,	*Read mask C
		RRETMASK=RRETMASK,	*Returned read mask C
		WSNDMSK=WSNDMASK,	*Write mask C
		WRETMASK=WRETMASK,	*Returned write mask C
		ESNDMSK=ESNDMASK,	*Exception mask C
		ERETMSK=ERETMASK,	*Returned exception mask C
		ECB=ECBSELE,	*Post this ECB when activity occurs C
		ERRNO=ERRNO,	*- ECB points to an ECB plus 100 C
		RETCODE=RETCODE,	*- bytes of workarea for socket C
		ERROR=EZAERROR	*- interface to use.
	ICM	R2,15,RETCODE	*If Retcode < zero it is
	BM	EZAERROR	*- an error
*			
SELMASKS	DS	0F	
RSNDMASK	DC	XL8'00000000'	*Read mask
RRETMASK	DC	XL8'00000000'	*Returned read mask
WSNDMASK	DC	XL8'00000000'	*Write mask
WRETMASK	DC	XL8'00000000'	*Returned write mask
ESNDMASK	DC	XL8'00000000'	*Exception mask
ERETMASK	DC	XL8'00000000'	*Returned exception mask
*			
NOSELCD	DC	A(0)	*Keep track of selected sd's
SELTIMEO	DC	A(3600,0)	*One hour timeout
ECBSELE	DC	A(0)	*Select ECB
	DC	100X'00'	*Required by EZASMI
*			
TPIMMAXD	DC	AL4(50)	*Maximum descriptor number
*			
ERRNO	DC	A(0)	*Error number from EZASMI
RETCODE	DC	A(0)	*Returncode from EZASMI

In the above select() call, the asynchronous facilities of the socket assembler macro interface is used. By placing an ECB parameter on the EZASMI macro call, the select() call does not block the process;

we receive control immediately, even if none of the specified socket descriptors had activity. Use this technique to enter a wait, which waits for a series of events of which the completion of a `select()` call is just one. In the sample application, the main process was placed into a wait from which it would return when any of the following events occurred:

- Socket descriptor activity occurred, and the `select()` call was posted.
- One of your subtasks terminated unexpectedly.
- The MVS operator issued a MODIFY command to stop the server.

The number of socket descriptors with pending activity is returned in the `RETCODE` field. You must process all selected socket descriptors before you issue a new `select()` call. A selected socket descriptor is selected only once.

When a connection request is pending on the socket for which the main process issued the `listen()` call, it is reported as a pending read.

When the main process has given a socket, and the subtask has taken the socket, the main process socket descriptor is selected with an exception condition. The main process is expected to close the socket descriptor when this happens.

Applications can handle multiple sockets. In such situations, use the `select()` call to determine the sockets that have data to be read, those that are ready for data to be written, and the sockets that have pending exceptional conditions. An example of how the `select()` call is used is shown in [Figure 22 on page 41](#).

```
fd_set readsocks;
fd_set writesocks;
fd_set exceptsocks;
struct timeval timeout;
int number_of_sockets;
int number_found;
:
/* set bits in read write except bit masks.
 * To set mask for a descriptor's use
 * FD_SET(s, &readsocks)
 * FD_SET(s, &writesocks)
 * FD_SET(s, &exceptsocks)
 *
 * set number of sockets to be checked (plus 1)
 * number_of_sockets = x;
 */
:
number_found = select(number_of_sockets,
                      &readsocks, &writesocks, &exceptsocks, &timeout)
```

Figure 22. An application using the `select()` call

In this example, the application uses bit sets to indicate that the sockets are being tested for certain conditions, and also indicates a timeout. If the timeout parameter is `NULL`, the call does not wait for any socket to become ready. If the timeout parameter is nonzero, the `select()` call waits for the amount of time required for at least one socket to become ready under the indicated condition. This process is useful for applications servicing multiple connections that cannot afford to block, thus waiting for data on one connection. For a description, see [“select\(\)” on page 140](#).

Client connection requests in a concurrent server program

As shown in [Figure 20 on page 37](#), the listener socket is selected with a pending read. Then, a new connection request arrives, and the following `socket()` call must accept.

[Figure 23 on page 42](#) illustrates this type of connection request.

```

*-----*
* ACCEPT the connection from a client                                     *
*-----*
EZASMI  TYPE=ACCEPT,      *Accept new connection                C
        S=TPIMSN0,        *On listener socket descriptor        C
        NAME=SOCSTRUC,    *Returned client socket structure      C
        ERRNO=ERRNO,      C
        RETCODE=RETCODE,C
        ERROR=EZAERROR
        ICM              R15,15,RETCODE *OK ?
        BM               EZAERROR      *- No, error indicated
        STH              R15,NEWSOC     *Returned new socket descriptor
*
SOCSTRUC DS 0F                *ACCEPT Socket address structure
SSTRFAM  DC AL2(2)            *TCP/IP Addressing family
SSTRPORT DC AL2(0)            *Port number
SSTRADDR DC AL4(0)            *IP Address
SSTRRESV DC 8X'00'           *Reserved
*
TPIMSN0  DC AL2(0)            *Listen socket descriptor
*
NEWSOC   DC AL2(0)            *Returned socket descriptor
*
ERRNO    DC A(0)              *Error number from EZASMI
RETCODE  DC A(0)              *Returncode from EZASMI

```

Figure 23. Accepting a client connection

The accept call returns a new socket descriptor representing the connection with the client. The original listen socket descriptor is available to a new select() call.

Passing sockets in a concurrent server program

This topic contains concepts and tasks information about passing sockets.

Common interface concepts

To help you better understand socket passing, the following topics explain common interface concepts.

• Blocking versus nonblocking

A socket is in blocking mode when an I/O() call waits for an event to complete. If blocking mode is set for a socket, the calling program is suspended until the expected event completes.

If nonblocking is set by calls FCNTL() or IOCTL(), the calling program continues even though the I/O() call might not have completed. If the I/O() call could not be completed, it returns with ERRNO 35 (EWOULDBLOCK). The calling program should use select() to test for completion of any socket call returning an ERRNO 35.

The default mode is blocking.

- If data is not available for the socket, and the socket is in blocking and synchronous modes, the read() call blocks the caller until data arrives.

• Concurrent servers versus iterative servers

An iterative server handles one client at a time. A concurrent server receives connection requests from multiple clients and creates subtasks to process those client requests.

When a subtask is created, the concurrent server gets a new socket, passes the new socket to the subtask, and disassociates itself from the connection. (The CICS listener program is an example of a concurrent server.)

- To pass a socket, the concurrent server first calls givesocket(). If the subtask address space name and subtask ID are specified in the givesocket() call, only a subtask having a matching address space and subtask ID can take the socket. If this field is set to blanks, any MVS address space requesting a socket can take this socket.
- The concurrent server starts the subtask and passes to it the socket descriptor and concurrent server ID obtained from earlier socket() and getclientid() calls.

- The subtask calls `takesocket()` using the concurrent server ID and socket descriptor.
- The concurrent server issues the `select()` call to test the socket for the `takesocket-completion` exception condition.
- When `takesocket()` has successfully completed, the concurrent server issues the `close()` call to free the socket.
- If the queue has no pending connection requests, `accept()` blocks the socket when blocking mode is set on. You can set the socket to nonblocking by calling `fcntl` or `ioctl`.
- Issuing a `select()` call before the `accept()` call ensures that a connection request is pending. Using the `select()` call in this way prevents the `accept()` call from blocking.
- TCP/IP does not screen clients, but you can control the connection requests accepted by closing a connection immediately after you determine the identity of the client.
- A given TCP/IP host can have multiple aliases and multiple host internet addresses.

A server handling more than one client simultaneously acts like a dispatcher at a messenger service. A messenger dispatcher gets telephone calls from people who want items delivered, and the dispatcher sends out messengers to do the work. In a similar manner, the server receives client requests, and then spawns tasks to handle each client.

Tasks can pass sockets with the `givesocket()` and `takesocket()` calls. The task passing the socket uses `givesocket()`, and the task receiving the socket uses `takesocket()`. The following topics describe these processes.

givesocket and takesocket

In the UNIX operating system, a new process is dispatched with the `fork()` system call after the server has established the connection; the new process automatically inherits the socket attached to the client. In MVS, an independent task is started using the `attach()` supervisor call instruction. A server can perform an `attach()` call for a subtask after each connection is established in a way similar to the UNIX operating system, or it can request an `attach()` several times when it begins execution and pass clients to tasks that exist. In either case, the server must manually give the new socket to the subtask. Because each task has its own socket table, it is not sufficient to pass only the socket number to the subtask. Socket Number 4 for Task A is not the same as socket Number 4 for Task B.

For C programs using TCP/IP Services, each task is given a unique 8-byte name. The task uses the `getclientid()` call to determine its unique name. The main server task passes the following arguments to the `givesocket()` call:

- The socket number it wants to give
- Its own name
- The name of the task to which it wants to give the socket

If the server does not know the name of the subtask to receive the socket, it blanks out the name of the subtask. The first subtask calling `takesocket()` using the server unique name receives the socket. However, the subtask must know the main task unique name, and the number of the socket it is to receive. This information can be passed in a common work area that you define.

When `takesocket()` acquires the socket, it assigns a new socket number for the subtask to use, but the new socket number represents the same line of communication as the parent socket. The transferred socket can be referred to as socket Number 4 by the parent task, and as socket Number 3 by the subtask. However, both sockets represent the same connection to the TCPIP address space.

After the socket has successfully been transferred, the TCPIP address space posts an exception condition on the parent socket. The parent uses the `select()` call to test for this condition. After the notification, the parent task must issue `close()` call on its socket to deallocate the socket.

[Appendix A, “Multitasking C socket sample program,” on page 733](#) contains examples of a server, a subtask, and a client. Three examples are written in C, and one example is written in System/370 assembler language.

The C sample programs are included as members of the file SEZAINST partitioned data set. The member names are:

- MTCSRVR
- MTCCSUB
- MTCCLNT

For information about the JCL needed to use the multitasking facility (MTF), see *IBM C/370 User's Guide*.

Giving a socket to a subtask

The socket represented by the new socket descriptor has to be passed to an available subtask. Which technique the main process uses to find an available subtask is not important. Assume that the main process has located an available subtask to which it gives the socket by way of a `givesocket()` call as shown in [Figure 24 on page 44](#):

```

*-----*
*Give socket to subtask                                     *
*-----*
MVC    CLNNAM,TPIMCNAM    *Our Client ID Address Space Name
MVC    CLNTASK,TPISTCBE   *Give to this subtask
EZASMI TYPE=GIVESOCKET,  *Givesocket                                C
      S=NEWSOC,          *Give this socket descriptor              C
      CLIENT=CLNSTRUC,    *- to a specific child process          C
      ERRNO=ERRNO,        C
      RETCODE=RETCODE,    C
      ERROR=EZAERROR      C
      ICM  R15,15,RETCODE *OK ?
      BM   EZAERROR      *- No, tell about it.
*
*      CLNSTRUC DS      0F      *GIVESOCKET: Client structure
CLNFAM  DC      A(2)          *TCP/IP Addressing family
CLNNAM  DC      CL8' '        *Address space name of target
CLNTASK  DC      CL8' '        *Task ID of child process subtask
CLNRESV  DC      XL20'00'      *Reserved
*
NEWSOC  DC      AL2(0)        *Socket descriptor from Accept
*
ERRNO   DC      A(0)          *Error number from EZASMI
RETCODE  DC      A(0)          *Returncode from EZASMI

```

Figure 24. Giving a socket to a subtask

If you are programming in C, you might not be able to determine the full client ID of the subtask. In that case, you can pass the task ID field as eight blanks on the `givesocket()` call, which means that any task within your own address space can take the socket, but only the task to which you pass the socket descriptor number will actually take it.

After you have issued the `givesocket()` call, you must include the given socket descriptor in the exception select mask on the next `select()` call.

Your main process is now ready to wake up the selected subtask by way of a post system call.

If no other sockets were selected on the previous `select()` call, your main process can build a new set of select masks, and issue a new `select()` call.

Taking sockets from the main process

As shown in [Figure 20 on page 37](#), the subtask is reactivated by the `post()` call issued from the main process, and immediately issues a `takesocket()` call to receive the socket passed from the main process. [Figure 25 on page 45](#) illustrates this process.


```

*-----*
* Take socket from main process                                     *
*-----*
EZASMI TYPE=TAKESOCKET,      *Takesocket
      CLIENT=TPIMCLNI,      *Main task client id structure
      SOCRECV=TPISSOD,      *Main task socket descriptor
      ERRNO=ERRNO,
      RET CODE=RETCODE,
      ERROR=EZAERROR
      ICM R15,15,RETCODE      *Did we do well ?
      BM EZAERROR            *- No, deal with it.
      STH R15,TPISNSOD        *Server subtask socket descr.no
*
TPIMCLNI DS      0C          *Main task client id
TPIMCDOM DC      A(0)        *Domain: AF-INET
TPIMCNAM DC      CL8' '      *Our address space name
TPIMCTSK DC      CL8' '      *Main task TCB address in EBCDIC
      DC      20X'00'        *Reserved (part of clientid)
*
TPISSOD DC      AL2(0)       *Parent socket descr. no.
TPISNSOD DC      AL2(0)       *Subtask socket descr. no.

```

Figure 25. Taking sockets from the main process

In order to take a socket, the subtask must know the client ID of the task that gave the socket, and the socket descriptor used by that task. These values must be passed to the subtask from the main process before a `takesocket()` call can be issued.

On the `takesocket()` call, you specify the full client ID of the process that gave the socket, and you specify the socket descriptor number used by the process that gave the socket.

A new socket descriptor number to be used by the subtask is returned in the `RETCODE` when the `takesocket()` call is successful. As soon as your subtask has taken the socket, the main process is posted in its pending select with a pending exception activity; this means that the main process must close its socket descriptor.

In Figure 25 on page 45, the client sends its request to the subtask, which processes it and sends back a reply.

Finally, the client process and the server subtask close their sockets, and the server subtask reenters wait-for-work status.

Transferring data between sockets in a concurrent server program

See Chapter 7, “Transferring data between sockets,” on page 51.

Closing a concurrent server program

See Chapter 3, “Designing an iterative server program,” on page 23.

Chapter 5. Designing a client program

This information explains how to design a client program. All examples are shown using an address family of AF_INET (IPv4). All concepts also can be applied to an address family of AF_INET6 (IPv6).

- [“Allocating a socket in a client program” on page 47](#)
- [“Connecting to a server in a client program” on page 47](#)
- [“Transferring data between sockets in a client program” on page 48](#)
- [“Closing a client program” on page 48](#)

Allocating a socket in a client program

From their own perspective, clients must first issue the `socket()` call to allocate a socket from which to communicate as follows:

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

For more information, see [“Allocating sockets in an iterative server program” on page 23](#).

Connecting to a server in a client program

To connect to a server, the client must know the server name. This topic describes how to determine a server name and connect to that server.

Note: Examples are written in C language and REXX.

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 `bind()` call. If the client does not know the server IP address, but it does know the server host name, the `gethostbyname()` call is called to translate the host name into its IP address. Any trailing blanks will be removed from the specified name before trying to resolve it to an IP address.

The client then calls `connect()` as shown in the following C language example of the `connect()` call:

```
connect(s, name, namelen);
```

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

If you need to determine a server name while writing in REXX and you know only the host name, you must resolve the host name into one or more IP addresses using the `gethostbyname()` call as shown in [Figure 26 on page 47](#):

```
/*-----*/
/* Find IP addresses of server host */
/*-----*/
servipaddr = DoSocket('Gethostbyname', tpiserver)
if sockrc <> 0 then do
  say 'Gethostbyname failed, rc='sockrc
  say sockval
  x=Doclean
  exit(sockrc)
end
```

Figure 26. Finding the IP address of a server host using `gethostbyname()`

The REXX `gethostbyname()` call returns a list of IP addresses if the host is multiply defined as a home host. You can parse the REXX string and place the IP addresses into a REXX stem variable using the following piece of REXX code:

```

/*-----*/
/* Parse returned IP address list */
/*-----*/
numips = words(servipaddr)
do i = 1 to numips
  sipaddr.i = word(servipaddr, i)
end
sipaddr.0 = numips

```

If you issue a connect call to an IP address currently not available, your connect call times out with an error number of 60 (ETIMEDOUT). The socket you used on such a failed connect call cannot be reused for another connect() call. You have to close the existing socket and get a new socket before you reissue the connect call using the next IP address in the list of IP addresses returned by the gethostbyname() call.

The connect call can be placed in a loop that terminates when a connect is successful, or the list of IP addresses is exhausted. The following sample illustrates this process.

```

/*-----*/
/* */
/* Get a socket and try to connect to the server */
/* */
/* If connect fails (ETIMEDOUT), we must close the socket, */
/* get a new one and try to connect to the next IP address */
/* in the list, we received on the gethostbyname call. */
/* */
/*-----*/
i=1
connected = 0
do until (i > sipaddr.0 | connected)
  sockdescr = DoSocket('Socket')
  if sockrc <> 0 then do
    say 'Socket failed, rc='sockrc
    exit(sockrc)
  end
  name = 'AF_INET '||tpiport||' '||sipaddr.i
  sockval = DoSocket('Connect', sockdescr, name)
  if sockrc = 0 then do
    connected = 1
  end
  else do
    sockval = DoSocket('Close', sockdescr)
    if sockrc <> 0 then do
      say 'Close failed, rc='sockrc
      exit(sockrc)
    end
  end
  i = i + 1
end
if connected then do
  say 'Connect failed, rc='sockrc
  exit(sockrc)
end

```

Transferring data between sockets in a client program

See [Chapter 7, “Transferring data between sockets,”](#) on page 51.

Closing a client program

See [Chapter 3, “Designing an iterative server program,”](#) on page 23.

Chapter 6. Designing a program to use datagram sockets

This information explains how to design a program to use datagram sockets. All examples are shown using an address family of AF_INET (IPv4). All concepts also can be applied to an address family of AF_INET6 (IPv6). Topics include:

- [“Datagram socket characteristics” on page 49](#)
- [“Understanding datagram socket program structure” on page 49](#)
- [“Allocating a datagram socket” on page 50](#)
- [“Binding datagram sockets to port numbers” on page 50](#)
- [“Streamline data transfer using connect call” on page 50](#)
- [“Transferring data between datagram sockets” on page 50](#)

Datagram socket characteristics

The most significant characteristics of datagram sockets follow:

- Datagram sockets are connectionless.

There is no connection setup affected by the UDP protocol layer. No data is exchanged between sending and receiving UDP protocol layers until your application issues its first send call.

If your UDP server program has not been started or it resides on a host that cannot be reached from your client host, your client UDP application can wait forever to receive a reply to the datagram it sent to a UDP server. You have to implement timeout logic in your client UDP program to recognize this situation.

- The UDP protocol layer does not implement reliability functions.

The implicit significance of this fact is that a datagram sent from one UDP program to another might never arrive. Neither the sending program nor the target program ever learns from the UDP protocol layer that such a condition exists.

If your UDP application must be reliable, you must add reliability code to your UDP client and server programs. Such code must include the ability to detect missing datagrams, datagrams arriving out of sequence, duplicate datagrams, and corrupt datagrams.

You can use TCP protocols instead of UDP protocols if your application must be reliable.

- Unlike a TCP socket, where there is no one-to-one relationship between send() and recv() calls, UDP socket send corresponds exactly to a UDP socket recv() call.

Understanding datagram socket program structure

The datagram socket program terms client and server can be misleading. Two socket programs that have each bound a socket to a local address can send any number of datagrams to each other in any sequence. The program that sends the first data will act as a client. Any datagram sent to a destination address for which no program has bound a socket is lost. Care must be taken so that the program you intend to be the client does not begin sending datagrams until the server program has bound its socket to the destination address expected.

Typically, the structure for a datagram socket resembles the iterative server discussed in [Chapter 3, “Designing an iterative server program,” on page 23](#).

Allocating a datagram socket

See [“Allocating sockets in an iterative server program”](#) on page 23.

Binding datagram sockets to port numbers

The server program must bind its socket to a predefined server port number, so the clients know the port to which they should send their datagrams. In the socket address structure that the server passes on the `bind()` call, it can specify if it will accept datagrams from the available network interfaces, or whether only from a specific network interface. This is done by setting the IP address field of the socket address structure to either `INADDR_ANY`, or a specific IP address.

The client program needs to bind its socket to a local address if it wants the server program to be able to return a datagram to it. In contrast to the server, the client does not need to specify a specific port number on the `bind()` call; an ephemeral port number chosen by the UDP protocol layer is sufficient. This is called a dynamic bind.

Streamline data transfer using connect call

While you can use the `connect()` call on a datagram socket, it does not act for a datagram socket as it acts for a stream socket.

On a `connect()` call, you specify the remote socket address with which you want to exchange datagrams. This serves the following purposes:

- On succeeding calls to send datagrams, you can use the `send()` call without specifying a destination socket address; the datagram is sent to the socket address you specified on the `connect()` call.
- On succeeding calls to receive datagrams, only datagrams that originate from the socket address specified on the `connect()` call are passed to your program from the UDP protocol layer.

Note: A `connect()` call for a datagram socket does not establish a connection. No data is exchanged over the IP network as the result of the `connect()` call. The functions performed are local, and control is returned immediately to your application.

Transferring data between datagram sockets

See [Chapter 7, “Transferring data between sockets,”](#) on page 51.

Chapter 7. Transferring data between sockets

This topic contains information about transferring data between sockets. All examples are shown using an address family of AF_INET (IPv4). All concepts also can be applied to an address family of AF_INET6 (IPv6). The following topics are included:

- [“Transferring data between sockets: Streams and messages” on page 51](#)
- Data representation

AF_INET6 (IPv6) sockets can communicate with AF_INET (IPv4) sockets using mapped addresses. See [z/OS Communications Server: IPv6 Network and Appl Design Guide](#) for details.

- Using send() and recv() calls
- Using sendto() and recvfrom() calls

Transferring data over a datagram socket is similar to working with MVS records. You send and receive data records. One send() call results in exactly one recv() call.

If your sending program sends a datagram of 8192 bytes, and your receiving program issues a recv() call in which it specifies a buffer size of, for example, 4096 bytes, it will receive the 4096 bytes it requested. The remaining 4096 bytes in the datagram are discarded by the UDP protocol layer without further notification to either sender or receiver.

z/OS Communications Server includes a performance enhancement that when both the source and destination of a packet are known to and managed by a single TCP/IP stack, the IP layer can be bypassed. This provides an overall pathlength savings when processing such packets, and the decrease in pathlength through the stack results in an overall throughput improvement for applications that reside on the same MVS systems and communicate with each other through the same TCP/IP stack. Socket application programmers can take advantage of this performance enhancement by using a non-loopback home address when sending data between applications that reside on the same MVS system and communicate with each other through the same TCP/IP stack. See [z/OS Communications Server: New Function Summary](#) for additional information.

Transferring data between sockets: Streams and messages

This topic describes how to design an application protocol so that the partner program can divide the receive stream into individual messages.

Some socket applications are simple, and the receiver can continue to receive data until the sender closes the socket, for example, a simple file transfer application. Most applications are not that simple and usually require that the stream can be divided into a number of distinct messages.

A message exchanged between two socket programs must imbed information so that the receiver can decide how many bytes to expect from the sender and (optionally) what to do with the received message.

A few common techniques are used to imbed information about the length of a message into the stream, as follows:

- The message type identifier technique

If your messages are fixed length, you can implement a message ID per message type worked with. Each message type has a predefined length that is known by your client and server programs. If you place the message ID at the start of each message, the receiving program can determine how long the message is if it knows the content of the first few bytes in the message. This is illustrated in [Figure 27 on page 52](#):

```

*-----*
* Layout of a message between TPI client and TPI server      *
*-----*
01 tpi-message.
05 tpi-message-id          pic x.
88 tpi-request-add         value '1'.
88 tpi-request-update      value '2'.
88 tpi-request-update      value '2'.
88 tpi-request-query       value '3'.
88 tpi-request-query       value '3'.
88 tpi-request-delete      value '4'.
88 tpi-query-reply         value 'A'.
88 tpi-response            value 'B'.
05 tpi-constant            pic x(4).
88 tpi-identifier          value 'TPI '.

```

Figure 27. Layout of a message between a TPI client and a TPI server

Each message ID is associated with a fixed length known to your application.

- The record descriptor word (RDW) technique

If your messages are variable length, you can implement a length field in the beginning of each message. Normally, you would implement the length in a halfword binary length with the value encoded in network byte order, but you can implement it as a text field, as shown in [Figure 28 on page 52](#).

```

*-----*
* Transaction Request Message segment                        *
*-----*
01 TRM-message.
05 TRM-message-length      pic 9(4) Binary Value 20.
05 filler                  pic x(2) Value low-value.
05 TRM-identifier          pic x(8) Value '*TRNREQ*'.
05 TRM-trancode            pic x(8) Value '?????'.

```

Figure 28. Transaction request message segment

- The end-of-message marker technique

A third technique most often seen in C programs is to send a null-terminated string. A null-terminated string is a string of bytes terminated by a byte of binary 0. The receiving program reads whatever data is on the stream and then loops through the received buffer separating each record at the point where a null-byte is found. When the received records have been processed, the program issues a new read for the next block of data on the stream.

If your messages contain only character data, you can designate any non-display byte value as your end-of-message marker. Although this technique is most often seen in C programs, it can be used with any programming language.

- The TCP/IP buffer flushing technique

This technique is based on the observed behavior of the TCP protocol, where a `send()` call followed by a `recv()` call forces the sending TCP protocol layer to flush its buffers and forward whatever data might exist on the stream to the receiving TCP protocol layer. You can use this method to implement a half-duplex, flip-flop application protocol, where your two partner programs acknowledge the receipt of each message with, for example, a 1-byte application acknowledgment message.

[Figure 29 on page 53](#) shows the TCP buffer flush technique.

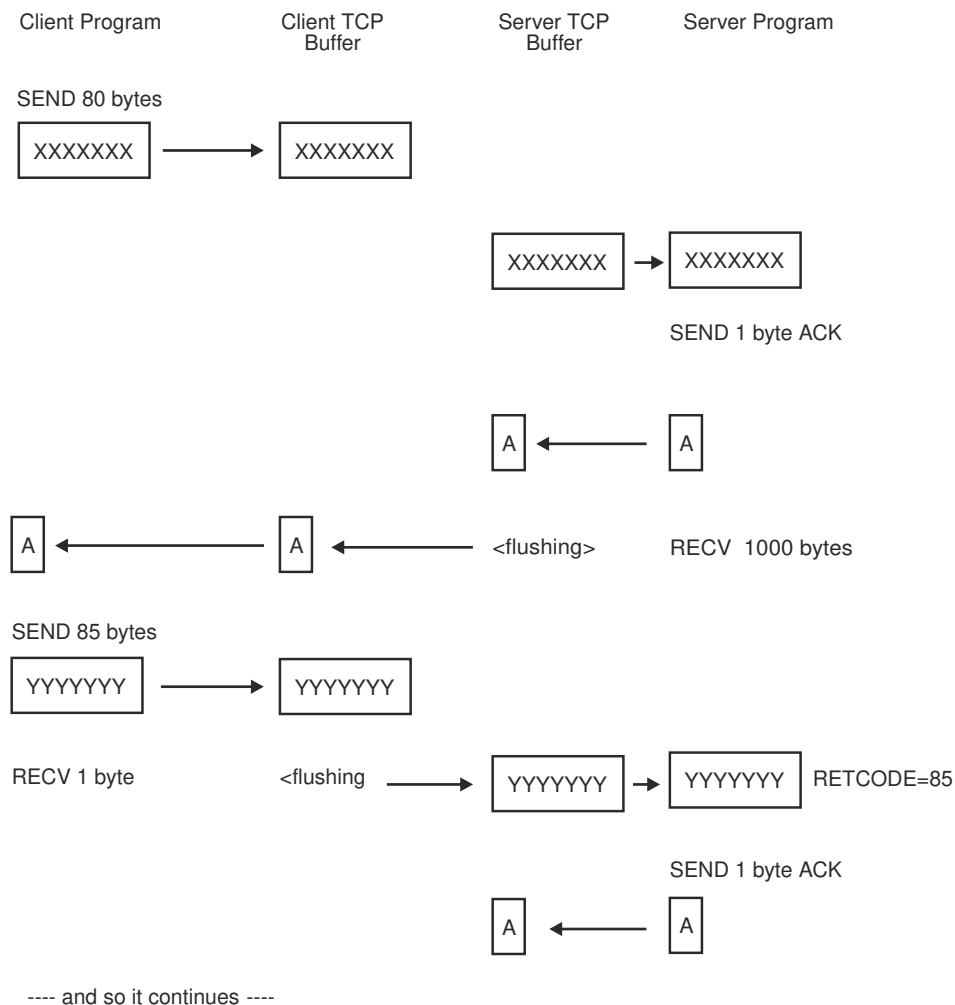


Figure 29. The TCP buffer flush technique

In Figure 29 on page 53, the client sends an 80-byte message. The server has issued a `recv()` call for 1000 bytes, but receives only the 80 bytes (`RETCODE=80`). This presents a problem because there is no guarantee the server will receive the full 80-byte message on its receive call. It might only receive 30 bytes, but with this technique it has no way of knowing that it is missing another 50 bytes. The smaller the messages are, the less likely the server will receive only a part of the full message.

Note: This technique is widely used, but you should use it only in controlled environments, or in programs where you use non-blocking socket calls to implement your own timeout logic.

The message type identifier and the record descriptor word techniques require that the receiving program be able to learn the content of the first bytes in the message before it reads the entire message.

If this is a problem for your application, use the peek flag on a `recv socket()` call.

A `recv()` call with the peek flag on does not remove the data from the TCP buffers, but copies the number of bytes you requested into the application buffer you specified on the `recv()` call.

For example, if your message length field or message ID field is located within the first 5 bytes of each message, issue the following `recv()` call:

```
*-----*
* Peek buffer and length fields for RECV call      *
*-----*
01  socket-recv                                pic x(16) value 'RECV'.
01  recv-flag-peek                             pic 9(8) binary value 2.
01  recv-peek-len                              pic 9(8) binary value 5.
01  recv-peek-buffer.
05  message-id                                pic x value space.
88  tpi-query-reply                           value 'A'.
```

```

      88 tpi-response          value 'B'.
    05 message-constant      pic x(4).
      88 tpi-identifier      value 'TPI'.
    01 socket-descriptor     pic 9(4) binary value 0.
    01 errno                pic 9(8) binary value 0.
    01 retcode               pic s9(8) binary value 0.
*-----*
* Peek at first 5 bytes of client data                                *
*-----*
      call 'EZASOCKET' using socket-recv
      socket-descriptor
      rcv-flag-peek
      rcv-peek-len
      rcv-peek-buffer
      errno
      retcode.
    if retcode < 0 then
      - process error -
    if retcode = 0 then
      - process client closed socket -
    if not TPI-identifier then
      - translate rcv-peek-buffer from ASCII to EBCDIC -

```

The `recv()` call blocks until some bytes have been received or the sender closes its socket. The above example is not complete since you cannot be sure that you actually received the 5 bytes requested. Your call might come back to you with only 1 byte received. In order to manage the situation, you need to repeat your `recv()` call until all 5 bytes have been received and recognized as such.

If the other half of the connection closes the socket, the `recv()` call returns 0 in the *retcode* field.

The data is copied into your application program buffer only, but it is still available to a `recv()` call, in which you can specify the full length of the message you now know to be available.

Transferring data between sockets: Data representation

If you use the socket API, your application must handle the issues related to different data representations occurring on different hardware platforms. For character-based data, some hosts use ASCII, while other hosts use EBCDIC. Translation between the two representations must be handled by your application.

For integers, some hardware platforms use big endian byte order (S/370/390, Motorola style), while others use little endian byte order (Intel style). An example of the difference between big and little endian byte orders is shown in [Figure 30 on page 54](#).

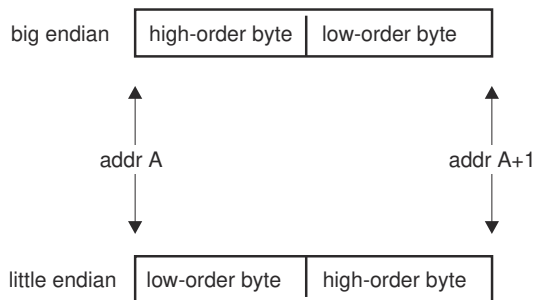


Figure 30. Big or little endian byte order for a 2-byte integer

IBM S/370 and IBM S/390®-based computers all use big endian byte order, while the IBM PS/2 uses the little endian byte order. TCP/IP has defined a network byte order standard to be used for all 16-bit and 32-bit integers that appear in protocol headers. This network byte order is based on the big endian byte order. This is the reason you find the following information in the C-socket interface:

htons

Translates a short integer (two bytes) from host byte order to network byte order

ntohs

Translates a short integer from network byte order to host byte order

htonl

Translates a long integer (four bytes) from host byte order to network byte order

ntohl

Translates a long integer from network byte order to host byte order

The socket-based application should manage the application data portion of a message. If you develop a server that serves clients on different hardware platforms, define your own standard and implement it as part of your application protocol.

In some cases, it is easier to base your messages on text data. If you, as part of your message design, define a fixed text string in the beginning of each message, your application can test the contents of this string and decide whether the data is in EBCDIC or ASCII. If the data is in ASCII, you can translate the full message from ASCII to EBCDIC on input, and translate from EBCDIC to ASCII on output from MVS. An example of this design is the transaction request message (TRM) format used by the IMS Listener program. Bytes 4 to 11 have a fixed value of *TRNREQ*, which is used both to distinguish this message from other messages and to find out whether the client is transmitting data in ASCII or EBCDIC.

If you mix text data and binary data in your messages, be sure to apply translation only between ASCII and EBCDIC to the text fields in your message.

If you use binary integer fields in your messages, you can use the network byte order standard that TCP/IP uses for all integers in protocol headers. If you design your messages according to the network byte order standard, your MVS programs do not need to translate or rearrange the bytes in binary integer fields. Your programs executing on little endian hosts must use the integer conversion routines to convert integers between local format and the format used in the messages they exchange with your MVS programs.

Text data and 2- and 4-byte binary integers are easy to handle in a heterogeneous computer environment. In more complex data types like floating point numbers or packed decimal, it becomes much more complicated because there is no generally accepted standard and there is no easy support for transformation between the formats. If you include these data types in your messages, be sure that the partner program knows how to interpret them. If the two computer systems use the same architecture, this is valid. If you exchange messages by way of socket programs between two MVS systems, you do not need to be concerned about conversion.

Using send() and recv() calls

This topic provides information about sending and receiving calls.

The send() and recv() call conversation

Client and server communicate using send() and recv() as shown below:

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

The send() and recv() calls specify:

- The socket *s* on which to communicate
- The address in storage 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*)
- A flag that tells how the data is to be sent

Flag 0 tells TCP/IP to transfer the data normally. The server uses the socket that is returned from the accept() call.

These functions return the amount of data that was sent or received. Because stream sockets send and receive information in streams of data, it can take more than one send() or recv() to transfer all of the data. It is up to the client and the server to agree on some mechanism to signal that all of the data has been transferred.

When the conversation is over, both the client and the server call `close()` to end the connection. `close()` also 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, it can no longer accept new connections, but it can still converse with the clients to which it is connected. The `close()` call is represented as follows:

```
close(s);
```

If you are writing a client application, you might want to verify the processes the server will use. Both client applications and the servers with which they communicate must be aware of the sequence of events each will follow.

Using socket calls in a network application

You can use the following example to write a socket network application. The example is written using C socket syntax conventions, but the principles described apply to all of the following APIs:

- TCP/IP C socket API
- X/Open Transport Interface
- Macro API for IPv4 or IPv6 that is written in z/OS assembler language
- Call instruction API for IPv4 or IPv6 socket applications
- z/OS Communications Server socket API for REXX
- Pascal language for IPv4 socket API

Clients and servers wanting to transfer data have many calls from which to choose. The `read()` and `write()`, `readv()` and `writv()`, and the `send()` and `recv()` calls can be used only on sockets that are connected. The `sendto()` and `recvfrom()`, and `sendmsg()` and `recvmsg()` calls can be used at any time. The example listed in [Figure 31 on page 56](#) illustrates the use of `send()` and `recv()` calls:

```
int send(int socket, char *buf, int buflen, int flags);
int recv(int socket, char *buf, int buflen, int flags);
.
.
.
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 31. An application using the `send()` and `recv()` calls

The example in [Figure 31 on page 56](#) shows an application sending data to 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. For more information about these routines, see the following information:

- [“read\(\) ” on page 134](#)
- [“readv\(\) ” on page 135](#)
- [“recv\(\) ” on page 136](#)
- [“send\(\) ” on page 144](#)
- [“write\(\) ” on page 162](#)
- [“writv\(\) ” on page 163](#)

There are three groups of calls to use for reading and writing data over sockets:

read and write

These calls can be used only with connected sockets. No processing flags can be passed on these calls.

recv and send

These calls also work with connected sockets only. You can pass processing flags on these calls:

- NOFLAG - Read or write data as a read call or a write call would.
- OOB - Read or write Out Of Band data (expedited data).
- PEEK - Peek at data, but do not remove data from the buffers.

recvfrom and sendto

These calls work with both connected and non-connected sockets. You can pass addressing information directly (as parameters) on these calls. The available flags are the same as those for recv and send.

A connected socket is either a stream socket for which a connection has been established, or it is a datagram socket for which you have issued a connect() call to specify the remote datagram socket address.

Reading and writing data from and to a socket

Stream sockets during read and write calls might behave in a way that you would expect to be an error. The read() call might return fewer bytes, and the write() call may write fewer bytes, than requested. This is not an error, but a normal situation that your programs must deal with when they read or write data over a socket.

You might need to use a series of read calls to read a given number of bytes from a stream socket. Each successful read() call returns in the retcode field the number of bytes actually read. If you know you have to read, for example, 4000 bytes and the read call returns 2500, you have to reissue the read call with a new requested length of 4000 minus the 2500 already received (1500).

If you develop your program in COBOL, the following example shows an implementation of such logic. In this example, the message to be read has a fixed size of 8192 bytes:

```
*-----*
* Variables used by the READ call                                     *
*-----*
01 read-request-read          pic 9(8) binary value 0.
01 read-request-remaining     pic 9(8) binary value 0.
01 read-buffer.
   05 read-buffer-total       pic x(8192) value space.
   05 read-buffer-byte redefines read-buffer-total
                                   pic x occurs 8192 times.
*-----*
* Read 8K block from server                                           *
*-----*
move zero to read-request-read.
move 8192 to read-request-remaining.
Perform until read-request-remaining = 0
    call 'EZASOKET' using socket-read
        socket-descriptor
        read-request-remaining
        read-buffer-byte(read-request-read + 1)
        errno
        retcode
    if retcode < 0 then
        - process error and exit -
    end-if
    add retcode to read-request-read
    subtract retcode from read-request-remaining
    if retcode = 0 then
        Move zero to read-request-remaining
    end-if
end-perform.
```

An actual execution of the program, following the above logic, used four read calls to retrieve 8K of data. The first call returned 1960 bytes, the second call 3920 bytes, the third call 1960 bytes and the final call 352 bytes. It is not possible to predict how many calls will be needed to retrieve the message. That depends on the internal buffer utilization of a TCP/IP. In some cases, only two calls were needed to retrieve 8K of data.

It is good programming practice, whenever you know the number of bytes to read, to issue read calls imbedded in logic, which is similar to the method described above.

If you work with short messages, you usually receive the full message on the first read() call, but there is no guarantee.

The behavior of a write() call is similar to that of a read() call. You might need to repeat more write() calls to write out all the data you want written. The following example illustrates this technique.

```
*-----*
* Buffer and length fields for write operation *
*-----*
01  send-request-sent          pic 9(8) binary value 0.
01  send-request-remaining     pic 9(8) binary value 0.
01  send-buffer.
    05  send-buffer-total      pic x(8192) value space.
    05  send-buffer-byte redefines send-buffer-total
                                pic x occurs 8192 times.
*-----*
* Send 8K data block *
*-----*
    move 8192 to send-request-remaining.
    move 0 to send-request-sent.
    Perform until send-request-remaining = 0
        call 'EZASOCKET' using socket-write
            socket-descriptor
            send-request-remaining
            send-buffer-byte(send-request-sent + 1)
            errno
            retcode
        if retcode < 0 then
            - process error and exit -
        end-if
        add retcode to send-request-sent
        subtract retcode from send-request-remaining
    if retcode = 0 then
        Move zero to send-request-remaining
    end-if
end-perform.
```

Using sendto() and recvfrom() calls

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 of the sendto() and recvfrom() calls is listed in [Figure 32 on page 59](#):

```

int sendto(int socket, char *buf, int buflen, int flags,
           struct sockaddr *addr, int addrlen);
int recvfrom(int socket, char *buf, int buflen, int flags,
             struct sockaddr *addr, int *addrlen);
:
int bytes_sent;
int bytes_received;
char data_sent[256];
char data_received[256];
struct sockaddr_in to;
struct sockaddr from;
int addrlen;
int s;
:
memset(&to, 0, sizeof(to));
to.sin_family = AF_INET;
to.sin_addr = inet_addr("129.5.24.1");
to.sin_port = htons(1024);
:
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, &from, &addrlen)

```

Figure 32. An application using the sendto() and recvfrom() Calls

The sendto() and recvfrom() calls take additional parameters to allow the caller to specify the recipient of the data, or to be notified of the sender of the data. See [“recvfrom\(\)” on page 137](#), [“sendmsg\(\)” on page 145](#), and [“sendto\(\)” on page 147](#) for more information about these additional parameters. Usually, sendto() and recvfrom() are used for datagram sockets, and send() and recv() are used for stream sockets.

Chapter 8. Designing IPv6 programs

The following information contain details on how to enable an IPv6 application:

- [z/OS Communications Server: IPv6 Network and Appl Design Guide](#)
- RFC 2553, *Basic Socket Interface Extensions for IPv6*. The Basic Socket API extension covers the socket calls that the majority of TCP/IP applications use.
- See [“Introducing TCP/IP concepts: Socket libraries” on page 6](#) for information on which APIs support IPv6.
- See [Appendix C, “Address family cross reference,” on page 761](#) for information about which commands support IPv6. Refer to the description and syntax for each command that was enhanced for IPv6 support.

Chapter 9. Designing multicast programs

This topic describes IP multicasting and how an application can exploit multicasting using the TCP/IP socket APIs. IP multicasting concepts in IPv4 and IPv6 protocols are very similar; however there are some differences, such as the IP addresses used for multicasting with each protocol. The topic that follows introduces the basic concepts for IP multicasting with an emphasis on IPv4. However, most of the concepts described here apply to IPv6 multicast applications as well. A more detailed description of IPv6 multicast options follows in the next topic.

IPv4 has three types of IP addresses: unicast, broadcast, and multicast. When an IP datagram is sent to an individual IP address, it is called a unicast IP datagram. The process of sending the datagram is called unicasting. Unicasting is used when two IP nodes are communicating with each other.

When an IP datagram is sent to all nodes on a specific network, it is called broadcasting. Broadcasting support can be both limited and directed.

Multicasting is used to send an IP datagram to a group of systems identified by a class D address. The class D address is used as the destination address. When an application program requests that it receive datagrams with a particular class D destination IP address, it is said to have joined a multicast group. Multicast datagrams (datagrams with a class D destination address) are discarded by a host system unless an application on that host has joined the matching multicast group. The UDP application must bind in order to receive multicast datagrams, after which the application can then receive an IP datagram. The application can receive an IP datagram in two ways:

- The application must bind to the same port that is being used by the sender of the multicast datagram.
- The application can bind to a unicast address, `inaddr_any`, or to a class D address. However, if multiple applications need to receive datagrams for the same multicast group, they should bind to the class D address and set the `SO_REUSEADDR` socket option.

When a host is added to a group that group is referred to as a *host group*. A host group may span multiple networks. Hosts may join and leave a host group as necessary and there is no restriction to the number of hosts involved in a group. A host does not have to belong to a group to send a message to that group. Any hosts on an IP Internet can join a multicast group. The hosts need not be on a single LAN and may be separated by routers. When an application joins a group, it joins the multicast group on a specific interface. Routers use this information to determine if multicast datagrams should be forwarded from one interface to another.

Routers and hosts use a multicast routing protocol called Internet Group Management Protocol (IGMP) to share information about multicast groups. Through this protocol, hosts inform routers when they join or leave a multicast group. Routers can query hosts about groups they have joined and use this information in determining whether to forward multicast datagrams. Some multicast group addresses are referred to as permanent host groups. These addresses are assigned by the Internet Assigned Numbers Authority group as well-known addresses similar to the well-known TCP and UDP port numbers. For example, 224.0.0.1 means all systems on this subnet, and 224.0.0.2 means all routers on this subnet. For a review of The Internet Assigned Number RFC to familiarize yourself with more of the well-known standard multicast address see <http://www.iana.org/assignments/multicast-addresses> for IPv4 multicast address assignments and RFC 2375 for IPv6 multicast address assignments.

Note: z/OS (OMPROUTE) does not support a multicast routing protocol.

Designing multicast programs: Multicast source filters

Source filter APIs enable an application to filter the datagrams that it receives based on the source address. There are two categories of source filter APIs: Basic and advanced. Both categories enable multicast receiver applications to designate the unicast addresses (source addresses) and the multicast group (destination address).

Basic (delta-based) APIs

Some applications need the simplicity of a delta-based API in which each function call specifies a single source address to be added to or removed from the filter. Such applications typically fall into the following types:

Any-source multicast

By default, all source addresses are accepted. Individual source addresses can be turned off and back on as needed. This type is also known as the exclude mode, because the source filter contains a list of excluded sources. The following SETSOCKOPT options are included.

Address family	SETSOCKOPT options
IPv4	IP_ADD_MEMBERSHIP
	IP_BLOCK_SOURCE
	IP_DROP_MEMBERSHIP
	IP_UNBLOCK_SOURCE
Protocol independent	MCAST_BLOCK_SOURCE
	MCAST_JOIN_GROUP
	MCAST_LEAVE_GROUP
	MCAST_UNBLOCK_SOURCE

Source-specific multicast

Only the source addresses that are specified in a list are accepted. The list is initially empty; IP addresses can be added to or deleted from the list one at a time. This filter type also is known as the include mode, because the source filter contains a list of included sources. The following SETSOCKOPT options are included.

Address family	SETSOCKOPT options
IPv4	IP_ADD_SOURCE_MEMBERSHIP
	IP_DROP_MEMBERSHIP
	IP_DROP_SOURCE_MEMBERSHIP
Protocol independent	MCAST_JOIN_SOURCE_GROUP
	MCAST_LEAVE_GROUP
	MCAST_LEAVE_SOURCE_GROUP

Advanced (full-state) APIs

These APIs enable an application to use a source filter that is comprised of zero or more source addresses. The application can retrieve the current filter or replace it with a new filter. The following IOCTL commands are included:

- SIOCGIPMSFILTER
- SIOCGMSFILTER
- SIOCSIPMSFILTER
- SIOCSMSFILTER

Consider the following points about multicast source filters:

- Sockets cannot mix IPv4 group-membership APIs with protocol-independent group-membership APIs. Within each API type, delta-based options cannot be mixed with full-state options. Mixing these calls for the same socket results in an EINVAL error.

- The number of groups that can be joined depends on the socket type; there is a maximum of 20 groups for UDP sockets and a maximum of 256 groups for RAW sockets. Exceeding this limit results in an ETOOMANYREFS error.
- Within each group or group and interface pair, an application can use calls for only one type of basic API, either any-source multicast or source-specific multicast. Mixing options will result in an EINVAL error. However, an application can use different methods for different sockets.
- If the filter mode is set to include and the source list is empty, then the entry corresponding to the requested interface and multicast address is deleted, if present. If no such entry is present, then the request is ignored.
- For each socket, you can specify a maximum of 64 source filters for each multicast address and interface address pair. If the call causes the number of filters to exceed this maximum, an ENOBUFS error is returned.

Tip: z/OS UNIX Assembler Callable Services and z/OS Language Environment C/C++ APIs also support the multicast source filter APIs. See [z/OS C/C++ Runtime Library Reference](#) and [z/OS UNIX System Services Programming: Assembler Callable Services Reference](#) for more information.

Designing multicast programs: IPv4 multicast options

IPv4 multicast supports the following socket options for the Macro, Callable, and REXX Sockets APIs:

- IP_ADD_MEMBERSHIP
- IP_ADD_SOURCE_MEMBERSHIP
- IP_BLOCK_SOURCE
- IP_DROP_MEMBERSHIP
- IP_DROP_SOURCE_MEMBERSHIP
- IP_MULTICAST_IF
- IP_MULTICAST_LOOP
- IP_MULTICAST_TTL
- IP_UNBLOCK_SOURCE

IPv4 multicast supports the following socket options for the C Sockets API:

- IP_ADD_MEMBERSHIP
- IP_DROP_MEMBERSHIP
- IP_MULTICAST_IF
- IP_MULTICAST_LOOP
- IP_MULTICAST_TTL

Use the C, Macro, Callable, or REXX Sockets API SETSOCKOPT call to set these options. Use the C, Macro, Callable or REXX Sockets API GETSOCKOPT call to get the current settings. The status of the IP_ADD_MEMBERSHIP, IP_DROP_MEMBERSHIP, IP_ADD_SOURCE_MEMBERSHIP, IP_DROP_SOURCE_MEMBERSHIP, IP_BLOCK_SOURCE, and IP_UNBLOCK_SOURCE options are exceptions, because they are SETSOCKOPT options only.

IP_ADD_MEMBERSHIP and IP_DROP_MEMBERSHIP

Use the IP_ADD_MEMBERSHIP option to join an IPv4 multicast group on a local IPv4 interface. Use the SETSOCKOPT API and specify the address of the IP_MREQ structure that contains these addresses. The application can join multiple multicast groups on a single socket; it also can join the same group on multiple interfaces on the same socket. However, there is a maximum limit of 20 groups for a single UDP socket, and there is a maximum limit of 256 groups for a single RAW socket. The stack chooses a default multicast interface if an interface with the value 0 is passed. The format of the IP_MREQ structure is in the BPXYSOCK macro. The assembler program example in [Figure 33 on page 68](#) shows this socket option using the EZASMI macro:

```

*****
*
*          Issue INITAPI to connect to interface
*
*****
      POST ECB,1          NEXT IS ALWAYS SYNCH
      EZASMI TYPE=INITAPI,  ISSUE INITAPI MACRO          X
          SUBTASK=SUBTASK,  SPECIFY SUBTASK IDENTIFIER    X
          MAXSOC=MAXSOC,    SPECIFY MAXIMUM NUMBER OF SOCKETS X
          MAXSNO=MAXSNO,    (HIGHEST SOCKET NUMBER ASSIGNED) X
          ERRNO=ERRNO,      (SPECIFY ERRNO FIELD)          X
          RETCODE=RETCODE,  (SPECIFY RETCODE FIELD)        X
          APITYPE=APITYPE,  (SPECIFY APITYPE FIELD)        X
          ERROR=ERROR      ABEND IF ERROR ON MACRO
      BAL  R14,RCHECK      --> DID IT WORK?

```

```

*****
*
*          Issue SOCKET Macro to obtain a datagram socket descriptor
*
*****
      EZASMI TYPE=SOCKET,  ISSUE SOCKET MACRO          X
          AF='INET',      INET OR IUCV                X
          SOCTYPE='DATAGRAM', DATAGRAM(UDP)            X
          PROTO=ZERO,      PROTOCOL                    X
          ERRNO=ERRNO,      (SPECIFY ERRNO FIELD)      X
          RETCODE=RETCODE,  (SPECIFY RETCODE FIELD)    X
          ERROR=ERROR      ABEND IF MACRO ERROR
      BAL  R14,RCHECK      CHECK FOR SUCCESSFUL CALL
*-----*
*          Get socket descriptor number
*-----*
      STH  R8,S          SAVE RETCODE (=SOCKET DESCRIPTOR)

```

```

*****
*
*          ISSUE GETHOSTID CALL
*
*****
      EZASMI TYPE=GETHOSTID,  ISSUE GETHOSTID MACRO    X
          RETCODE=RETCODE,    (SPECIFY RETCODE FIELD)  X
          ERROR=ERROR        ABEND IF MACRO ERROR
      BAL  R14,RCHECK      CHECK FOR SUCCESSFUL CALL
      ST   R8,ADDR        SAVE OUR ID

```

```

*****
*
*          Issue SETSOCKOPT to allow multiple application on the same
*          stack to bind to the same multicast address and port.
*
*****
      EZASMI TYPE=SETSOCKOPT,  ISSUE SETSOCKOPT        X
          S=S,                SOCKET DESCRIPTOR        X
          OPTLEN=OPTLEN4,     OPTION LENGTH            X
          OPTNAME='SO_REUSEADDR', OPTION NAME          X
          OPTVAL=OPTVALON,    OPTION VALUE             X
          RETCODE=RETCODE,    (SPECIFY RETCODE FIELD)  X
          ERROR=ERROR        ABEND IF MACRO ERROR
      BAL  R14,RCHECK      --> CHECK IT

```

```

*****
*
*          Issue BIND socket
*
*****
      MVC  PORT(2),PORTS      Load port #
      MVC  ADDRESS(4),ADDR    Load IP address
      EZASMI TYPE=BIND,      ISSUE BIND MACRO          X
          S=S,              DATAGRAM                  X
          NAME=NAME,        SOCKET ADDRESS STRUCTURE  X
          RETCODE=RETCODE,  (SPECIFY RETCODE FIELD)    X
          ERROR=ERROR      ABEND IF MACRO ERROR
      BAL  R14,RCHECK      CHECK FOR SUCCESSFUL CALL
*
* Here you will add code to set the multicast interface, time-to-live,
* or determine if outgoing datagrams are copied to loopback. See the

```

```
* next sections for the details.
*
```

```
*****
*
*      Issue SETSOCKOPT - IP_ADD_MEMBERSHIP
*
*****
MVC    IMR_MULTIADD,MY_MULTICAST_ADDRESS
MVC    IMR_INTERFAC,MY_MULTICAST_INTERFACE
EZASMI TYPE=SETSOCKOPT,  ISSUE SETSOCKOPT
      S=S,                SOCKET DESCRIPTOR
      OPTLEN=OPTLEN8,      OPTION LENGTH
      OPTNAME='IP_ADD_MEMBERSHIP', OPTION NAME
      OPTVAL=IP_MREQ,      OPTION VALUE
      RETCODE=RETCODE,     (SPECIFY RETCODE FIELD)
      ERROR=ERROR          ABEND IF MACRO ERROR
BAL    R14,RCHECK         --> CHECK IT
      X
      X
      X
      X
      X
      X
```

```
*
* Here your program will perform normal processing such as sending or
* receiving message.
*
```

```
*****
*
*      Issue SETSOCKOPT - IP_DROP_MEMBERSHIP
*
*****
MVC    IMR_MULTIADD,MY_MULTICAST_ADDRESS
MVC    IMR_INTERFAC,MY_MULTICAST_INTERFACE
EZASMI TYPE=SETSOCKOPT,  ISSUE SETSOCKOPT
      S=S,                SOCKET DESCRIPTOR
      OPTLEN=OPTLEN8,      OPTION LENGTH
      OPTNAME='IP_DROP_MEMBERSHIP', OPTION NAME
      OPTVAL=IP_MREQ,      OPTION VALUE
      RETCODE=RETCODE,     (SPECIFY RETCODE FIELD)
      ERROR=ERROR          ABEND IF MACRO ERROR
BAL    R14,RCHECK         --> CHECK IT
      X
      X
      X
      X
      X
      X
```

```
*****
*
*      Terminate Connection to API
*
*****
POST ECB,1          FOLLOWING IS ALWAYS SYNCH
EZASMI TYPE=TERMAPI  ISSUE EZASMI MACRO FOR TERMAPI API
*
* GETSOCKOPT and SETSOCKOPT parms
*
OPTLEN1 DC    F'1'
OPTLEN4 DC    F'4'
OPTLEN8 DC    F'8'
OPTLEN12 DC   F'12'
*
OPTVAL4 DC    CL4' '
SAMEINTERFACE DC F'0'
SAMESUBNET DC  F'1'
SAMESITE DC    F'32'
SAMEREGION DC  F'64'
OPTVALON DC    F'1'          OPTVAL field ON
OPTVALOFF DC   F'0'          OPTVAL field OFF
*
* BIND PARMS
*
CNOPI 0,4
NAME   DC    0CL16' '        SOCKET NAME STRUCTURE
      DC    AL2(2)          FAMILY
PORT   DC    H'0'           PORT
ADDRESS DC    F'0'           IP ADDRESS
      DC    XL8'00'         RESERVED
ADDR   DC    AL1(224),AL1(9),AL1(9),AL1(9) IP ADDRESS TO BIND
PORTS  DC    H'11007'        PORT TO BIND
*
* My multicast address, source, and interface
*
MY_MULTICAST_ADDRESS DC AL1(224),AL1(9),AL1(9),AL1(9)
      X
```

```

                                Multicast address
MY_MULTICAST_INTERFACE DC AL1(204),AL1(59),AL1(83),AL1(19) Internet
*                               address
MY_MULTICAST_SOURCE DC AL1(203),AL1(12),AL1(83),AL1(19) Internet
*                               address
* Multicast Interface
MULTIFA DC AL1(204),AL1(59),AL1(83),AL1(19) Internet Address
*
MULTIFO DC CL4' ' SOCKET MULTICAST INTERFACE OUTPUT
          BPXYSOCK DSECT=NO,LIST=YES
          IP_MREQ DS 0F 01-BPXYSOCK
          IMR_MULTIADDR DS CL4 IP MULTICAST ADDR OF GROUP 01-BPXYSOCK
          IMR_INTERFACE DS CL4 LOCAL IP ADDR OF INTERFACE 01-BPXYSOCK
          IP_MREQ_SOURCE DS 0F 01-BPXYSOCK
          IMRS_MULTIADDR DS CL4 IP MULTICAST ADDR OF GROUP 01-BPXYSOCK
          IMRS_SOURCEADDR DS CL4 IP ADDRESS OF SOURCE 01-BPXYSOCK
          IMRS_INTERFACE DS CL4 LOCAL IP ADDR OF INTERFACE 01-BPXYSOCK

```

Figure 33. *IP_ADD_MEMBERSHIP* and *IP_DROP_MEMBERSHIP*

To remove the host from the multicast host group you must issue a `SETSOCKOPT` call with the `IP_DROP_MEMBERSHIP` option. Using this call is similar to using the `IP_ADD_MEMBERSHIP` option; both use the `IP_MREQ` structure to declare the IPv4 multicast address and the local IPv4 address interface. You can also use the `IP_DROP_MEMBERSHIP` option to remove all sources for a given multicast group (see [Figure 33 on page 68](#)).

While the application is a member of the multicast host group, datagrams can be sent or received as required. To see the multicast groups that are joined on an interface, use the `Netstat DEvlinks/-d` command. To see the multicast groups that are joined on a socket, use the `Netstat ALL/-A` command.

IP_ADD_SOURCE_MEMBERSHIP and IP_DROP_SOURCE_MEMBERSHIP

Use the `IP_ADD_SOURCE_MEMBERSHIP` option to join an IPv4 multicast group on an IPv4 interface and specify the IPv4 source-filter address. Set these values by using the `SETSOCKOPT` API and specifying the address of the `IP_MREQ_SOURCE` structure that contains these addresses. The application can join multiple source multicast groups on a single socket and can also join the same group on multiple interfaces on the same socket. However, there is a maximum limit of 20 groups per single UDP socket and there is a maximum limit of 256 groups per single RAW socket. The stack chooses a default multicast interface if an interface with the value 0 is passed. The format of the `IP_MREQ_SOURCE` structure is in the `BPXYSOCK` macro. The assembler program example in [Figure 34 on page 69](#) shows this socket option using the `EZASMI` macro:


```

*****
*
*      Issue SETSOCKOPT - IP_ADD_SOURCE_MEMBERSHIP
*
*****
MVC  IMRS_MULTIADD,MY_MULTICAST_ADDRESS
MVC  IMRS_SOURCEADDR,MY_MULTICAST_SOURCE
MVC  IMRS_INTERFAC,MY_MULTICAST_INTERFACE
EZASMI TYPE=SETSOCKOPT,  ISSUE SETSOCKOPT
      S=S,                SOCKET DESCRIPTOR
      OPTLEN=OPTLEN12,    OPTION LENGTH
      OPTNAME='IP_ADD_SOURCE_MEMBERSHIP', OPTION NAME
      OPTVAL=IP_MREQ_SOURCE, OPTION VALUE
      RETCODE=RETCODE,    (SPECIFY RETCODE FIELD)
      ERROR=ERROR         ABEND IF MACRO ERROR
BAL  R14,RCHECK          --> CHECK IT
*
* Here your program will perform normal processing such as sending or
* receiving message.
*

```

```

*****
*
*      Issue SETSOCKOPT - IP_DROP_SOURCE_MEMBERSHIP
*
*****
MVC  IMR_MULTIADD,MY_MULTICAST_ADDRESS
MVC  IMRS_SOURCEADDR,MY_MULTICAST_SOURCE
MVC  IMRS_INTERFAC,MY_MULTICAST_INTERFACE
EZASMI TYPE=SETSOCKOPT,  ISSUE SETSOCKOPT
      S=S,                SOCKET DESCRIPTOR
      OPTLEN=OPTLEN12,    OPTION LENGTH
      OPTNAME='IP_DROP_SOURCE_MEMBERSHIP', OPTION NAME
      OPTVAL=IP_MREQ_SOURCE, OPTION VALUE
      RETCODE=RETCODE,    (SPECIFY RETCODE FIELD)
      ERROR=ERROR         ABEND IF MACRO ERROR
BAL  R14,RCHECK          --> CHECK IT

```

Figure 34. *IP_ADD_SOURCE_MEMBERSHIP* and *IP_DROP_SOURCE_MEMBERSHIP*

To remove the host from the source multicast host group you must issue a SETSOCKOPT call with the IP_DROP_SOURCE_MEMBERSHIP option. Using this call is similar to using the IP_ADD_SOURCE_MEMBERSHIP option; both use the IP_MREQ_SOURCE structure to declare the IPv4 multicast address, IPv4 source address, and the local IPv4 address interface.

While the application is a member of the source multicast host group, datagrams can be sent or received as required. To see the multicast groups that are joined on an interface, use the Netstat DEvlinks/-d command. To see the multicast groups that are joined on a socket, use the Netstat ALL/-A command.

IP_BLOCK_SOURCE and IP_UNBLOCK_SOURCE

The IP_BLOCK_SOURCE socket option enables the application to block IPv4 multicast packets that have a source address that matches the given IPv4 source address. The specified multicast group must have been joined previously. The IP_UNBLOCK_SOURCE socket option enables the application to unblock a previously blocked source for a given multicast group.

To block or unblock IPv4 multicast packets, use the SETSOCKOPT API and specify the IP_MREQ_SOURCE structure containing IPv4 multicast address, IPv4 source address, and the local IPv4 interface address. The format of the IP_MREQ_SOURCE structure is in the BPXYSOCK macro. The assembler program example in [Figure 35 on page 70](#) shows this socket option using the EZASMI macro:

```

*****
*
*      Issue SETSOCKOPT - IP_BLOCK_SOURCE
*
*****
MVC  IMRS_MULTIADD,MY_MULTICAST_ADDRESS
MVC  IMRS_SOURCEADDR,MY_MULTICAST_SOURCE
MVC  IMRS_INTERFAC,MY_MULTICAST_INTERFACE
EZASMI TYPE=SETSOCKOPT,  ISSUE SETSOCKOPT
      S=S,                SOCKET DESCRIPTOR
      OPTLEN=OPTLEN12,    OPTION LENGTH
      OPTNAME='IP_BLOCK_SOURCE', OPTION NAME
      OPTVAL=IP_MREQ_SOURCE, OPTION VALUE
      RETCODE=RETCODE,    (SPECIFY RETCODE FIELD)
      ERROR=ERROR         ABEND IF MACRO ERROR
BAL  R14,RCHECK          --> CHECK IT
*
* Here your program will perform normal processing such as sending or
* receiving message.
*

```

```

*****
*
*      Issue SETSOCKOPT - IP_UNBLOCK_SOURCE
*
*****
MVC  IMRS_MULTIADD,MY_MULTICAST_ADDRESS
MVC  IMRS_SOURCEADDR,MY_MULTICAST_SOURCE
MVC  IMRS_INTERFAC,MY_MULTICAST_INTERFACE
EZASMI TYPE=SETSOCKOPT,  ISSUE SETSOCKOPT
      S=S,                SOCKET DESCRIPTOR
      OPTLEN=OPTLEN12,    OPTION LENGTH
      OPTNAME='IP_UNBLOCK_SOURCE', OPTION NAME
      OPTVAL=IP_MREQ_SOURCE, OPTION VALUE
      RETCODE=RETCODE,    (SPECIFY RETCODE FIELD)
      ERROR=ERROR         ABEND IF MACRO ERROR
BAL  R14,RCHECK          --> CHECK IT

```

Figure 35. IP_BLOCK_SOURCE and IP_UNBLOCK_SOURCE

While the application is a member of the multicast host group, datagrams can be sent or received as required. To see the multicast groups that are joined on an interface, use the Netstat DEvlinks/-d command. To see the multicast groups that are joined on a socket, use the Netstat ALL/-A command.

IP_MULTICAST_IF

In order to control which interface multicast datagrams will be sent on, the API provides the IP_MULTICAST_IF socket option. This option can be used to set the interface for sending outbound multicast datagrams from the sockets application. Multicast datagrams can be transmitted on only one interface at a time. You can determine the interface being used by the way of the GETSOCKOPT API with IP_MULTICAST_IF as the OPTNAME. Figure 36 on page 71 illustrates the use of IP_MULTICAST_IF by the use of the SETSOCKOPT and GETSOCKOPT APIs.

```

*****
*                                     *
*      Issue SETSOCKOPT/GETSOCKOPT - IP_MULTICAST_IF      *
*                                     *
*****
EZASMI TYPE=SETSOCKOPT,  ISSUE SETSOCKOPT                      X
      S=S,              SOCKET DESCRIPTOR                    X
      OPTLEN=OPTLEN4,   OPTION LENGTH                        X
      OPTNAME='IP_MULTICAST_IF', OPTION NAME                  X
      OPTVAL=MULTIF,    OPTION VALUE                          X
      RETCODE=RETCODE,  (SPECIFY RETCODE FIELD)              X
      ERROR=ERROR      ABEND IF MACRO ERROR
BAL    R14,RCHECK      --> CHECK IT
XC     MULTIFO,MULTIFO
EZASMI TYPE=GETSOCKOPT,  ISSUE GETSOCKOPT                      X
      S=S,              SOCKET DESCRIPTOR                    X
      OPTLEN=OPTLEN4,   OPTION LENGTH                        X
      OPTNAME='IP_MULTICAST_IF', OPTION NAME                  X
      OPTVAL=MULTIFO,   OPTION VALUE                          X
      RETCODE=RETCODE,  (SPECIFY RETCODE FIELD)              X
      ERROR=ERROR      ABEND IF MACRO ERROR
BAL    R14,RCHECK      --> CHECK IT

```

Figure 36. *IP_MULTICAST_IF*

IP_MULTICAST_LOOP

The API uses `IP_MULTICAST_LOOP` socket option to enable or disable the loopback of outgoing multicast datagrams. The default is enabled. This option is used to enable an application with multiple senders and receivers on a system to loop datagrams back so that each process receives the transmissions of the other senders on the system. [Figure 37 on page 72](#) illustrates the use of `IP_MULTICAST_IF` by the use of the `SETSOCKOPT` and `GETSOCKOPT` APIs.

```

*****
*                                     *
*      Issue SETSOCKOPT/GETSOCKOPT - IP_MULTICAST_LOOP ENABLED      *
*                                     *
*****
EZASMI TYPE=SETSOCKOPT,  ISSUE SETSOCKOPT                                X
S=S,                    SOCKET DESCRIPTOR                              X
OPTLEN=OPTLEN1,         OPTION LENGTH                                  X
OPTNAME='IP_MULTICAST_LOOP', OPTION NAME                              X
OPTVAL=OPTVALON,        OPTION VALUE                                  X
RETCODE=RETCODE,        (SPECIFY RETCODE FIELD)                       X
ERROR=ERROR            ABEND IF MACRO ERROR
BAL  R14,RCHECK        --> CHECK IT
*
XC  OPTVAL4,OPTVAL4
EZASMI TYPE=GETSOCKOPT,  ISSUE GETSOCKOPT                                X
S=S,                    SOCKET DESCRIPTOR                              X
OPTLEN=OPTLEN1,         OPTION LENGTH                                  X
OPTNAME='IP_MULTICAST_LOOP', OPTION NAME                              X
OPTVAL=OPTVAL4,        OPTION VALUE                                  X
RETCODE=RETCODE,        (SPECIFY RETCODE FIELD)                       X
ERROR=ERROR            ABEND IF MACRO ERROR
BAL  R14,RCHECK        --> CHECK IT

*****
*                                     *
*      Issue SETSOCKOPT/GETSOCKOPT - IP_MULTICAST_LOOP DISABLED      *
*                                     *
*****
EZASMI TYPE=SETSOCKOPT,  ISSUE SETSOCKOPT                                X
S=S,                    SOCKET DESCRIPTOR                              X
OPTLEN=OPTLEN1,         OPTION LENGTH                                  X
OPTNAME='IP_MULTICAST_LOOP', OPTION NAME                              X
OPTVAL=OPTVALOFF,       OPTION VALUE                                  X
RETCODE=RETCODE,        (SPECIFY RETCODE FIELD)                       X
ERROR=ERROR            ABEND IF MACRO ERROR
BAL  R14,RCHECK        --> CHECK IT
XC  OPTVAL4,OPTVAL4
EZASMI TYPE=GETSOCKOPT,  ISSUE GETSOCKOPT                                X
S=S,                    SOCKET DESCRIPTOR                              X
OPTLEN=OPTLEN1,         OPTION LENGTH                                  X
OPTNAME='IP_MULTICAST_LOOP', OPTION NAME                              X
OPTVAL=OPTVAL4,        OPTION VALUE                                  X
RETCODE=RETCODE,        (SPECIFY RETCODE FIELD)                       X
ERROR=ERROR            ABEND IF MACRO ERROR
BAL  R14,RCHECK        --> CHECK IT

```

Figure 37. IP_MULTICAST_LOOP

IP_MULTICAST_TTL

The IP_MULTICAST_TTL socket option allows the application to primarily limit the lifetime of the packet in the Internet and prevent it from circulating indefinitely. This option also serves to allow the application to specify administrative boundaries. This administrative region is specified in terms such as "this site", "this company", or "this state", and is relative to the starting point of the packet. The region associated with a multicast packet is called its *scope*. The default value is 1, meaning multicast is available only to the local subnet. Figure 38 on page 73 illustrates the use of IP_MULTICAST_TTL by the use of the SETSOCKOPT and GETSOCKOPT APIs.

```

*****
*
*      Issue SETSOCKOPT/GETSOCKOPT - IP_MULTICAST_TTL
*
*****
*
* SET TTL TO SAME SITE
*
      EZASMI TYPE=SETSOCKOPT,   ISSUE SETSOCKOPT           X
          S=S,                 SOCKET DESCRIPTOR         X
          OPTLEN=OPTLEN1,      OPTION LENGTH             X
          OPTNAME='IP_MULTICAST_TTL', OPTION NAME         X
          OPTVAL=SAME SITE,    OPTION VALUE              X
          RETCODE=RETCODE,     (SPECIFY RETCODE FIELD)    X
          ERROR=ERROR          ABEND IF MACRO ERROR
      BAL   R14,RCHECK        --> CHECK IT
*
* DISPLAY TTL, SHOULD BE 32
*
      XC   OPTVAL4,OPTVAL4
      EZASMI TYPE=GETSOCKOPT,   ISSUE GETSOCKOPT           X
          S=S,                 SOCKET DESCRIPTOR         X
          OPTLEN=OPTLEN1,      OPTION LENGTH             X
          OPTNAME='IP_MULTICAST_TTL', OPTION NAME         X
          OPTVAL=OPTVAL4,     OPTION VALUE              X
          RETCODE=RETCODE,     (SPECIFY RETCODE FIELD)    X
          ERROR=ERROR          ABEND IF MACRO ERROR
      BAL   R14,RCHECK        --> CHECK IT

```

Figure 38. IP_MULTICAST_TTL

Designing multicast programs: IPv6 multicast options

To enable your application to support the IPv6 version of multicast support, the following socket options will be discussed:

- IPV6_JOIN_GROUP
- IPV6_LEAVE_GROUP
- IPV6_MULTICAST_IF
- IPV6_MULTICAST_LOOP
- IPV6_MULTICAST_HOPS

Use the Macro, Callable, and REXX Sockets API SETSOCKOPT call to set these options. Use the Macro, Callable, or REXX Sockets API GETSOCKOPT call to get the current settings. The status of the IPV6_JOIN_GROUP and IPV6_LEAVE_GROUP are exceptions as they are SETSOCKOPT options only.

If you want to enable your application to support the IPv6 multicast source filter, see [“Designing multicast programs: Protocol-independent multicast options”](#) on page 78.

IPV6_JOIN_GROUP and IPV6_LEAVE_GROUP

IPV6_JOIN_GROUP is used to join a multicast group. This is accomplished by using the SETSOCKOPT API and specifying the address of the IPV6_MREQ structure containing the IPv6 multicast address and the local IPv6 multicast interface index. The stack chooses a default multicast interface if an interface index of 0 is passed. The values specified in the IPV6_MREQ structure used by IPV6_JOIN_GROUP and IPV6_LEAVE_GROUP must be symmetrical. The format of the IPV6_MREQ structure can be found in the BPXYSOCK macro.

The assembler program example in [Figure 39 on page 76](#) illustrates this socket option in EZASMI Macro form:

```

*****
*
*      Issue INITAPI to connect to interface
*
*****
*
      POST ECB,1          NEXT IS ALWAYS SYNCH
      EZASMI TYPE=INITAPI, ISSUE INITAPI MACRO           X

```

```

SUBTASK=SUBTASK,    SPECIFY SUBTASK IDENTIFIER      X
MAXSOC=MAXSOC,      SPECIFY MAXIMUM NUMBER OF SOCKETS X
MAXSNO=MAXSNO,      (HIGHEST SOCKET NUMBER ASSIGNED) X
ERRNO=ERRNO,        (SPECIFY ERRNO FIELD)             X
RETCODE=RETCODE,    (SPECIFY RETCODE FIELD)           X
APITYPE=APITYPE,    (SPECIFY APITYPE FIELD)           X
ERROR=ERROR         ABEND IF ERROR ON MACRO
BAL R14,RCHECK      --> DID IT WORK?
*****
*
* Issue SOCKET Macro to obtain a socket descriptor
*
*****
EZASMI TYPE=SOCKET,    ISSUE SOCKET MACRO              X
AF='INET6',           INET OR IUCV                     X
SOCTYPE='DATAGRAM',   DATAGRAM(UDP)                   X
PROTO=ZERO,           PROTOCOL                         X
ERRNO=ERRNO,          (SPECIFY ERRNO FIELD)            X
RETCODE=RETCODE,      (SPECIFY RETCODE FIELD)          X
ERROR=ERROR           ABEND IF MACRO ERROR
BAL R14,RCHECK        CHECK FOR SUCCESSFUL CALL
*-----*

* Get socket descriptor number
*-----*
STH R8,S              SAVE RETCODE (=SOCKET DESCRIPTOR)
*****
*
* ISSUE GETHOSTID CALL
*
*****
EZASMI TYPE=GETHOSTID, ISSUE GETHOSTID MACRO           X
RETCODE=RETCODE,      (SPECIFY RETCODE FIELD)         X
ERROR=ERROR           ABEND IF MACRO ERROR
BAL R14,RCHECK        CHECK FOR SUCCESSFUL CALL
ST R8,ADDR            SAVE OUR ID

*****
*
* Issue SETSOCKOPT to allow multiple application on the same
* stack to bind to the same multicast address and port.
*
*****
EZASMI TYPE=SETSOCKOPT, ISSUE SETSOCKOPT              X
S=S,                  SOCKET DESCRIPTOR               X
OPTLEN=OPTLEN4,       OPTION LENGTH                   X
OPTNAME='SO_REUSEADDR', OPTION NAME                   X
OPTVAL=OPTVALON,      OPTION VALUE                    X
RETCODE=RETCODE,      (SPECIFY RETCODE FIELD)         X
ERROR=ERROR           ABEND IF MACRO ERROR
BAL R14,RCHECK        --> CHECK IT

*****
*
* Issue BIND socket
*
*****
MVC PORT(2),PORTS     Load port #
MVC ADDRESS(16),ADDR   Load IPv6 internet address
EZASMI TYPE=BIND,      ISSUE BIND MACRO                X
S=S,                  DATAGRAM                         X
NAME=NAME,            SOCKET ADDRESS STRUCTURE         X
RETCODE=RETCODE,      (SPECIFY RETCODE FIELD)         X
ERROR=ERROR           ABEND IF MACRO ERROR
BAL R14,RCHECK        CHECK FOR SUCCESSFUL CALL

*
* Here you will add code to set the multicast interface, hops,
* or determine if outgoing datagrams are copied to loopback. See the
* next sections for the details.
*

*****
*
* Issue SETSOCKOPT - IPV6_JOIN_GROUP
*
*****

*
* Either hard code a multicast address and index or use the
* SIOCGIFNAMEINDEX IOCTL to obtain the interface index from the stack.

```

```

*
MVC  IV6MR_MULTIADD,MY_MULTICAST_ADDRESS
MVC  IV6MR_INTERFAC,MY_MULTICAST_INTERFACE
EZASMI TYPE=SETSOCKOPT,  ISSUE SETSOCKOPT                                X
      S=S,                SOCKET DESCRIPTOR                            X
      OPTLEN=OPTLEN20,    OPTION LENGTH                                X
      OPTNAME='IPV6_JOIN_GROUP', OPTION NAME                          X
      OPTVAL=IPV6_MREQ,   OPTION VALUE                                X
      RETCODE=RETCODE,    (SPECIFY RETCODE FIELD)                    X
      ERROR=ERROR        ABEND IF MACRO ERROR
BAL  R14,RCHECK          --> CHECK IT

*
* Here your program will perform normal processing such as sending or
* receiving messages.
*

*****
*                                     *
*      Issue SETSOCKOPT - IPV6_LEAVE_GROUP                                *
*                                     *
*****

*
* Either hard code a multicast address and index or use the
* SIOCGIFNAMEINDEX IOCTL to obtain the interface index from the stack.
*

MVC  IV6MR_MULTIADD,MY_MULTICAST_ADDRESS
MVC  IV6MR_INTERFAC,MY_MULTICAST_INTERFACE
EZASMI TYPE=SETSOCKOPT,  ISSUE SETSOCKOPT                                X
      S=S,                SOCKET DESCRIPTOR                            X
      OPTLEN=OPTLEN20,    OPTION LENGTH                                X
      OPTNAME='IPV6_LEAVE_GROUP', OPTION NAME                          X
      OPTVAL=IPV6_MREQ,   OPTION VALUE                                X
      RETCODE=RETCODE,    (SPECIFY RETCODE FIELD)                    X
      ERROR=ERROR        ABEND IF MACRO ERROR
BAL  R14,RCHECK          --> CHECK IT

*****
*                                     *
*      Terminate Connection to API                                        *
*                                     *
*****
      POST ECB,1          FOLLOWING IS ALWAYS SYNCH
EZASMI TYPE=TERMAPI      ISSUE EZASMI MACRO FOR TERMAPI

*
* GETSOCKOPT and SETSOCKOPT parms
*
OPTLEN1 DC  F'1'
OPTLEN4 DC  F'4'
OPTLEN8 DC  F'8'
OPTLEN20 DC F'20'
OPTVAL4 DC  CL4' '
SAMEINTERFACE DC F'0'
SAMESUBNET DC  F'1'
SAMESITE DC  F'32'
SAMEREGION DC  F'64'
OPTVALON DC  F'1'          OPTVAL field ON
OPTVALOFF DC F'0'         OPTVAL field OFF
*
* BIND PARMS
*
NAME      DC  0CL16' '      SOCKET NAME STRUCTURE
          DC  AL2(2)        FAMILY
PORT      DC  H'0'         PORT
FLOWINFO  DC  F'0'         FLOWINFO
ADDRESS   DC  F'0'         IP ADDRESS
SCOPEID   DC  F'0'         SCOPEID
ADDR      DC  XL16'FF0201010101010105050505050505' IP ADDR TO BIND
PORTS     DC  H'11007'     PORT TO BIND
*
* My Multicast address and interface
*
MY_MULTICAST_ADDRESS DC XL16'FF0201010101010505050505050505' X
                        Multicast Address
MY_MULTICAST_INTERFACE DC XL4'0000000E' Interface Index
*
MULTIFO  DC  CL4' '          SOCKET MULTICAST INTERFACE OUTPUT  *
*
BPXYSOCK DSECT=NO,LIST=YES

```

IPV6_MREQ	DS 0F		01-BPXYS
IPV6MR_MULTIADDR	DS CL16	IPv6 Addr	01-BPXYS
IPV6MR_INTERFACE	DS F	Interface Index	01-BPXYS

Figure 39. IPV6_JOIN_GROUP and IPV6_LEAVE_GROUP

IPV6_LEAVE_GROUP is used to remove a host from the multicast group. You must issue a SETSOCKOPT API and specify the address of the IPV6_MREQ structure containing the IPv6 multicast address and the local IPv6 multicast interface index. See also [Figure 39 on page 76](#).

While the application is a member of the multicast host group, datagrams may be sent or received as required. To see the multicast groups that are joined on an interface, use the Netstat DEVlinks/-d command. To see the multicast groups that are joined on a socket, use the Netstat ALL/-A command.

IPV6_MULTICAST_IF

In order to control which interface multicast datagrams will be sent on, the API provides the IPV6_MULTICAST_IF socket option. This option can be used to set the interface for sending outbound multicast datagrams from the sockets application. Multicast datagrams can be transmitted on only one interface at a time. You can determine the interface being used by the way of the GETSOCKOPT API with IPV6_MULTICAST_IF as the OPTNAME. The IPV6_MULTICAST_IF socket option requires that the option value be the value of the IPv6 interface index.

[Figure 40 on page 76](#) illustrates the use of IPV6_MULTICAST_IF by the use of the SETSOCKOPT and GETSOCKOPT APIs.

```
*****
*
*      Issue SETSOCKOPT/GETSOCKOPT - IPV6_MULTICAST_IF
*
*****
EZASMI TYPE=SETSOCKOPT,  ISSUE SETSOCKOPT      X
      S=S,                SOCKET DESCRIPTOR      X
      OPTLEN=OPTLEN4,     OPTION LENGTH          X
      OPTNAME='IPV6_MULTICAST_IF', OPTION NAME    X
      OPTVAL=MY_MULTICAST_INTERFACE, OPTION VALUE X
      RETCODE=RETCODE,    (SPECIFY RETCODE FIELD) X
      ERROR=ERROR         ABEND IF MACRO ERROR
BAL    R14,RCHECK        --> CHECK IT
XC     MULTIFO,MULTIFO
EZASMI TYPE=GETSOCKOPT,  ISSUE GETSOCKOPT      X
      S=S,                SOCKET DESCRIPTOR      X
      OPTLEN=OPTLEN4,     OPTION LENGTH          X
      OPTNAME='IPV6_MULTICAST_IF', OPTION NAME    X
      OPTVAL=MULTIFO,     OPTION VALUE           X
      RETCODE=RETCODE,    (SPECIFY RETCODE FIELD) X
      ERROR=ERROR         ABEND IF MACRO ERROR
BAL    R14,RCHECK        --> CHECK IT
```

Figure 40. IPV6_MULTICAST_IF

IPV6_MULTICAST_LOOP

The API uses IPV6_MULTICAST_LOOP socket option to enable or disable the loopback of outgoing multicast datagrams. The default is enabled. This option is used to enable an application with multiple senders and receivers on a system to loop datagrams back so that each process receives the transmissions of the other senders on the system. [Figure 41 on page 77](#) illustrates the use of IPV6_MULTICAST_LOOP by the use of the SETSOCKOPT and GETSOCKOPT APIs.


```

*****
*
*      Issue SETSOCKOPT/GETSOCKOPT - IPV6_MULTICAST_LOOP ENABLED
*
*****
EZASMI TYPE=SETSOCKOPT,  ISSUE SETSOCKOPT      X
      S=S,              SOCKET DESCRIPTOR      X
      OPTLEN=OPTLEN1,   OPTION LENGTH          X
      OPTNAME='IPV6_MULTICAST_LOOP', OPTION NAME X
      OPTVAL=OPTVAL0N,  OPTION VALUE           X
      RETCODE=RETCODE,  (SPECIFY RETCODE FIELD) X
      ERROR=ERROR      ABEND IF MACRO ERROR
BAL    R14,RCHECK      --> CHECK IT

*
XC     OPTVAL4,OPTVAL4
EZASMI TYPE=GETSOCKOPT,  ISSUE GETSOCKOPT      X
      S=S,              SOCKET DESCRIPTOR      X
      OPTLEN=OPTLEN1,   OPTION LENGTH          X
      OPTNAME='IPV6_MULTICAST_LOOP', OPTION NAME X
      OPTVAL=OPTVAL4,   OPTION VALUE           X
      RETCODE=RETCODE,  (SPECIFY RETCODE FIELD) X
      ERROR=ERROR      ABEND IF MACRO ERROR
BAL    R14,RCHECK      --> CHECK IT

```

```

*****
*
*      Issue SETSOCKOPT/GETSOCKOPT - IPV6_MULTICAST_LOOP DISABLED
*
*****
EZASMI TYPE=SETSOCKOPT,  ISSUE SETSOCKOPT      X
      S=S,              SOCKET DESCRIPTOR      X
      OPTLEN=OPTLEN1,   OPTION LENGTH          X
      OPTNAME='IPV6_MULTICAST_LOOP', OPTION NAME X
      OPTVAL=OPTVAL0FF, OPTION VALUE           X
      RETCODE=RETCODE,  (SPECIFY RETCODE FIELD) X
      ERROR=ERROR      ABEND IF MACRO ERROR
BAL    R14,RCHECK      --> CHECK IT
:
:
XC     OPTVAL4,OPTVAL4
EZASMI TYPE=GETSOCKOPT,  ISSUE GETSOCKOPT      X
      S=S,              SOCKET DESCRIPTOR      X
      OPTLEN=OPTLEN1,   OPTION LENGTH          X
      OPTNAME='IPV6_MULTICAST_LOOP', OPTION NAME X
      OPTVAL=OPTVAL4,   OPTION VALUE           X
      RETCODE=RETCODE,  (SPECIFY RETCODE FIELD) X
      ERROR=ERROR      ABEND IF MACRO ERROR
BAL    R14,RCHECK      --> CHECK IT

```

Figure 41. IPV6_MULTICAST_LOOP

IPV6_MULTICAST_HOPS

The IPV6_MULTICAST_HOPS socket option allows the application to primarily limit the lifetime of the packet in the Internet and prevent it from circulating indefinitely. The default value is 1, meaning multicast is available only to the local subnet.

Figure 42 on page 78 illustrates the use of IPV6_MULTICAST_HOPS by the use of the SETSOCKOPT and GETSOCKOPT APIs.

```

*****
*                                     *
*          Issue SETSOCKOPT/GETSOCKOPT - IPV6_MULTICAST_HOPS          *
*                                     *
*****
* SET TTL TO SAME SITE
*
      EZASMI TYPE=SETSOCKOPT,      ISSUE SETSOCKOPT                      X
      S=S,                        SOCKET DESCRIPTOR                    X
      OPTLEN=OPTLEN1,             OPTION LENGTH                        X
      OPTNAME='IPV6_MULTICAST_HOPS', OPTION NAME                      X
      OPTVAL=SAMESITE,            OPTION VALUE                         X
      RETCODE=RETCODE,            (SPECIFY RETCODE FIELD)             X
      ERROR=ERROR                  ABEND IF MACRO ERROR
      BAL R14,RCHECK               --> CHECK IT

*
* DISPLAY HOPS, SHOULD BE 32
*
      XC OPTVAL4,OPTVAL4
      EZASMI TYPE=GETSOCKOPT,      ISSUE GETSOCKOPT                      X
      S=S,                        SOCKET DESCRIPTOR                    X
      OPTLEN=OPTLEN1,             OPTION LENGTH                        X
      OPTNAME='IPV6_MULTICAST_HOPS', OPTION NAME                      X
      OPTVAL=OPTVAL4,             OPTION VALUE                         X
      RETCODE=RETCODE,            (SPECIFY RETCODE FIELD)             X
      ERROR=ERROR                  ABEND IF MACRO ERROR
      BAL R14,RCHECK               --> CHECK IT

```

Figure 42. IPV6_MULTICAST_HOPS

Designing multicast programs: Protocol-independent multicast options

The following socket options enable your application to support IPv4 and IPv6 multicast:

- MCAST_BLOCK_SOURCE
- MCAST_JOIN_GROUP
- MCAST_JOIN_SOURCE_GROUP
- MCAST_LEAVE_GROUP
- MCAST_LEAVE_SOURCE_GROUP
- MCAST_UNBLOCK_SOURCE

Use the Macro, Callable, and REXX Sockets API SETSOCKOPT call to set these options.

MCAST_JOIN_GROUP and MCAST_LEAVE_GROUP

Use the MCAST_JOIN_GROUP socket option to join a multicast group and set the IPv4 or IPv6 multicast address and the local interface index. Use the SETSOCKOPT API and specify the address of the GROUP_REQ structure that contains the address and the interface index. The application can join multiple multicast groups on a single socket and can also join the same group on multiple interfaces on the same socket. However, there is a maximum limit of 20 groups per single UDP socket and there is a maximum limit of 256 groups per single RAW socket. The stack chooses a default multicast interface if the interface index 0 is passed. The format of the GROUP_REQ structure is in the BPXYSOCK macro. The assembler program example in [Figure 43 on page 79](#) shows this socket option using the EZASMI macro:

```

*****
*
*      Issue SETSOCKOPT - MCAST_JOIN_GROUP
*
*****

EZASMI TYPE=SETSOCKOPT,  ISSUE SETSOCKOPT          X
      S=S,                SOCKET DESCRIPTOR        X
      OPTLEN=OPTL136,      OPTION LENGTH            X
      OPTNAME='MCAST_JOIN_GROUP', OPTION NAME        X
      OPTVAL=GROUP_REQ1,   OPTION VALUE             X
      ERRNO=ERRNO,         (SPECIFY ERRNO FIELD)     X
      RETCODE=RETCODE,     (SPECIFY RETCODE FIELD)   X
      ERROR=ERROR          ABEND IF MACRO ERROR
BAL    R14,RCHECK        --> CHECK IT

*
* Here your program will perform normal processing such as sending or
* receiving message.
*

*****
*
*      Issue SETSOCKOPT - MCAST_LEAVE_GROUP
*
*****

EZASMI TYPE=SETSOCKOPT,  ISSUE SETSOCKOPT          X
      S=S,                SOCKET DESCRIPTOR        X
      OPTLEN=OPTL136,      OPTION LENGTH            X
      OPTNAME='MCAST_LEAVE_GROUP', OPTION NAME        X
      OPTVAL=GROUP_REQ1,   OPTION VALUE             X
      ERRNO=ERRNO,         (SPECIFY ERRNO FIELD)     X
      RETCODE=RETCODE,     (SPECIFY RETCODE FIELD)   X
      ERROR=ERROR          ABEND IF MACRO ERROR
BAL    R14,RCHECK        --> CHECK IT

*
* GETSOCKOPT and SETSOCKOPT pairs
*
OPTLEN1 DC F'1'
OPTLEN4 DC F'4'
OPTLEN8 DC F'8'
OPTL136 DC F'136'
OPTL264 DC F'264'

*-----*
* Group_Req Structure
*-----*
GROUP_REQ1 DS 0CL136
GR_INTF1   DC XL4'00010005'      Interface Index
          DS CL4                  Padding
GR_GROUP1  DS 0CL128            Group Address
          DC XL16'10020000E01111110000000000000000'
          DC XL112'00'

*-----*
* Group_Source_Req Structure
*-----*
GSR_REQ1   DS 0CL264
GSR_INTF1  DC XL4'00010005'      Interface Index
          DS CL4                  Padding
GSR_GROUP1 DS 0CL128            Group Address
          DC XL16'10020000E01111110000000000000000'
          DC XL112'00'
GSR_SRCAD1 DS 0CL128            Source Address
          DC XL16'10020000424242000000000000000000'
          DC XL112'00'

```

Figure 43. MCAST_JOIN_GROUP and MCAST_LEAVE_GROUP

To remove the host from the multicast host group you must issue a SETSOCKOPT call with the MCAST_LEAVE_GROUP option. Using this call is similar to using the MCAST_JOIN_GROUP option because it also uses the GROUP_REQ structure to declare the multicast address and the local interface index. You can also use MCAST_LEAVE_GROUP option to remove all sources for a given multicast group.

While the application is a member of the multicast host group, datagrams can be sent or received as required. To see the multicast groups that are joined on an interface, use the Netstat DEvlinks/-d command. To see the multicast groups that are joined on a socket, use the Netstat ALL/-A command.

MCAST_JOIN_SOURCE_GROUP and MCAST_LEAVE_SOURCE_GROUP

Use the MCAST_JOIN_SOURCE_GROUP option to set the IPv4 or IPv6 multicast address, source address, and the local interface index. Use the SETSOCKOPT API and specify the address of the GROUP_SOURCE_REQ structure that contains these addresses and the interface index. The application can join multiple source multicast groups on a single socket and can also join the same group on multiple interfaces on the same socket. However, there is a maximum limit of 20 groups per single UDP socket and there is a maximum limit of 256 groups per single RAW socket. The stack chooses a default multicast interface if the interface index 0 is passed. The format of the GROUP_SOURCE_REQ structure is in the BPXYSOCK macro. The assembler program example in [Figure 44 on page 80](#) shows this socket option using the EZASMI macro:

```
*****
*
*      Issue SETSOCKOPT - MCAST_JOIN_SOURCE_GROUP
*
*****
EZASMI TYPE=SETSOCKOPT,  ISSUE SETSOCKOPT          X
      S=S,                SOCKET DESCRIPTOR        X
      OPTLEN=OPTL264,      OPTION LENGTH            X
      OPTNAME='MCAST_JOIN_SOURCE_GROUP', OPTION NAME X
      OPTVAL=GSR_REQ1,     OPTION VALUE             X
      ERRNO=ERRNO,         (SPECIFY ERRNO FIELD)     X
      RETCODE=RETCODE,     (SPECIFY RETCODE FIELD)   X
      ERROR=ERROR          ABEND IF MACRO ERROR
BAL    R14,RCHECK        --> CHECK IT

*
* Here your program will perform normal processing such as sending or
* receiving message.
*
*****
*
*      Issue SETSOCKOPT - MCAST_LEAVE_SOURCE_GROUP
*
*****
EZASMI TYPE=SETSOCKOPT,  ISSUE SETSOCKOPT          X
      S=S,                SOCKET DESCRIPTOR        X
      OPTLEN=OPTL264,      OPTION LENGTH            X
      OPTNAME='MCAST_LEAVE_SOURCE_GROUP', OPTION NAME X
      OPTVAL=GSR_REQ1,     OPTION VALUE             X
      ERRNO=ERRNO,         (SPECIFY ERRNO FIELD)     X
      RETCODE=RETCODE,     (SPECIFY RETCODE FIELD)   X
      ERROR=ERROR          ABEND IF MACRO ERROR
BAL    R14,RCHECK        --> CHECK IT
```

Figure 44. MCAST_JOIN_SOURCE_GROUP and MCAST_LEAVE_SOURCE_GROUP

To remove the host from the source multicast host group you must issue a SETSOCKOPT call with the MCAST_LEAVE_SOURCE_GROUP option. This call is similar to the MCAST_JOIN_SOURCE_GROUP option because it also uses the GROUP_SOURCE_REQ structure to declare the IPv4 or IPv6 multicast address, source address, and the local interface index (see [Figure 44 on page 80](#)). You can also use the MCAST_LEAVE_GROUP option to remove all sources for a given multicast group.

While the application is a member of the source multicast host group, datagrams can be sent or received as required. To see the multicast groups that are joined on an interface, use the Netstat DEvlinks/-d command. To see the multicast groups that are joined on a socket, use the Netstat ALL/-A command.

MCAST_BLOCK_SOURCE and MCAST_UNBLOCK_SOURCE

The MCAST_BLOCK_SOURCE socket option enables the application to block IPv4 or IPv6 multicast packets that have a source address that matches the given source address. The specified multicast group must have been joined previously. The MCAST_UNBLOCK_SOURCE socket option enables the application

to unblock a previously blocked source for a given source multicast group. Use the SETSOCKOPT API and specify the GROUP_SOURCE_REQ structure that contains the multicast address, the source address, and the local interface address. The format of the GROUP_SOURCE_REQ structure is in the BPXYSOCK macro. The assembler program example in [Figure 45 on page 81](#) shows the socket option using the EZASMI macro:

```
*****
*                                                                 *
*          Issue SETSOCKOPT - MCAST_BLOCK_SOURCE                  *
*                                                                 *
*****

EZASMI TYPE=SETSOCKOPT,      ISSUE SETSOCKOPT                    X
      S=S,                  SOCKET DESCRIPTOR                    X
      OPTLEN=OPTL264,        OPTION LENGTH                        X
      OPTNAME='MCAST_BLOCK_SOURCE', OPTION NAME                  X
      OPTVAL=GSR_REQ1,       OPTION VALUE                         X
      ERRNO=ERRNO,           (SPECIFY ERRNO FIELD)                X
      RETCODE=RETCODE,       (SPECIFY RETCODE FIELD)              X
      ERROR=ERROR            ABEND IF MACRO ERROR                 X
BAL    R14,RCHECK           --> CHECK IT

*
* Here your program will perform normal processing such as sending or
* receiving message.
*

*****
*                                                                 *
*          Issue SETSOCKOPT - MCAST_UNBLOCK_SOURCE                *
*                                                                 *
*****

EZASMI TYPE=SETSOCKOPT,      ISSUE SETSOCKOPT                    X
      S=S,                  SOCKET DESCRIPTOR                    X
      OPTLEN=OPTL264,        OPTION LENGTH                        X
      OPTNAME='MCAST_UNBLOCK_SOURCE', OPTION NAME                  X
      OPTVAL=GSR_REQ1,       OPTION VALUE                         X
      ERRNO=ERRNO,           (SPECIFY ERRNO FIELD)                X
      RETCODE=RETCODE,       (SPECIFY RETCODE FIELD)              X
      ERROR=ERROR            ABEND IF MACRO ERROR                 X
BAL    R14,RCHECK           --> CHECK IT
```

Figure 45. MCAST_BLOCK_SOURCE and MCAST_UNBLOCK_SOURCE

While the application is a member of the multicast host group, datagrams can be sent or received as required. To see the multicast groups that are joined on an interface, use the Netstat DEvlinks/-d command. To see the multicast groups that are joined on a socket, use the Netstat ALL/-A command.

Designing multicast programs: IOCTL multicast commands

The following IOCTL commands enable applications to support the advanced (full-state) multicast API for IPv4 and IPv6 addresses:

- SIOCGIPMSFILTER
- SIOCGMSFILTER
- SIOCSIPMSFILTER
- SIOCSMSFILTER

Use the Macro, Callable, and REXX Sockets API IOCTL call for these options.

SIOCGIPMSFILTER

An SIOCGIPMSFILTER IOCTL enables an application to retrieve a list of the IPv4 source addresses that comprise the source filter, with the current mode on a given interface and a multicast group for a socket. The source filter can include or exclude the set of source addresses, depending on the filter mode (MCAST_INCLUDE or MCAST_EXCLUDE), which is defined in the IP_MSFILTER structure of the BPXYIOCC macro.

SIOCGMSFILTER

An SIOCGMSFILTER IOCTL enables an application to retrieve a list of the IPv4 or IPv6 source addresses that comprise the source filter along with the current mode on a given interface index and a multicast group for a socket. The source filter may either include or exclude the set of source address, depending on the filter mode (MCAST_INCLUDE or MCAST_EXCLUDE), which is defined in the GROUP_FILTER structure of the BPXYIOCC macro.

SIOCSIPMSFILTER

An SIOCSIPMSFILTER IOCTL enables an application to specify or modify a list of IPv4 source addresses on a given interface and to specify or modify a multicast group for a socket. The source filter can include or exclude the set of source addresses, depending on the filter mode (MCAST_INCLUDE or MCAST_EXCLUDE), which is defined in the IP_MSFILTER structure of the BPXYIOCC macro. The application can join multiple source multicast groups on a single socket; it also can join the same group on multiple interfaces on the same socket. However, there is a maximum limit of 20 groups per single UDP socket and there is a maximum limit of 256 groups per single RAW socket.

SIOCSMSFILTER

An SIOCSMSFILTER IOCTL enables an application to specify or modify a list of IPv4 or IPv6 source addresses on a given interface index and to specify or modify a multicast group for a socket. The source filter can include or exclude the set of source address, depending on the filter mode (MCAST_INCLUDE or MCAST_EXCLUDE), which is defined in the GROUP_FILTER structure of the BPXYIOCC macro. The application can join multiple source multicast groups on a single socket; it also can join the same group on multiple interfaces on the same socket. However, there is a maximum limit of 20 groups per single UDP socket and there is a maximum limit of 256 groups per single RAW socket.

Part 3. Application program interfaces

The following application programming interfaces (API's) are described:

- [Chapter 10, “C Socket application programming interface,” on page 85](#)
- [Chapter 11, “X/Open Transport Interface ,” on page 173](#)
- [Chapter 12, “Macro application programming interface,” on page 193](#)
- [Chapter 13, “CALL instruction application programming interface,” on page 389](#)
- [Chapter 14, “REXX socket application programming interface,” on page 577](#)
- [Chapter 15, “Pascal application programming interface,” on page 687](#)

Chapter 10. C Socket application programming interface

Note: The TCP/IP C socket API is not being enhanced for IPv6. The use of the UNIX C socket library is encouraged for IPv4 application development and is required for IPv6 application development. For more information, see [z/OS C/C++ Runtime Library Reference](#).

This information describes the C IPv4 socket application program interface (API) provided with TCP/IP. Use the socket routines to interface with the TCP, UDP, and IP protocols. The socket routines allow you to communicate with other programs across networks. You can, for example, use socket routines when you write a client program that must communicate with a server program running on another computer.

Topics include:

- Compiler restrictions
- Compiling and linking C applications
- Compiler messages
- Program abends
- C socket implementation
- C socket header files
- C structures
- Error messages and return codes
- C socket calls
- Sample C socket programs

To use the C socket API, you must know C language programming. For more information about C language programming, see [z/OS XL C/C++ Programming Guide](#).

Compiler restrictions with C applications

This topic tells you how to move your application to the z/OS Communications Server system.

- When you need to recompile, use the compiler shipped with this product.
- All applications linked to the TCP/IP C sockets library must run on the LE run-time library shipped with z/OS Communications Server.
- To access system return values, you need only use include statement `errno.h` supplied with the compiler. To access network return values, you must add the following include statement:

```
#include <tcperrno.h>
```

- To print system errors only, use `perror()`, a procedure available from the C compiler run-time library. To print both system and network errors, use `tcperror()`, a procedure provided by IBM and included with z/OS Communications Server.

Note to CICS users

Do not use `tcperror()`. Add statement `#include <ezacichd.h>` and compile the statement as non-reentrant. For more information, see the information about C Language Programming in the [z/OS Communications Server: IP CICS Sockets Guide](#).

- If your C language statements contain information, such as sequence numbers, that are not part of the input for the C/C++ compiler, you must exclude that information during compilation. The C/C++ compiler provides several ways to do this, one of which is:

```
#pragma margins (1,72)
```

In this example, we are presuming you have sequence numbers in columns 73 through 80.

- By default, prototype C socket functions and their parameters for the current release are defined. If you need to access the TCP/IP V3R1 definitions, specify the following information during a compile:

```
#define_TCP31_PROTOS
```

- Use of C socket functions by routines that are a part of fetched modules or DLLs might not yield the desired results. Applications that use these C language features need to be designed so that only one copy of the API code is used within the execution environment. Also note that proper cleanup of the supporting data structures relies on the termination logic defined with the `atexit()` function and has all of the corresponding restrictions listed for it (see [z/OS C/C++ Runtime Library Reference](#) for details). Improper use will likely cause new copies of the associated data structures to be allocated in the application's address space each time the fetched module or DLL is loaded.

Compiling and linking C applications

There are several ways to compile, link-edit, and execute z/OS Communications Server C source program in MVS. To run a C source program under MVS batch using IBM supplied cataloged procedures, you must include data sets. This topic contains information about the data sets that you must include.

The following data set name is used as an example in the sample Job Control Language (JCL) statements.

USER.MYPROG.H

Contains user `#include` files.

C application compatibility considerations

Unless noted in [z/OS Communications Server: New Function Summary](#), an application program compiled and link edited on a release of z/OS Communications Server IP can be used on higher level releases. That is, the API is upward compatible.

Application programs that are compiled and link edited on a release of z/OS Communications Server IP cannot be used on older releases. That is, the API is not downward compatible.

C socket API: Non-reentrant modules

You must make additions to the compile step of your cataloged procedure to compile a non-reentrant module. The following lines describe these additions. Cataloged procedures are included in the IBM-supplied samples for your MVS system.

Note: Compile all C code source using the `def(MVS)` preprocessor symbol.

- Add the following line as the first `//SYSLIB DD` statement:

```
//SYSLIB DD DSN=SEZACMAC,DISP=SHR
```

- Add the following `//USERLIB DD` statement:

```
//USERLIB DD DSN=USER.MYPROG.H,DISP=SHR
```

The following lines describe the additions that you must make to the link-edit step of your cataloged procedure to link-edit a non-reentrant module.

- To link-edit programs that use C sockets library functions, add the following statement as the first `//SYSLIB DD` statement:

```
//SYSLIB DD DSN=SEZACMTX,DISP=SHR
```

Figure 46 on page 87 shows a sample JCL to be used when compiling non-reentrant modules. Modify the lines to conform to the naming conventions of your site:

```
//COMPIT JOB ,COMPILE,MSGLEVEL=(1,1)
//*****
//*
//* SAMPLE JCL THAT COMPILES A TEST PROGRAM AS NORENT *
//* USING THE C/C++ COMPILER C/MVS IN NON-OE ENVIRONMENT *
//* INPUT : USER71.TEST.SRC(&INFILE) *
//* OUTPUT : USER71.TEST.OBJ(&OUTFILE) *
//* *
//*****
//*
//CPPC PROC CREGSIZ='4M',
// INFILE=CTEST,
// OUTFILE=CTEST,
// CPARM1=NORENT,
// CPARM2=' LIS,S0,EXP,OPT,DEF(MVS) ',
// DCB80='(RECFM=FB,LRECL=80,BLKSIZE=3200)',
// DCB3200='(RECFM=FB,LRECL=3200,BLKSIZE=12800)',
// LIBPRFX1='CEEL.OSV2R7',
// LIBPRFX2='CEE.OSV2R7',
// COMPRFX='CBC.OSV2R7'
//
//*-----
//* COMPILE STEP:
//*-----
//COMPILE EXEC PGM=CCNDVR,PARM=('&CPARM1','&CPARM2'),
// REGION=&CREGSIZ
//STEPLIB DD DSN=&LIBPRFX1..SCEERUN,DISP=SHR
// DD DSN=&COMPRFX..SCBCCMP,DISP=SHR
//SYMSGS DD DUMMY,DSN=&COMPRFX..SCBC3MSG(EDCMSGE),DISP=SHR
//SYSIN DD DSN=USER71.TEST.SRC(&INFILE),DISP=SHR

// DD DSN=&LIBPRFX2..SCEEH.H,DISP=SHR
//SYSLIN DD DSN=USER16.TEST.OBJ(&OUTFILE),DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSCPRT DD SYSOUT=*
//SYSUT1 DD UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB80
//SYSUT5 DD UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT6 DD UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT7 DD UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT8 DD UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT9 DD UNIT=VIO,SPACE=(32000,(30,30)),
// DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SYSUT10 DD SYSOUT=*
//
// PEND
// EXEC PROC=CPPC
```

Figure 46. Sample JCL for compiling non-reentrant modules

Figure 47 on page 88 shows a sample JCL to be used when linking non-reentrant modules. Modify the lines to conform to the naming conventions of your site:

```
//LINKIT JOB ,LINK,MSGLEVEL=(1,1)
//*****
//*
//* SAMPLE JCL THAT LINKS A NON_REENTRANT TEST PROGRAM
//* USING THE C/C++ COMPILER C/MVS
//* INPUT LIBRARY: USER71.TEST.OBJ(&MEM)
//* OUTPUT LIBRARY: USER71.TEST.LMOD(&MEM)
//*
//*****
//EDCL PROC USER=USER71
//TCPIP EXEC PGM=IEWL,
// PARM=' ,MAP,RMODE(ANY),SIZE=(320K,64K) '
//SYSPRINT DD SYSOUT=*
//SYSUT1 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSLMOD DD DSN=&USER..TEST.LMOD(&MEM),DISP=SHR
//SYSLIN DD DSN=&USER..TEST.OBJ(&MEM),DISP=SHR

// DD DSN=CEE.OSV2R7.SCEELKED,DISP=SHR
// PEND
// EXEC EDCL,MEM=CTEST
```

Figure 47. Sample JCL for linking non-reentrant modules

Figure 48 on page 88 shows JCL to be used when running non-reentrant modules. Modify the lines to conform to the naming conventions of your site:

```
//RUNTST JOB ,RUN,MSGLEVEL=(1,1),CLASS=A,REGION=4096K
//*****
//*
//* SAMPLE JCL THAT RUNS A TEST PROGRAM, CTEST
//*
//*****
//S1 EXEC PGM=CTEST
//STEPLIB DD DSN=CEEL.OSV2R7.SCEERUN,DISP=SHR
// DD DSN=USER71.TEST.LMOD,DISP=SHR
//SYSPRINT DD SYSOUT=*
```

Figure 48. Sample JCL for running non-reentrant modules

Note: For more information about compiling and linking, see [z/OS XL C/C++ Compiler and Runtime Migration Guide for the Application Programmer](#).

C socket API: Reentrant modules

The following lines describe the additions that you must make to the compile step of your cataloged procedure to compile a reentrant module. Cataloged procedures are included in the IBM-supplied samples for your MVS system.

Note: Compile all C source code using the def(MVS) preprocessor symbol.

Be sure to use the RENT compiler option if your code is reentrant.

- Add the following line as the first //SYSLIB DD statement:

```
//SYSLIB DD DSN=SEZACMAC,DISP=SHR
```

- Add the following //USERLIB DD statement:

```
//USERLIB DD DSN=USER.MYPROG.H,DISP=SHR
```

The following lines describe the additions that you must make to the prelink-edit and link-edit steps of your cataloged procedure to create a reentrant module.

To prelink programs that use the C sockets library function, put the following statement first in the SYSLIB concatenation:

```
//SYSLIB DD DSN=SEZARNT1,DISP=SHR
```

Guideline: The system administrator should have followed the instructions for program reentrancy in the [z/OS XL C/C++ Programming Guide](#) topic that contains information related to restrictions for using MVS TCP/IP API with z/OS UNIX.

To link-edit programs that have the C sockets library function, the following statement must be first in the SYSLIB concatenation:

```
//SYSLIB DD DSN=SEZACMTX,DISP=SHR
```

Notes:

1. If Language Environment libraries are concatenated ahead of SEZACMTX, socket errors can occur because the link-edit uses the Language Environment z/OS UNIX socket library, not the TCP/IP library.
2. For more information about compiling and linking, see [z/OS XL C/C++ Compiler and Runtime Migration Guide for the Application Programmer](#).

Figure 49 on page 89 shows sample JCL to be used when compiling a test program with reentrancy. Modify the lines to conform to the naming conventions of your site:

```
//*****
//*
//*  SAMPLE JCL THAT COMPILES A TEST PROGRAM, CTEST, AS 'RENT',
//*  USING THE C/C++ COMPILER C/MVS
//*
//*****
//*
//CPPC PROC CREGSIZ='4M',
//  INFILE=CTEST,
//  CPARM1=RENT,
//  CPARM2=' LIS,S0,EXP,OPT,DEF(MVS),SHOWINC',
//  DCB80=' (RECFM=FB,LRECL=80,BLKSIZE=3200)',
//  DCB3200=' (RECFM=FB,LRECL=3200,BLKSIZE=12800)',
//  LIBPRFX1='CEEL.OSV2R7',
//  LIBPRFX2='CEE.OSV2R7',
//  COMPRFX='CBC.OSV2R7'
//*
//*-----
//*  COMPILE STEP:
//*-----
//COMPILE EXEC PGM=CCNDVR,PARM=(
//  '&CPARM1','&CPARM2'),REGION=&CREGSIZ
//STEPLIB DD DSN=&LIBPRFX1..SCEERUN,DISP=SHR
// DD DSN=&COMPRFX..SCBCCMP,DISP=SHR
//SYMSGS DD DUMMY,DSN=&COMPRFX..SCBC3MSG(EDCMSGE),DISP=SHR
//SYSIN DD DSN=USER71.TEST.SRC(&INFILE),DISP=SHR

// DD DSN=&LIBPRFX2..SCEEH.H,DISP=SHR
//SYSLIN DD DSN=USER71.TEST.RENTDS,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSPRT DD SYSOUT=*
//SYSUT1 DD UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB80
//SYSUT5 DD UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT6 DD UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT7 DD UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT8 DD UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT9 DD UNIT=VIO,SPACE=(32000,(30,30)),
// DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SYSUT10 DD SYSOUT=*
//*
// PEND
// EXEC PROC=CPPC
```

Figure 49. Sample JCL for compiling reentrant modules

Figure 50 on page 90 shows sample JCL to be used when prelinking and linking a reentrant program using the C socket library. Modify the lines to conform to the naming conventions of your site:

```

//*****
//*
//* PRE-LINK AND LINK FOR REENTRANCY WITH C/C++ COMPILER,
//* z/OS RUNTIME LIBRARY.
//*
//* NOTES:
//* - SPECIFY 'RENT' ON LINK STEP
//* - RENTDS WAS PREVIOUSLY COMPILED WITH 'RENT'
//* - THE MEMBER @@DC370$ CAN BE USED TO BRING IN ALL C SOCKET
//* MEMBERS. THIS IS EASIER THAN SPECIFYING ALL THE INDIVIDUAL
//* MEMBER INCLUDES.
//* - TCP.SEZARNT1 IS THE REENTRANT C SOCKET LIBRARY.
//* IT IS USED ON THE PRE-LINK STEP.
//* - TCP.SEZACMTX IS THE GENERIC SOCKET LIBRARY.
//* IT IS USED ON THE LINK STEP TO RESOLVE OTHER C SOCKET
//* MODULES THAT DO NOT EXIST IN TCP.SEZARNT1.
//* - THE PRE-LINK REQUIRES THE 'UPCASE' PARM SO THAT THE
//* OTHER MODULES FROM SEZACMTX (WHICH ARE KNOWN BY
//* THEIR UPPERCASE NAMES) CAN BE FOUND.
//*****
//*
//*-----
//* MODIFY THE FOLLOWING LINES TO CONFORM TO THE
//* NAMING CONVENTIONS AT YOUR SITE.
//*-----
//RENTTEST PROC MYHLQ='USER71.TEST',
// LIBPRFX1='CEE',
// LIBPRFX2='CEE'
//*-----
//* PRE-LINKEDIT STEP:
//*-----
//PLKED EXEC PGM=EDCPRLK,
// REGION=2048K,PARM='UPCASE'
//STEPLIB DD DSN=&LIBPRFX1..SCEERUN,DISP=SHR
//SYMSMSG DD DSN=&LIBPRFX2..SCEEMSGP(EDCPMSG),DISP=SHR
//SYSLIB DD DSN=TCP.SEZARNT1,DISP=SHR
//SYSMOD DD DSN=&&PLKSET,UNIT=SYSDA,DISP=(MOD,PASS),
// SPACE=(32000,(30,30)),
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//OBJLIB DD DSN=&MYHLQ..OBJ,DISP=SHR
//MYRENT DD DSN=&MYHLQ..RENTDS,DISP=SHR
//SYSOUT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//*
//*-----
//* LINKEDIT STEP:
//*-----
//LKED EXEC PGM=IEWL,COND=(4,LT,PLKED),
// REGION=2048K,PARM='RENT,AMODE=31,MAP'
//SYSLIB DD DSN=TCP.SEZACMTX,DISP=SHR
// DD DSN=&LIBPRFX2..SCEELKED,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSN=&*.PLKED.SYSMOD,DISP=(OLD,DELETE)
//SYSLMOD DD DSN=&MYHLQ..LMOD(CTESTRNT),DISP=SHR
//SYSUT1 DD UNIT=SYSDA,SPACE=(32000,(30,30))
// PEND
//S1 EXEC PROC=RENTTEST
//PLKED.SYSIN DD *
// INCLUDE MYRENT
// INCLUDE SYSLIB(@@DC370$)
//*

```

Figure 50. Sample JCL for prelinking and linking reentrant modules

Figure 51 on page 90 shows sample JCL to be used when running the reentrant program prelinked and linked in the previous JCL sample. Modify the lines to conform to the naming conventions of your site:

```

//RUNTST JOB ,RUN,MSGLEVEL=(1,1),CLASS=A,REGION=4096K
//*****
//*
//* SAMPLE JCL THAT RUNS A TEST PROGRAM, CTESTRNT
//*
//*****
//S1 EXEC PGM=CTESTRNT
//STEPLIB DD DSN=CEEL.OSV2R7.SCEERUN,DISP=SHR
// DD DSN=USER71.TEST.LMOD,DISP=SHR
//SYSPRINT DD SYSOUT=*

```

Figure 51. Sample JCL for running the reentrant program

C program compiler messages

z/OS Communications Server uses the C/C++/390 compiler. For C programs, migrating from AD/Cycle to the C/C++/390 compiler can pose a few minor problems. See [z/OS XL C/C++ Messages](#) or [z/OS XL C/C++ Compiler and Runtime Migration Guide for the Application Programmer](#) for more information.

C program abends

A C program might compile and link correctly, but at run-time it might abend or behave peculiarly. The following lists some reasons for unexpected behavior, and suggests some fixes.

Errno values

Code depends on specific errno values. This might be a problem, as errno values can change from release to release. Take the following actions:

1. Check for any error conditions.
2. Make sure your logic has a default section that can be used if the specific errno has changed or is no longer available.

Printing errno values: The `tcperror()` function converts errno values to strings, which you can then print using `printf()` or a similar command. This procedure is provided by IBM and included with z/OS Communications Server, and is similar to the `strerror()` function in the standard C library.

Return values

Code depends on a specific return value. Some RTL functions, such as `remove()`, specify that the return code be nonzero on failure. In earlier releases, checking for -1 was sufficient; with release V1R4, the correct check is for nonzero.

Unfortunately, there is no checklist of functions that might generate this problem. If you get an abend, work backwards from the failure and examine prior RTL function return-code checking.

Built-in RTL functions

If RTL functions were built-in during your compile, ensure that they perform the same way as the non-built-in functions from the RTL.

Functions that might have this problem include `abs`, `cds`, `cs`, `decabs`, `decchk`, `decfix`, `fabs`, `fortrc`, `memchr`, `memcpy`, `memcmp`, `memset`, `strcat`, `strchr`, `strcmp`, `strcpy`, `strlen`, `strncat`, `strncmp`, `strncpy`, `strrchr`, and `tsched`.

SCEERUN missing

Ensure that SCEERUN is the first library in STEPLIB encountered by your compile procedure.

Uninitialized storage

Check for uninitialized storage. Storage for automatic variables is guaranteed to be garbage.

C socket implementation

The IBM socket implementation differs from the Berkeley socket implementation. The following list summarizes the differences in the two methods:

- The IBM implementation does not support `AF_INET6` sockets.
- Under IBM implementation, you must make reference to the additional header file, `TCPERRNO.H`, if you want to refer to the networking errors other than those described in the compiler-supplied `ERRNO.H` file.
- Under IBM implementation, you must use the `tcperror()` routine to print the networking errno messages. `tcperror()` should be used only after socket calls, and `perror()` should be used only after C library calls.
- Under IBM implementation, you must include `MANIFEST.H` to remap the socket function long names to eight-character names.
- The IBM `ioctl()` call implementation might differ from the current Berkeley `ioctl()` call implementation. See [“ioctl\(\)” on page 129](#) for a description of the functions supported by the IBM implementation.

- The IBM `getsockopt()` and `setsockopt()` calls support only a subset of the options available. See [“getsockopt\(\)” on page 119](#) and [“setsockopt\(\)” on page 152](#) for details about the supported options.
- The IBM `fcntl()` call supports only a subset of the options available. See [“fcntl\(\)” on page 103](#) for details about the supported commands.
- The IBM implementation supports an increased maximum number (2000) of simultaneous sockets through the use of the `maxdesc()` call. (Only 1997 simultaneous sockets can be used, however.) The default maximum number of sockets is 47, any or all of which can be `AF_INET` sockets.

Keep the following information in mind while creating your C socket application:

- Compile all C source using the `def(MVS)` preprocessor symbol.
- During debugging, set `sock_do_teststor` (1) to *on* to validate all storage addresses. After debugging, use `sock_do_teststor` (0) set to *off*.
- If the TCPIP stack of your C application is restarted, your C application must also be restarted. Otherwise, its socket calls will fail with an error value of `EIBMINVTCPCONNECTION`.

C socket header files

To use the socket routines described in this topic, you must have the following header files available to your compiler. They can be found in the SEZACMAC data set.

- `bsdtime.h`
- `bsdtocms.h`
- `bsdtypes.h`
- `fcntl.h`
- `if.h`
- `in.h`
- `inet.h`
- `ioctl.h`
- `manifest.h`
- `netdb.h`
- `rtroute.h`
- `socket.h`
- `tcperrno.h`
- `types.h`
- `uio.h`

Note: The C socket header files have been enhanced to allow the user to specify the coded character set to be used. When including the header files in an application, the `bsdtypes.h` file must precede the `socket.h` file.

Manifest.h header file

Under IBM implementation, `MANIFEST.H` is used to remap socket function long names to eight-character names. To refer to the names, you must include the following statement as the first `#include` at the beginning of each program:

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


Prototyping

Under TCP/IP z/OS Communications Server, the prototyping of C socket functions and their parameters is the default. If you are migrating your applications, you can bypass the new prototyping by specifying `#define_TCP31_PROTOS` during a C compile.

C structures

The parameter lists for some C language socket calls include a pointer to a data structure defined by a C structure. [Table 6 on page 93](#) shows the C structures used, and the corresponding assembler language syntax.

Table 6. C structures in assembler language format

C structure	Assembler language equivalent
<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 timeval { long tv_sec; long tv_usec; };</pre>	<pre>TVSEC DS F TVUSEC DS F</pre>
<pre>struct linger { int l_onoff; int l_linger; };</pre>	<pre>ONOFF DS F LINGER 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 0CL16 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 ifconf { int ifc_len; union { caddr_t ifcu_buf; struct ifreq *ifcu_req; } ifc_ifcu; };</pre>	<pre>IFCLEN DS F IGNORED DS F</pre>

Table 6. 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>

C socket API error messages and return codes

For information about error messages, see [z/OS Communications Server: IP Messages Volume 1 \(EZA\)](#).

The most common return codes (ERRNOs) returned by TCP/IP are listed following each socket call.

For information about all return codes see [Appendix B, “Socket call error return codes,”](#) on page 743.

C socket calls

This topic lists the syntax, parameters, and other information appropriate to each C socket call supported by TCP/IP.

accept()

The `accept()` call is used by a server to accept a connection request from a client. The call accepts the first connection on its queue of pending connections. The `accept()` call creates a new socket descriptor with the same properties as `s` and returns it to the caller. If the queue has no pending connection requests, `accept()` blocks the caller unless `s` is in nonblocking mode. If no connection requests are queued and `s` is in nonblocking mode, `accept()` returns -1 and sets `errno` to `EWOULDBLOCK`. The new socket descriptor cannot be used to accept new connections. The original socket, `s`, remains available to accept additional connection requests.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>
int accept(int s, struct sockaddr *addr, int *addrlen)
```

Parameter

Description

s

The socket descriptor.

addr

The socket address of the connecting client that is filled by `accept()` before it returns. The format of `addr` is determined by the domain in which the client resides. `addr` is specified by `accept()` only when both `addr` and `addrlen` are nonzero values.

addrlen

Must initially point to an integer that contains the size in bytes of the storage pointed to by `addr`. If `addr` is NULL, then `addrlen` is ignored and can be NULL.

The `s` parameter is a stream socket descriptor created using the `socket()` call. It is usually bound to an address using the `bind()` call. The `listen()` call marks the socket as one that accepts connections and allocates a queue to hold pending connection requests. The `listen()` call allows the caller to place an upper boundary on the size of the queue.

The `addr` parameter points to a buffer into which the connection requester address is placed. The `addr` parameter is optional and can be set to NULL. If `addr` or `addrlen` is null or 0, `addr` is not specified.

The exact format of `addr` depends on the addressing domain from which the communication request

originated. For example, if the connection request originated in the AF_INET domain, *addr* points to a *sockaddr_in* structure as defined in the header file IN.H. The *addrlen* parameter is used only when *name* is not NULL. Before calling `accept()`, you must set the integer pointed to by *addrlen* to the size of the buffer, in bytes, pointed to by *addr*. If the buffer is not large enough to hold the address, only the *addrlen* number of bytes of the requester address is copied.

Note: This call is used only with SOCK_STREAM sockets. There is no way to screen requesters without calling `accept()`. The application cannot determine which system from which requesters connections will be accepted. However, the caller can choose to close a connection immediately after discovering the identity of the requester.

A socket can be checked for incoming connection requests using the `select()` call.

Return values

A nonnegative socket descriptor indicates success; the value -1 indicates an error. `Errno` identifies the specific error.

Errno

Description

EBADF

The *s* parameter is not a valid socket descriptor.

ENOBUFS

Indicates insufficient buffer space available to create the new socket.

EINVAL

The *s* parameter is not of type SOCK_STREAM.

EFAULT

Using *addr* and *addrlen* would result in an attempt to copy the address into a portion of the caller address space to which information cannot be written.

EWOULDBLOCK

The socket descriptor *s* is in nonblocking mode, and no connections are in the queue.

Example

Following are two examples of the `accept()` call. In the first, the caller wants to have the requester's address returned. In the second, the caller does not want the requester address returned's.

```
int clientsocket;
int s;
struct sockaddr clientaddress;
int addrlen;
int accept(int s, struct sockaddr *addr, int *addrlen);
/* socket(), bind(), and listen() have been called */
/* EXAMPLE 1: I want the address now */
addrlen = sizeof(clientaddress);
clientsocket = accept(s, &clientaddress, &addrlen)
/* EXAMPLE 2: I can get the address later using getpeername() */
addrlen = 0;
clientsocket = accept(s, (struct sockaddr *) 0, (int *) 0);
```

Related calls

`bind()`, `connect()`, `getpeername()`, `listen()`, `socket()`

bind()

The `bind()` call binds a unique local name to the socket using descriptors. After calling `socket()`, the descriptor does not have a name associated with it. However, it does belong to a particular addressing family, as specified when `socket()` is called. The exact format of a name depends on the addressing family. The `bind()` call also allows servers to specify the network interfaces from which they want to receive UDP packets and TCP connection requests.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>
int bind(int s, struct sockaddr *name, int namelen)
```

Parameter Description

s

Socket descriptor returned by a previous socket() call

name

Points to a *sockaddr* structure containing the name to be bound to s

namelen

Size of *name* in bytes, which specifies the length of the *sockaddr* structure.

The s parameter is a socket descriptor of any type created by calling socket().

The *name* parameter points to a buffer containing the name to be bound to s. The *namelen* parameter is the size, in bytes, of the buffer pointed to by *name*.

Related information

Socket descriptor created in the AF_INET domain

If the socket descriptor s was created in the AF_INET domain, then the format of the name buffer is expected to be *sockaddr_in*, as defined in the header file IN.H.

```
struct in_addr
{
    u_long s_addr;
};
struct sockaddr_in
{
    short    sin_family;
    u_short  sin_port;
    struct   in_addr sin_addr;
    char     sin_zero[8];
};
```

The *sin_family* field must be set to AF_INET.

The *sin_port* field identifies the port to which the application must bind. It must be specified in network byte order. If *sin_port* is set to 0, the caller expects the system to assign an available port. The application can call getsockname() to discover the port number assigned.

The in_addr *sin_addr* field is set to the internet address and must be specified in network byte order. On hosts with more than one network interface (called multihomed hosts), a caller can select the interface to which it should bind. Subsequently, only UDP packets and TCP connection requests from this interface (the one value matching the bound name) are routed to the application. If this field is set to the constant INADDR_ANY, as defined in IN.H, the caller is requesting that the socket be bound to all network interfaces on the host. Subsequently, UDP packets and TCP connections from all interfaces matching the bound name are routed to the application. This becomes important when a server offers a service to multiple networks. By leaving the address unspecified, the server can accept all UDP packets and TCP connection requests made of its port, regardless of the network interface on which the requests arrived.

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

Socket descriptor created in the AF_IUCV domain

If the socket descriptor s is created in the AF_IUCV domain, the format of the name buffer is expected to be *sockaddr_iucv*, as defined in the header file SAIUCV.H.

```
struct sockaddr_iucv
{
    short          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_family* field must be 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 set to zero.
- The *siucv_nodeid* field must be set to exactly eight blank characters.
- The *siucv_userid* field is set to the MVS user ID of the application making the bind call. This field must be eight characters long, padded with blanks to the right. It cannot contain the NULL character.
- The *siucv_name* field is set to the application name by which the socket is to be known. It must be unique, because only one socket can be bound to a given name. The preferred form of the name contains eight characters, padded with blanks to the right. The eight characters for a connect() call executed by a client must exactly match the eight characters passed in the bind() call executed by the server.

Note: Internally, dynamic names are built using hexadecimal character strings representing the internal storage address of the socket. You should choose names that contain at least one non-hexadecimal character to prevent potential conflict. Hexadecimal characters include 0–9, and a–f. Uppercase A–F are considered non-hexadecimal and can be used by the user to build dynamic names.

Return values

The value 0 indicates success; the value -1 indicates an error. Errno identifies the specific error.

Errno

Description

EADDRINUSE

The address is already in use. See the SO_REUSEADDR option described under “[getsockopt\(\)](#)” on [page 119](#) and the SO_REUSEADDR option described under the “[setsockopt\(\)](#)” on [page 152](#) for more information. This Errno might also be returned if the port is configured as RESERVED on a port reservation statement in the TCP/IP profile. For details, see [TCP/IP profile](#) information in the [z/OS Communications Server: IP Configuration Reference](#).

EADDRNOTAVAIL

The address specified is not valid on this host. For example, the internet address does not specify a valid network interface.

EAFNOSUPPORT

The address family is not supported (it is not AF_INET).

EBADF

The s parameter is not a valid socket descriptor.

EFAULT

The name or namelen parameter specified an address outside of the caller address space.

EINVAL

The socket is already bound to an address. For example, an attempt to bind a name to a socket that is in the connected state. This Errno might also be returned if an invalid parameter was passed on the bind() invocation. Check the passed parameter values and ensure that they are specified as described above.

Example

The following examples show the bind() call. The internet address and port must be in network byte order. To put the port into network byte order, the htons() utility routine is called to convert a short integer from host byte order to network byte order. The *address* field is set using another utility routine, inet_addr(), which takes a character string representing the dotted decimal address of an interface and returns the binary internet address representation in network byte order. Finally, it is a good idea to clear

the structure before using it to ensure that the name requested does not set any reserved fields. See [“connect\(\)” on page 99](#) for examples how a client might connect to servers.

This example illustrates the `bind()` call binding to interfaces in the `AF_INET` domain.

```
int rc;
int s;
struct sockaddr_in myname;
struct sockaddr_iucv mymvsname;
int bind(int s, struct sockaddr *name, int namelen);
/* Bind to a specific interface in the internet domain */
/* make sure the sin_zero field is cleared */
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr = inet_addr("129.5.24.1"); /* specific interface */
myname.sin_port = htons(1024);
:
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
/* Bind to all network interfaces in the internet domain */
/* make sure the sin_zero field is cleared */
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr.s_addr = INADDR_ANY; /* specific interface */
myname.sin_port = htons(1024);
:
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
/* Bind to a specific interface in the internet domain.
   Let the system choose a port */
/* make sure the sin_zero field is cleared */
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr = inet_addr("129.5.24.1"); /* specific interface */
myname.sin_port = 0;
:
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
```

This example illustrates the `bind()` call binding to interfaces in the `AF_IUCV` domain.

```
/* Bind to a name in the IUCV domain */
/* make sure the siucv_addr, siucv_port fields are zeroed and the
   siucv_nodeid fields is set to blanks */
memset(&mymvsname, 0, sizeof(mymvsname));
strncpy(mymvsname.siucv_nodeid, " ", 8);
strncpy(mymvsname.siucv_userid, " ", 8);
strncpy(mymvsname.siucv_name, " ", 8);
mymvsname.siucv_family = AF_IUCV;
strncpy(mymvsname.siucv_userid, "MVSUSER1", 8);
strncpy(mymvsname.siucv_name, "APPL", 4);
:
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
```

The binding of a stream socket is not complete until a successful call to `bind()`, `listen()`, or `connect()` is made. Applications using stream sockets should check the return values of `bind()`, `listen()`, and `connect()` before using any function that requires a bound stream socket.

Related calls

`gethostbyname()`, `getsockname()`, `htons()`, `inet_addr()`, `listen()`, `socket()`

close()

The `close()` call shuts down the socket associated with the socket descriptor `s` and frees resources allocated to the socket. If `s` refers to an open TCP connection, the connection is closed. If a stream socket is closed when there is input data queued, the TCP connection is reset, not cleanly closed.

If you specify 0 on `SO_LINGER` on the `setsockopt()` call, the data is canceled and the `CLOSE` is immediately returned. If you do not specify a value for `SO_LINGER` on the `setsockopt()` call, the `CLOSE` returns and TCP/IP tries to immediately resend the data.

Note: Issue a `shutdown()` call before issuing a `close()` call for any socket.

```
#include <manifest.h>
#include <socket.h>
int close(int s)
```

Parameter

Description

s

Descriptor of the socket to be closed

Return values

The value 0 indicates success; the value -1 indicates an error. Errno identifies the specific error.

Errno

Description

EBADF

The `s` parameter is not a valid socket descriptor.

Related calls

`accept()`, `getsockopt()`, `setsockopt()`, `socket()`

connect()

For stream sockets, the `connect()` call attempts to establish a connection between two sockets. For UDP sockets, the `connect()` call specifies the peer for a socket. The `s` parameter is the socket used to originate the connection request. The `connect()` call performs two tasks when called for a stream socket. First, it completes the binding necessary for a stream socket [in case it has not been previously bound using the `bind()` call]. Second, it attempts to connect to another socket.

The `connect()` call on a stream socket is used by the client application to connect to a server. The server must have a passive open pending. If the server is using sockets, this means the server must successfully call `bind()` and `listen()` before a connection can be accepted by the server using `accept()`. Otherwise, `connect()` returns -1 and `errno` is set to `ECONNREFUSED`.

If `s` is in blocking mode, the `connect()` call blocks the caller until the connection is set up, or until an error is received. If the socket is in nonblocking mode, then `connect()` returns -1 with `errno` set to `EINPROGRESS` if the connection can be initiated (no other errors occurred). The caller can test completion of the connection setup by calling `select()` and testing ability to write to the socket.

When called for a datagram or raw socket, `connect()` specifies the peer with which this socket is associated. This gives the application the ability to use data transfer calls reserved for sockets that are in the connected state. In this case, `read()`, `write()`, `readv()`, `writev()`, `send()`, and `recv()` calls are available in addition to `sendto()`, `recvfrom()`, `sendmsg()`, and `recvmsg()` calls. Stream sockets can call `connect()` only once, but datagram sockets can call `connect()` multiple times to change their association. Datagram sockets can dissolve their association by connecting to an incorrect address, such as a null address (all fields cleared).

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>
int connect(int s, struct sockaddr *name, int namelen)
```

Parameter

Description

s

Socket descriptor

name

Points to a *socket address* structure containing the address of the socket to which connection will be attempted

namelen

Size of the *socket address*, in bytes, pointed to by *name*

The *name* parameter points to a buffer containing the name of the peer to which the application needs to connect. The *namelen* parameter is the size, in bytes, of the buffer pointed to by *name*.

Related information**Servers in the AF_INET domain**

If the server is in the AF_INET domain, the format of the name buffer is expected to be *sockaddr_in* as defined in the header file IN.H.

```
struct in_addr
{
    u_long s_addr;
};
struct sockaddr_in
{
    short    sin_family;
    u_short  sin_port;
    struct   in_addr sin_addr;
    char     sin_zero[8];
};
```

The *sin_family* field must be set to AF_INET. The *sin_port* field identifies the port to which the server is bound; it must be specified in network byte order. The *sin_addr* field specifies a 32-bit Internet address. The *sin_zero* field is not used, and must be set to all zeros.

Servers in the AF_IUCV domain

If the server is in the AF_IUCV domain, the format of the name buffer is expected to be *sockaddr_iucv* as defined in the header file SAIUCV.H.

```
struct sockaddr_iucv
{
    short          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_family* field must be set to AF_IUCV.

Note: The *siucv_port*, *siucv_addr*, and *siucv_nodeid* fields are reserved for future use.

The *siucv_port* and *siucv_addr* fields must be set to 0. Set the *siucv_nodeid* field to exactly eight blank characters. The *siucv_userid* field is set to the MVS user ID of the application to which the application is requesting a connection. This field must be eight characters long, padded with blanks to the right. It cannot contain the null character. The *siucv_name* field is set to the application name by which the server socket is known. The name should exactly match the eight characters passed in the bind() call executed by the server.

Return values

The value 0 indicates success; the value -1 indicates an error. Errno identifies the specific error.

Errno**Description**

EADDRINUSE

The address is already in use. See the `SO_REUSEADDR` option described under “`getsockopt()` ” on page 119 and the `SO_REUSEADDR` option described under the “`setsockopt()` ” on page 152 for more information. This `Errno` might also be returned if the port is configured as `RESERVED` on a port reservation statement in the TCP/IP profile. For details, see [TCP/IP profile information in the z/OS Communications Server: IP Configuration Reference](#).

EADDRNOTAVAIL

Calling host cannot reach the specified destination.

EAFNOSUPPORT

Address family is not supported.

EALREADY

Socket descriptor `s` is marked nonblocking, and a previous connection attempt is incomplete.

EBADF

The `s` parameter is not a valid socket descriptor.

ECONNREFUSED

The connection request was rejected by the destination host.

EFAULT

The `name` or `namelen` parameter specified an address outside of the caller address space.

EINPROGRESS

The socket descriptor `s` is marked nonblocking, and the connection cannot be completed immediately. The `EINPROGRESS` value does not indicate an error.

EISCONN

Socket descriptor `s` is already connected.

ENETUNREACH

Network cannot be reached from this host.

ETIMEDOUT

Connection attempt timed out before the connection was made.

Example

Following is a `connect()` call example. The internet address and port must be in network byte order. To put the port into network byte order, the `htons()` utility is called to convert a short integer from host byte order to network byte order. The `address` field is set using another utility, `inet_addr()`, which takes a character string representing the dotted decimal address of an interface and returns the binary internet address in network byte order. Set the structure to 0 before using it to ensure that the name requested does not set any reserved fields.

These examples could be used to connect to the servers shown in the examples listed with the call “`bind()` ” on page 95.

```
int s;
struct sockaddr_in servername;
struct sockaddr_iucv mvsservername;
int rc;
int connect(int s, struct sockaddr *name, int namelen);
/* Connect to server bound to a specific interface in the internet domain */
/* make sure the sin_zero field is cleared */
memset(&servername, 0, sizeof(servername));
servername.sin_family = AF_INET;
servername.sin_addr = inet_addr("129.5.24.1"); /* specific interface */
servername.sin_port = htons(1024);
:
rc = connect(s, (struct sockaddr *) &servername, sizeof(servername));
/* Connect to a server bound to a name in the IUCV domain */
/* make sure the siucv_addr, siucv_port, siucv_nodeid fields are cleared */
memset(&mvsservername, 0, sizeof(mvsservername));
mvsservername.siucv_family = AF_IUCV;
strncpy(mvsservername.siucv_nodeid, " ", 8);
/* The field is 8 positions padded to the right with blanks */
strncpy(mvsservername.siucv_userid, "MVSUSER1 ", 8);
```

```
stncpy(mvsservername.siucv_name, "APPL", 8);
:
rc = connect(s, (struct sockaddr *) &mvsservername, sizeof(mvsservername));
```

Related calls

bind(), htons(), inet_addr(), listen(), select(), selectex(), socket()

endhostent()

When indicated by sethostent(), the endhostent() call frees the cached information for the local host tables. The endhostent() call is available only where RESOLVE_VIA_LOOKUP is defined before MANIFEST.H is included. See [z/OS Communications Server: IP Configuration Guide](#) for information about using local host tables.

```
#include <manifest.h>
#include <socket.h>
void endhostent()
```

Parameters

None

Related calls

gethostbyname(), gethostent(), sethostent()

endnetent()

When indicated by setnetent(), the endnetent() call frees the cached information for the local host tables. The endnetent() call is available only where RESOLVE_VIA_LOOKUP is defined before MANIFEST.H is included. See [z/OS Communications Server: IP Configuration Guide](#) for information about using local host tables.

```
#include <manifest.h>
#include <socket.h>
void endnetent()
```

Parameters

None

Related calls

getnetbyname(), getnetent(), setnetent()

endprotoent()

The endprotoent() call closes the *hlq.ETC.PROTO* data set.

The *hlq.ETC.PROTO* data set contains information about networking protocols IP, ICMP, TCP, and UDP.

```
#include <manifest.h>
#include <socket.h>
void endprotoent()
```

Parameters

None

Related calls

getprotoent(), setprotoent()

endservent()

The endservent() call closes the *h/q.ETC.SERVICES* data set.

The *h/q.ETC.SERVICES* data set contains information about the networking services running on the host. Example services are domain name server, FTP, and Telnet.

```
#include <manifest.h>
#include <socket.h>
void endservent()
```

Parameters

None

Related calls

getservbyport(), getservent(), setservent()

fcntl()

The operating characteristics of sockets can be controlled with the fcntl() call.

Note: COMMAND values that are supported by the UNIX Systems Services fcntl() callable service are also supported.

```
#include <manifest.h>
#include <socket.h>
#include <bsdtypes.h>
#include <fcntl.h>
int fcntl(int s, int cmd, int arg)
```

Parameter

Description

s

Socket descriptor

cmd

Command to perform

arg

Data associated with *cmd*

The operations to be controlled are determined by *cmd*. The *arg* parameter is a variable, the meaning of which depends on the value of the *cmd* parameter. The following commands are valid fcntl() commands:

Command

Description

F_SETFL

Sets the status flags of socket descriptor *s*. (One flag, FNDELAY, can be set.)

F_GETFL

Returns the status flags of socket descriptor *s*. (One flag, FNDELAY, can be queried.)

The FNDELAY flag marks *s* as being in nonblocking mode. If data is not present on calls that can block [read(), readv(), and recv()] the call returns with -1, and errno is set to EWOULDBLOCK.

Note: This function does not reject other values that might be rejected downstream.

Return values

For the **F_GETFL** command, the return value is the flags, set as a bit mask. For the **F_SETFL** command, the value 0 indicates success; the value -1 indicates an error. Errno identifies the specific error.

Errno

Description

EBADF

The *s* parameter is not a valid socket descriptor.

EINVAL

The *arg* parameter is not a valid flag, or the command is not a valid command.

Example

```
int s;  
int rc;  
int flags;  
:  
/* Place the socket into nonblocking mode */  
rc = fcntl(s, F_SETFL, FNDELAY);  
/* See if asynchronous notification is set */  
flags = fcntl(s, F_GETFL, 0);  
if (flags & FNDELAY)  
    /* it is set */  
else  
    /* it is not */
```

Related calls

ioctl(), getsockopt(), setsockopt(), socket()

getclientid()

The getclientid() call returns the identifier by which the calling application is known to the TCP/IP address space. The *clientid* is used in givesocket() and takesocket() calls.

```
#include <manifest.h>  
#include <socket.h>  
#include <bsdtypes.h>  
int getclientid(int domain, struct clientid *clientid)
```

Parameter

Description

domain

The value in *domain* must be AF_INET.

clientid

Points to a *clientid* structure to be provided.

Return values

The value 0 indicates success. The value -1 indicates an error. Errno identifies the specific error.

Errno

Description

EFAULT

The *clientid* parameter as specified would result in an attempt to access storage outside the caller address space, or storage that cannot be modified by the caller.

EAFNOSUPPORT

The domain is not AF_INET.

Related calls

takesocket()

getdtablesize()

The TCP/IP address space reserves a fixed-size table for each address space using sockets. The size of this table equals the number of sockets an address space can allocate simultaneously. The `getdtablesize()` call returns the maximum number of sockets that can fit in the table.

To increase the table size, use `maxdesc()`. After calling `maxdesc()`, always use `getdtablesize()` to verify the change.

```
#include <manifest.h>
#include <socket.h>
int getdtablesize()
```

Parameters

None

Related calls

`maxdesc()`

gethostbyaddr()

The `gethostbyaddr()` call tries to resolve the IP address to a host name. The resolution attempted depends on how the resolver is configured and if any local host tables exist. If the symbol `RESOLVE_VIA_LOOKUP` is defined before including `MANIFEST.H`, `gethostbyaddr()` only uses local host tables and name servers are not used. See [z/OS Communications Server: IP Configuration Guide](#) for information about configuring the resolver and using local host tables.

```
#include <manifest.h>
#include <netdb.h>
struct hostent *gethostbyaddr(char *addr, int addrlen, int domain)
```

Parameter

Description

addr

Points to an unsigned long value containing the address of the host

addrlen

Size of *addr* in bytes

domain

Address domain supported (AF_INET)

The `gethostbyaddr()` call points to *hostent* structure for the host address specified on the call.

The `NETDB.H` header file defines the *hostent* structure, and contains the following elements:

Element**Description*****h_name***

The address of the official name of the host

h_aliases

A pointer to a zero-terminated list of addresses pointing to alternate names for the host

h_addrtype

The type of host address returned; currently, always set to AF_INET

h_length

The length of the host address, in bytes

h_addr

A pointer to a zero-terminated list of addresses pointing to the internet host addresses returned by the call

Return values

The return value points to static data that is overwritten by subsequent calls. A pointer to a *hostent* structure indicates success. A NULL pointer indicates an error or end-of-file. When a NULL pointer or 0 is returned, the value of `errno` is indeterminate: the output from a `tcperror()` call cannot be validated. The global variable `h_errno` identifies the specific error.

h_errno**Description****HOST_NOT_FOUND**

The name specified is unknown, the address domain specified is not supported, or the address length specified is not valid.

TRY_AGAIN

Temporary error; information not currently accessible.

NO_RECOVERY

Fatal error occurred.

Related calls

`gethostent()`, `sethostent()`, `endhostent()`

gethostbyname()

The `gethostbyname()` call tries to resolve the host address to an IP address. The resolution attempted depends on how the resolver is configured and if any local host tables exist. If the symbol `RESOLVE_VIA_LOOKUP` is defined before including `MANIFEST.H`, `gethostbyname()` only uses local host tables and name servers are not used. See [z/OS Communications Server: IP Configuration Guide](#) for information about configuring the resolver and using local host tables.

```
#include <manifest.h>
#include <netdb.h>
struct hostent *gethostbyname(char *name)
```

Parameter**Description*****name***

The name of the host being queried. Any trailing blanks will be removed from the specified name prior to trying to resolve it to an IP address. The maximum host name length is 255 characters.

The `gethostbyname()` call returns a pointer to a *hostent* structure for the host name specified on the call.

The `NETDB.H` header file defines the *hostent* structure and contains the following elements:

Element
Description

h_name

The address of the official name of the host

h_aliases

A pointer to a zero-terminated list of addresses pointing to alternate names for the host

h_addrtype

The type of host address returned; currently, set to AF_INET

h_length

The length of the host address in bytes

h_addr

A pointer to the network address of the host

Return values

The return value points to static data that is overwritten by subsequent calls. A pointer to a *hostent* structure indicates success. A NULL pointer indicates an error or EOF. When a NULL pointer or 0 is returned, the value of `errno` is indeterminate, and, therefore, the output from a `tcperror()` call is also not valid. Global variable `h_errno` identifies the specific error.

h_errno Value
Description

HOST_NOT_FOUND

The name specified is unknown.

TRY_AGAIN

Temporary error; information not currently accessible.

NO_RECOVERY

Fatal error occurred.

Related calls

`gethostent()`, `sethostent()`, `endhostent()`

gethostent()

The `gethostent()` call returns the next line in the local host table for a host name and points to the next host entry in the local host table. The `gethostent()` call also returns any aliases. The `gethostent()` call is available only when `RESOLVE_VIA_LOOKUP` is defined before `MANIFEST.H` is included. See [z/OS Communications Server: IP Configuration Guide](#) for information about using local host tables.

```
#include <manifest.h>
#include <netdb.h>
struct hostent *gethostent()
```

The `NETDB.H` header file defines the *hostent* structure and contains the following elements:

Element
Description

h_name

The address of the official name of the host

h_aliases

A pointer to a zero-terminated list of addresses pointing to alternate names for the host

h_addrtype

The type of host address returned; currently set to AF_INET

h_length

The length of the host address, in bytes

h_addr

A pointer to the network address of the host

Return values

The return value points to static data that is overwritten by subsequent calls. A pointer to a *hostent* structure indicates success. A NULL pointer indicates an error or EOF. When a NULL pointer or 0 is returned, the value of `errno` is indeterminate, and, therefore, the output from a `tcperror()` call is also not valid.

Related calls

`gethostbyname()`, `sethostent()`

gethostid()

The `gethostid()` call returns the 32-bit identifier unique to the current host. This value is the default home internet address.

This call can be used only in the `AF_INET` domain.

```
#include <manifest.h>
#include <socket.h>
unsigned long gethostid()
```

Return values

The `gethostid()` call returns the 32-bit identifier of the current host, which should be unique across all hosts. When a NULL pointer or 0 is returned, the value of `errno` is indeterminate, and, therefore, the output from a `tcperror()` call is also not valid.

Related calls

`gethostname()`

gethostname()

The `gethostname()` call returns the name of the host processor on which the program is running. Characters to the limit of *namelen* are copied into the name array. The value returned for host name is limited to 24 characters. The returned name is NULL-terminated unless truncated to the size of the name array.

Note: The host name returned is the host name the TCP/IP stack learned at startup from the `TCP/IP.DATA` file that was found.

This call can be used only in the `AF_INET` domain.

`Errno EINVAL` is returned when *namelen* is 0, or greater than 255 characters.

```
#include <manifest.h>
#include <socket.h>
int gethostname(char *name, int namelen)
```

Parameter	Description
-----------	-------------

name

Character array to be filled with the host name

namelen

Length of *name*; restricted to 255 characters

Return values

The value 0 indicates success; the value -1 indicates an error. Errno identifies the specific error.

Errno**Description****EFAULT**

The *name* parameter specified an address outside the caller address space.

Related calls

gethostbyname(), gethostid()

getibmopt()

The getibmopt() call returns the number of TCP/IP images installed on a given MVS system, and their status, version, and name.

Note: Images from pre-V3R2 releases of TCP/IP for MVS are excluded. The getibmopt() call is not meaningful to pre-V3R2 releases.

Using this information, the caller can dynamically choose the TCP/IP image with which to connect through the setibmopt() call. The getibmopt() call is optional. If it is not used, determine the connecting TCP/IP image as follows:

- Connect to the TCP/IP specified TCPIPJOBNAME in the active TCPIP.DATA file.
- Locate TCPIP.DATA using the search order described in the [z/OS Communications Server: IP Configuration Reference](#).

```
#include <manifest.h>
#include <socket.h>
int getibmopt(int cmd, struct ibm_gettcpinfo *buf)
struct ibm_tcpimage {
    unsigned short status;
    unsigned short version;
    char name[8];
}
struct ibm_gettcpinfo {
    int tcpcnt;
    struct ibm_tcpimage image[8];
}
```

Parameter**Description****cmd**

The command to perform. For TCP/IP V3R2 for MVS, IBMTCP_IMAGE is supported.

buf

Points to the structure that the call specifies.

The *buf* parameter is a pointer to the (*struct ibm_gettcpinfo*) buffer into which the TCP/IP image status, version, and name are placed.

On successful return, the *struct ibm_tcpimage* buffer contains the status, version, and name of up to eight active TCP/IP images.

The status field can contain the following information:

Status Field	Meaning
--------------	---------

X'8xxx'	Active
---------	--------

X'4xxx'	Terminating
---------	-------------

X'2xxx'	Down
---------	------

X'1xxx'	Stopped or stopping
---------	---------------------

Note: In the above status fields, xxx is reserved for IBM use and can contain any value.

When this field returns with a combination of Down and Stopped, TCP/IP was abended. Value stopped, when returned alone, indicates that TCP/IP has been stopped only.

The version field for z/OS V1R7 is X'0617'.

The TCP/IP character name field is the PROC name, left-justified, and padded with blanks.

The *tcpcnt* field of *struct ibm_gettcpinfo* is a count field into which the TCP/IP image count is placed. The caller uses this value to determine how many entries in the *ibm_tcpimage* structure of *buf* have been filled. If the *tcpcnt* returned is 0, there are no TCP/IP images present.

Return values

Zero indicates success; the value -1 indicates an error. Errno identifies the specific error.

Errno

Description

EOPNOTSUPP

This is returned for command that is not valid.

EFAULT

The *name* parameter specified an address outside of the caller address space.

getibmssockopt()

Like `getsockopt()` call, the `getibmssockopt()` call gets the options associated with a socket in the `AF_INET` domain. This call is for options specific to the IBM implementation of sockets. Currently, only the `SOL_SOCKET` level is supported.

This call can be used only in the `AF_INET` domain.

```
#include <manifest.h>
#include <socket.h>
int getibmssockopt(int s, int level, int optname, char *optval, int *optlen)
```

Parameter

Description

s

The socket descriptor

level

The level for which the option is set

optname

The name of a specified socket option

optval

Points to option data

optlen

Points to the length of the option data

Return values

The value 0 indicates success; the value -1 indicates an error. Errno identifies the specific error.

Errno**Description****EBADF**

The *s* parameter is not a valid socket descriptor.

EFAULT

Using *optval* and *optlen* parameters would result in an attempt to access storage outside the caller address space.

EINVAL

This is returned when *optlen* points to a length of 0.

Example

```
#include <manifest.h>
#include <socket.h>
#include <tcperror.h>
{ struct ibm_bulkmode_struct bulkstr;
  int optlen, rc;
  optlen = sizeof(bulkstr);
  rc = getibmssockopt(s, SOL_SOCKET, (char *), &bulkstr, &optlen);
  if (rc < 0)
  { tcperror("on getibmssockopt()");
    exit(1);
  }
  fprintf(stream, "%d byte buffer available for outbound queue.\n",
          bulkstr.b_max_send_queue_size_avail);
}
```

Related calls

ibmsflush(), setibmssockopt(), getsockopt()

getnetbyaddr()

The `getnetbyaddr()` call searches the local host tables for the specified network address. This call can be used only in the AF_INET domain. See [z/OS Communications Server: IP Configuration Guide](#) for information about using local host tables.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <netdb.h>
struct netent *getnetbyaddr(unsigned long net, int type)
```

Parameter**Description****net**

The network address

type

The address domain supported (AF_INET)

The *netent* structure is defined in the NETDB.H header file and contains the following elements:

Element	Description
---------	-------------

<i>n_name</i>	The official name of the network.
----------------------	-----------------------------------

<i>n_aliases</i>	An array, terminated with a NULL pointer, of alternative names for the network.
-------------------------	---

<i>n_addrtype</i>	The type of network address returned. The call always sets this value to AF_INET.
--------------------------	---

<i>n_net</i>	The network number, returned in host byte order.
---------------------	--

Return values

The return value points to static data that is overwritten by subsequent calls. A pointer to a *netent* structure indicates success. A NULL pointer indicates an error or end-of-file. When a NULL pointer or 0 is returned, the value of `errno` is indeterminate, and therefore, the output from a `tcperror()` call cannot be validated.

Related calls

`endnetent()`, `getnetbyname()`, `getnetent()`, `setnetent()`, `endhostent()`

getnetbyname()

The `getnetbyname()` call searches the local host tables for the specified network name. This call can be used only in the AF_INET domain. See [z/OS Communications Server: IP Configuration Guide](#) for information about using local host tables.

```
#include <manifest.h>
#include <netdb.h>
struct netent *getnetbyname(char *name)
```

Parameter	Description
-----------	-------------

<i>name</i>	Points to a network name.
--------------------	---------------------------

The `getnetbyname()` call returns a pointer to a *netent* structure for the network name specified on the call.

The *netent* structure is defined in the NETDB.H header file and contains the following elements:

Element	Description
---------	-------------

<i>n_name</i>	The official name of the network.
----------------------	-----------------------------------

<i>n_aliases</i>	An array, terminated with a NULL pointer, of alternative names for the network.
-------------------------	---

<i>n_addrtype</i>	The type of network address returned. The call always sets this value to AF_INET.
--------------------------	---

<i>n_net</i>	The network number, returned in host byte order.
---------------------	--

Return values

The return value points to static data that is overwritten by subsequent calls. A pointer to a *netent* structure indicates success. A NULL pointer indicates an error or end-of-file. When a NULL pointer or 0 is returned, the value of `errno` is indeterminate, and therefore, the output from a `tcperror()` call cannot be validated.

Related calls

`endnetent()`, `getnetbyaddr()`, `getnetent()`, `setnetent()`, `endhostent()`

getnetent()

The `getnetent()` call returns the next line in the local host table for a network name and points to the next network entry in the local host table. The `getnetent()` call also returns any aliases. The `getnetent()` call is available only when `RESOLVE_VIA_LOOKUP` is defined before `MANIFEST.H` is included. See [z/OS Communications Server: IP Configuration Guide](#) for information about using local host tables.

```
#include <manifest.h>
#include <netdb.h>
struct netent *getnetent()
```

The *netent* structure is defined in the `NETDB.H` header file and contains the following elements:

Element

Description

n_name

The official name of the network.

n_aliases

An array, terminated with a NULL pointer, of alternative names for the network.

n_addrtype

The type of network address returned. The call always sets this value to `AF_INET`.

n_net

The network number, returned in host byte order.

Return values

The return value points to static data that is overwritten by subsequent calls. A pointer to a *netent* structure indicates success. A NULL pointer indicates an error or end-of-file.

Related calls

`endnetent()`, `getnetbyaddr()`, `getnetbyname()`, `setnetent()`, `endhostent()`

getpeername()

The `getpeername()` call returns the name of the peer connected to socket descriptor *s*. For `AF_IUCV`, *namelen* must be initialized to reflect the size of the space pointed to by *name*; it is set to the number of bytes copied into the space before the call returns. For `AF_INET`, the input value in the contents of *namelen* is ignored, but is set before the call returns. The size of the peer name is returned in bytes. If the buffer of the local host is too small to receive the entire peer name, the name is truncated.

```
#include <manifest.h>
#include <bsdtypes.h>
```

```
#include <socket.h>
int getpeername(int s, struct sockaddr *name, int *namelen)
```

Parameter Description

s

The socket descriptor.

name

Points to a structure containing the internet address of the connected socket that is specified by `getpeername()` before it returns. The exact format of *name* is determined by the domain in which communication occurs.

namelen

Points to a fullword containing the size of the address structure pointed to by *name* in bytes.

Return values

The value 0 indicates success; the value -1 indicates an error. Errno identifies the specific error.

Errno Description

EBADF

The s parameter is not a valid socket descriptor.

EFAULT

Using the *name* and *namelen* parameters as specified would result in an attempt to access storage outside of the caller address space.

ENOTCONN

The socket is not in the connected state.

Related calls

`accept()`, `connect()`, `getsockname()`, `socket()`

getprotobyname()

The `getprotobyname()` call searches the *hlq.ETC.PROTO* data set for the specified protocol name.

The `getprotobyname()` call returns a pointer to a *protoent* structure for the network protocol specified on the call.

```
#include <manifest.h>
#include <netdb.h>
struct protoent *getprotobyname(char *name)
```

Parameter Description

name

Points to the specified protocol.

The *protoent* structure is defined in the NETDB.H header file and contains the following elements:

Element Description

p_name

The official name of the protocol

p_aliases

An array, terminated with a NULL pointer, of alternative names for the protocol

p_proto

The protocol number

Return values

The return value points to static data that is overwritten by subsequent calls. A pointer to a *protoent* structure indicates success. A NULL pointer indicates an error or end-of-file. When a NULL pointer or 0 is returned, the value of `errno` is indeterminate, and therefore, the output from a `tcperror()` call cannot be validated.

Related calls

`endprotoent()`, `getprotobynumber()`, `getprotoent()`, `setprotoent()`

getprotobynumber()

The `getprotobynumber()` call searches the *hlq*.ETC.PROTO data set for the specified protocol number.

The `getprotobynumber()` call returns a pointer to a *protoent* structure for the network protocol specified on the call.

```
#include <manifest.h>
#include <netdb.h>
struct protoent *getprotobynumber(int proto)
```

Parameter**Description*****proto***

Protocol number

The *protoent* structure is defined in the NETDB.H header file and contains the following elements:

Element**Description*****p_name***

The official name of the protocol

p_aliases

An array, terminated with a NULL pointer, of alternative names for the protocol

p_proto

The protocol number

Return values

The return value points to static data that is overwritten by subsequent calls. A pointer to a *protoent* structure indicates success. A NULL pointer indicates an error or end-of-file. When a NULL pointer or 0 is returned, the value of `errno` is indeterminate, and therefore, the output from a `tcperror()` call cannot be validated.

Related calls

`endprotoent()`, `getprotbyname()`, `getprotoent()`, `setprotoent()`

getprotoent()

The `getprotoent()` call reads the `hlq.ETC.PROTO` data set, and the `getprotoent()` call returns a pointer to the next entry in the `hlq.ETC.PROTO` data set.

```
#include <manifest.h>
#include <netdb.h>
struct protoent *getprotoent()
```

The *protoent* structure is defined in the NETDB.H header file and contains the following elements:

Element	Description
---------	-------------

<i>p_name</i>	The official name of the protocol
----------------------	-----------------------------------

<i>p_aliases</i>	An array, terminated with a NULL pointer, of alternative names for the protocol
-------------------------	---

<i>p_proto</i>	The protocol number
-----------------------	---------------------

Return values

The return value points to static data that is overwritten by subsequent calls. A pointer to a *protoent* structure indicates success. A NULL pointer indicates an error or end-of-file. When a NULL pointer or 0 is returned, the value of `errno` is indeterminate, and therefore, the output from a `tcperror()` call cannot be validated.

Related calls

`endprotoent()`, `getprotobyname()`, `getprotobynumber()`, `setprotoent()`

getservbyname()

The `getservbyname()` call searches the `hlq.ETC.SERVICES` data set for the specified service name. Service name searches must match the protocol, if a protocol is stated.

The `getservbyname()` call returns a pointer to a *servent* structure for the network service specified on the call.

```
#include <manifest.h>
#include <netdb.h>
struct servent *getservbyname(char *name, char *proto)
```

Parameter	Description
-----------	-------------

<i>name</i>	Points to the specified service name
--------------------	--------------------------------------

<i>proto</i>	Points to the specified protocol
---------------------	----------------------------------

The *servent* structure is defined in the NETDB.H header file and contains the following elements:

Element	Description
---------	-------------

<i>s_name</i>	The official name of the service
----------------------	----------------------------------

<i>s_aliases</i>	An array, terminated with a NULL pointer, of alternative names for the service
-------------------------	--

<i>s_port</i>	The port number of the service
----------------------	--------------------------------

s_proto

The protocol required to contact the service

Return values

The return value points to static data that is overwritten by subsequent calls. Points to a *servent* structure indicates success. A NULL pointer indicates an error or end-of-file. When a NULL pointer or 0 is returned, the value of `errno` is indeterminate, and therefore, the output from a `tcperror()` call cannot be validated.

Related calls

`endservant()`, `getservbyport()`, `getservent()`, `setservent()`

getservbyport()

The `getservbyport()` call searches the *h/q.ETC.SERVICES* data set for the specified port number. Searches for a port number must match the protocol, if a protocol is stated.

The `getservbyport()` call returns a pointer to a *servent* structure for the port number specified on the call.

```
#include <manifest.h>
#include <netdb.h>
struct servent *getservbyport(int port, char *proto)
```

Parameter**Description*****port***

Port number

proto

Points to the specified protocol

The *servent* structure is defined in the `NETDB.H` header file and contains the following elements:

Element**Description*****s_name***

The official name of the service

s_aliases

An array, terminated with a NULL pointer, of alternative names for the service

s_port

The port number of the service

s_proto

The protocol required to contact the service

Return values

The return value points to static data that is overwritten by subsequent calls. Points to a *servent* structure indicates success. A NULL pointer indicates an error or end-of-file. When a NULL pointer or 0 is returned, the value of `errno` is indeterminate, and therefore, the output from a `tcperror()` call cannot be validated.

Related calls

`endservant()`, `getservbyname()`, `getservent()`, `setservent()`

getservent()

The `getservent()` call reads the next line of the *hlq.ETC.SERVICES* data set and returns a pointer to the next entry in the *hlq.ETC.SERVICES* data set.

```
#include <manifest.h>
#include <netdb.h>
struct servent *getservent()
```

The *servent* structure is defined in the NETDB.H header file and contains the following elements:

Element	Description
---------	-------------

<i>s_name</i>	The official name of the service
----------------------	----------------------------------

<i>s_aliases</i>	An array, terminated with a NULL pointer, of alternative names for the service
-------------------------	--

<i>s_port</i>	The port number of the service
----------------------	--------------------------------

<i>s_proto</i>	The required protocol to contact the service
-----------------------	--

Return values

The return value points to static data that is overwritten by subsequent calls. Points to a *servent* structure indicate success. A NULL pointer indicates an error or end-of-file. When a NULL pointer or 0 is returned, the value of `errno` is indeterminate, and therefore, the output from a `tcperror()` call cannot be validated.

Related calls

`endservant()`, `getservbyname()`, `getservbyport()`, `setservent()`

getsockname()

The `getsockname()` call stores the current name for the socket specified by the *s* parameter into the structure pointed to by the *name* parameter. It 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 set and sets the rest of the structure to 0. For example, an inbound socket in the internet domain would cause the name to point to a *sockaddr_in* structure with the *sin_family* field set to `AF_INET`, and all other fields cleared.

Stream sockets are not assigned a name until a call is successful: `bind()`, `connect()`, or `accept()`.

The `getsockname()` call is often used to discover the port assigned to a socket after the socket has been implicitly bound to a port. For example, an application can call `connect()` without previously calling `bind()`. In this case, the `connect()` call completes the binding necessary by assigning a port to the socket. This assignment can be detected using a call to `getsockname()`.

For `AF_IUCV`, *namelen* must be initialized to indicate the size of the space pointed to by *name*, and is set to the number of bytes copied into the space before the call returns. For `AF_INET`, the input value in the contents of *namelen* is ignored, but set before the call returns.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
int getsockname(int s, struct sockaddr *name, int *namelen)
```

Parameter	Description
-----------	-------------

<i>s</i>	The socket descriptor.
-----------------	------------------------

name

The address of the buffer into which getsockname() copies the name of s.

namelen

Points to a fullword containing the size of the address structure pointed to by *name* in bytes.

Return values

A value of 0 indicates success; the value -1 indicates an error. Errno identifies the specific error.

Errno Value**Description****EBADF**

The s parameter is not a valid socket descriptor.

EFAULT

Using the *name* and *namelen* parameters as specified would result in an attempt to access storage outside of the caller address space.

Related calls

accept(), bind(), connect(), getpeername(), socket()

getsockopt()

The getsockopt() call gets options associated with a socket. It can be called only for sockets in the AF_INET domain. This call is not supported in the AF_IUCV domain. While options can exist at multiple protocol levels, they are always present at the highest socket level.

When manipulating socket options, you must specify the level at which the option resides and the name of that option. To manipulate options at the socket level, the *level* parameter must be set to SOL_SOCKET as defined in SOCKET.H. To manipulate options at the TCP level, the level parameter must be set to IPPROTO_TCP as defined in SOCKET.H. To manipulate options at any other level, such as the IP level, supply the appropriate protocol number for the protocol controlling the option. Currently, the SOL_SOCKET, IPPROTO_TCP, and IPPROTO_IP levels are supported. The getprotobyname() call can be used to return the protocol number for a named protocol.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>

int getsockopt(int s, int level, int optname, char *optval, int *optlen
```

Parameter**Description*****s***

The socket descriptor.

level

The level to which the option is set.

optname

The name of a specified socket option. See Appendix D, “GETSOCKOPT/SETSOCKOPT command values,” on page 767 for the numeric values of *optname*.

optval

Points to option data.

optlen

Points to the length of the option data.

The *optval* and *optlen* parameters are used to return data used by the particular get command. The *optval* parameter points to a buffer that is to receive the data requested by the get command. The *optlen*

parameter points to the size of the buffer pointed to by the *optval* parameter. Initially, this size must be set to the size of that buffer before calling `getsockopt()`. On return it is set to the size of the data actually returned.

All of the socket level options except `SO_LINGER` expect *optval* to point to an integer and *optlen* to be set to the size of an integer. When the integer is nonzero, the option is enabled. When it is 0, the option is disabled. The `SO_LINGER` option expects *optval* to point to a *linger* structure as defined in `SOCKET.H`. This structure is defined in the following example:

```
#include <manifest.h>
struct linger
{
    int    l_onoff;           /* option on/off */
    int    l_linger;         /* linger time */
};
```

The *l_onoff* field is set to 0 if the `SO_LINGER` option is being disabled. A nonzero value enables the option. The *l_linger* field specifies the amount of time to linger on close.

The following option is recognized at the TCP level (`IPPROTO_TCP`):

Option	
Description	

TCP_NODELAY

Returns the status of Nagle algorithm (RFC 896). This option is not supported for `AF_IUCV` sockets.

When *optval* is 0, Nagle algorithm is enabled and TCP will wait to send small packets of data until the acknowledgment for the previous data is received.

When *optval* is nonzero, Nagle algorithm is disabled and TCP will send small packets of data even before the acknowledgment for previous data sent is received.

The following options are recognized at the socket level (`SOL_SOCKET`):

Option	
Description	

SO_ACCEPTCONN

Indicates whether `listen()` was called for a stream socket.

SO_BROADCAST

Toggles the ability to broadcast messages. The default is *disabled*. When this option is enabled, it allows the application to send broadcast messages over *s* when the interface specified in the destination supports the broadcasting of packets. This option has no meaning for stream sockets.

SO_CLUSTERCONNTYPE

Returns a bit mapped 32-bit value. One or more than one of the following values will be returned:

- No Conn means that the socket is not connected.
- None means that the socket is active, but the partner is not in the same cluster. If this indicator is set, the other 3 indicators are 0.
- Same cluster means that the connection partners are in the same cluster.
- Same image means that the connection partners are in the same MVS image.
 `SO_CLUSTERCONNTYPE_SAME_CLUSTER` will also be set. If the connection partner is a distributed DVIPA, the same image bit will not be on since the exact hosting stack is not known.
- Internal means that communication from this node to the stack hosting the partner application is not sent on links/interfaces exposed outside the cluster (`sysplex`). To determine if both ends of the connection flow over internal links/interfaces, the partner application must also issue this `getsockopt()` and both ends exchange their results from this socket call (through an application-dependent method).

Note: If the destination IP address for a connection (partner's IP address) is a dynamic VIPA or distributed dynamic VIPA residing in the cluster, the internal indicator will not be on because traffic

for these connections can be forwarded to the target TCP/IP stacks over links or interfaces that are external to the cluster.

An internal indicator means that for this side of the connection, the link/interface type is one of the following types:

- CTC
- HiperSockets (iQDIO)
- MPCPTP (including XCF and IUTSAMEH connections)
- OSA-Express QDIO with CHPID type OSX or OSM
- Loopback
- Or both partners are owned by the same multi-homed stack

On return, one or more of the following bits are set:

```
00000000 00000000 00000000 00000001' -SO_CLUSTERCONNTYPE_NONE
00000000 00000000 00000000 00000010' -SO_CLUSTERCONNTYPE_SAME_CLUSTER
00000000 00000000 00000000 00000100' -SO_CLUSTERCONNTYPE_SAME_IMAGE
00000000 00000000 00000000 00001000' -SO_CLUSTERCONNTYPE_INTERNAL
00000000 00000000 00000000 00000000' -SO_CLUSTERCONNTYPE_NOCONN
```

Note: A value of all zeros means that there is no active connection on the socket. This is usually the case for a listening socket. This is also true for a client socket before the client issues `connect()`. The caller should first check for a returned value of `SO_CLUSTERCONNTYPE_NOCONN` before checking for any of the other returned indicators.

If `getsockopt()` (`SO_CLUSTERCONNTYPE`) is issued before the connection has been established, it results in a return value of 0.

If the application issues `getsockopt()` (`SO_CLUSTERCONNTYPE`) on a connected socket, and has received an indication of `SO_CLUSTERCONNTYPE_INTERNAL`, any subsequent rerouting decisions because of current route failure will either select an alternate route, which is also `SO_CLUSTERCONNTYPE_INTERNAL`, or fail the connection with no route available indications. This means that when an application has received an indication of `SO_CLUSTERCONNTYPE_INTERNAL` on a connection, any subsequent rerouting preserves that indication on the new route, or will fail the connection. This ensures that a connection that an application relies on as being internal does not transparently become non-internal because of a routing change.

If the application never issues the new `getsockopt()` or if the connection was previously reported as not `SO_CLUSTERCONNTYPE_INTERNAL`, rerouting decisions are made as at present, and the rerouting is transparent to the application as long as any alternate route exists.

SO_DEBUG

The `sock_debug()` call provides the socket library tracing facility. The *onoff* parameter can have a value of 0 or nonzero. When *onoff*=0 (the default), no socket library tracing is done; when *onoff*=nonzero, the system traces for socket library calls and interrupts.

SO_ERROR

Returns any error pending on the socket and clears the error status. It can be used to check for asynchronous errors on connected datagram sockets and for other asynchronous errors (errors returned explicitly by one of the socket calls).

SO_KEEPAIVE

Toggles the TCP keep alive mechanism for a stream socket. The default is *disabled*. When activated, the keep alive mechanism periodically sends a packet along an otherwise idle connection. If the remote TCP does not respond to the packet or to retransmissions of the packet, the connection is terminated with the error ETIMEDOUT.

SO_LINGER

Lingers on close if data is present. The default is *disabled*. When this option is enabled and there is unsent data present when `close()` is called, the calling application is blocked during the `close()` call until the data is transmitted, or the connection has timed out. If this option is disabled, the `close()` call returns without blocking the caller and the TCP/IP address space still waits before trying to

send the data. Although the data transfer is usually successful, it cannot be guaranteed, because the TCP/IP address space waits only a finite amount of time while trying to send the data. This option has meaning only for stream sockets.

SO_OOBLINE

Toggles reception of out-of-band data. The default is *disabled*. When this option is enabled, it causes out-of-band data to be placed in the normal data input queue as it is received, making it available to `recv()`, `recvfrom()`, and `recvmsg()` without specifying the `MSG_OOB` flag in those calls. When this option is disabled, it causes out-of-band data to be placed in the priority data input queue as it is received, making it available to `recv()`, `recvfrom()`, and `recvmsg()` only by specifying the `MSG_OOB` flag in those calls. This option has meaning only for stream sockets.

SO_RCVBUF

Returns the size of the data portion of the TCP/IP send buffer in *optval*. The size of the data portion of the receive buffer is protocol-specific.

SO_REUSEADDR

Toggles local address reuse. The default is disabled. This alters the normal algorithm used in the `bind()` call.

The normal `bind()` call algorithm allows each internet address and port combination to be bound only once. If the address and port have been bound already, a subsequent `bind()` will fail and return error `EADDRINUSE`.

After the `SO_REUSEADDR` option is active, the following situations are supported:

- A server can `bind()` the same port multiple times as long as every invocation uses a different local IP address, and the wildcard address `INADDR_ANY` is used only one time per port.
- A server with active client connections can be restarted and can bind to its port without having to close all of the client connections.
- For datagram sockets, multicasting is supported so multiple `bind()` calls can be made to the same class D address and port number.

SO_SNDBUF

Returns the size of the data portion of the TCP/IP send buffer in *optval*. The size of the data portion of the send buffer is protocol-specific.

SO_TYPE

Returns the type of the socket. On return, the integer pointed to by *optval* is set to `SOCK_STREAM`, `SOCK_DGRAM`, or `SOCK_RAW`.

The following options are recognized at the IP level (`IPPROTO_IP`):

Option

Description

IP_MULTICAST_TTL

Gets the IP time to live of outgoing multicast datagrams. The default value is 1 (multicast is available only to the local subnet).

IP_MULTICAST_LOOP

Used to determine whether outgoing multicast datagrams are looped back.

IP_MULTICAST_IF

Gets the interface for sending outbound multicast datagrams from the socket application.

Note: Multicast datagrams can be transmitted only on one interface at a time.

Return values

The value 0 indicates success; the value -1 indicates an error. `Errno` identifies the specific error.

Errno

Description

EBADF

The *s* parameter is not a valid socket descriptor.

EFAULT

Using *optval* and *optlen* parameters would result in an attempt to access storage outside the caller address space.

EINVAL

The *optname* parameter is unrecognized, or the *level* parameter is not SOL_SOCKET.

Example

The following examples show the `getsockopt()` call. See [“setsockopt\(\)” on page 152](#) to see how the `setsockopt()` call options are set.

Example 1

```
#include <manifest.h>

int rc;
int s;
int optval;
int optlen;
struct linger l;
int getsockopt(int s, int level, int optname, char *optval, int *optlen);

:
/* Is out of band data in the normal input queue? */
optlen = sizeof(int);
rc = getsockopt(
    s, SOL_SOCKET, SO_OOBINLINE, (char *) &optval, &optlen);
if (rc == 0)
{
    if (optlen == sizeof(int))
    {
        if (optval)
            /* yes it is in the normalqueue */
        else
            /* no it is not */
    }
}

:
```

Example 2

```
/* Do I linger on close? */
optlen = sizeof(l);
rc = getsockopt(
    s, SOL_SOCKET, SO_LINGER, (char *) &l, &optlen);
if (rc == 0)
{
    if (optlen == sizeof(l))
    {
        if (l.l_onoff)
            /* yes I linger */
        else
            /* no I do not */
    }
}

}
```

Related calls

`bind()`, `close()`, `getprotobyname()`, `setsockopt()`, `socket()`

givesocket()

The `givesocket()` call tells TCP/IP to make the specified socket available to a `takesocket()` call issued by another program. Any connected stream socket can be given. Typically, `givesocket()` is used by a master program that obtains sockets by means of `accept()` and gives them to secondary programs that handle one socket at a time.

This call can be used only in the `AF_INET` domain.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>

int givesocket(int d, struct clientid *clientid)
```

Parameter

Description

d

The descriptor of a socket to be given to another application.

clientid

Points to a client ID structure specifying the target program to which the socket is to be given.

To pass a socket, the master program first calls `givesocket()` with the client ID structure that is specified as follows:

Field

Description

domain

This call is supported only in the `AF_INET` domain.

name

The secondary program address space name, left-justified and padded with blanks. The secondary program can run in the same address space as the master program, in which case this field is set to the master program address space. If this field is set to blanks, any MVS address space can take the socket.

subtaskname

Specifies blanks.

reserved

Specifies binary zeros.

The master program then calls `getclientid()` to obtain its client ID, and passes its client ID, along with the descriptor of the socket to be given, to the secondary program. One way to pass a socket is by passing the secondary program startup parameter list.

The secondary program calls `takesocket()`, specifying the master program client ID and socket descriptor.

Waiting for the secondary program to take the socket, the master program uses `select()` to test the given socket for an exception condition. When `select()` reports that an exception condition is pending, the master program calls `close()` to free the given socket. If `select()` reports a timeout has occurred, that is, the socket has not been taken by secondary program, the master program should take the socket that was given by calling `takesocket()`. The master program then owns the socket again and should call `close()` to close the socket.

If your program closes the socket before a pending exception condition is indicated, the TCP connection is immediately reset, and the target program call to `takesocket()` call is unsuccessful. Calls other than the `close()` call issued on a given socket return a value of -1, with `errno` set to `EBADF`.

Sockets that have been given and not taken for a period of four days will be closed and become unavailable. If a `select` for the socket is outstanding, it is posted.

Return values

The value 0 indicates success. The value -1 indicates an error. `Errno` identifies a specific error.

Errno

Description

EBADF

The *d* parameter is not a valid socket descriptor. The socket has already been given. The socket domain is not AF_INET.

EBUSY

Listen() has been called for the socket.

EFAULT

Using the *clientid* parameter as specified would result in an attempt to access storage outside the caller address space.

EINVAL

The *clientid* parameter does not specify a valid client identifier.

ENOTCONN

The socket is not connected.

EOPNOTSUPP

The socket type is not SOCK_STREAM.

Related calls

accept(), close(), getclientid(), listen(), select(), takesocket()

htonl()

The htonl() call translates a long integer from host byte order to network byte order.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>

unsigned long htonl(unsigned long a)
```

Parameter

Description

a

The unsigned long integer to be put into network byte order.

Return values

Returns the translated long integer.

Related calls

htons(), ntohs(), ntohl()

htons()

The htons() call translates a short integer from host byte order to network byte order.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>

unsigned short htons(unsigned short a)
```

Parameter

Description

a

The unsigned short integer to be put into network byte order.

Return values

Returns the translated short integer.

Related calls

ntohs(), htonl(), ntohl()

inet_addr()

The `inet_addr()` call interprets character strings representing host addresses expressed in standard dotted decimal notation and returns host addresses suitable for use as internet addresses.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>
#include <inet.h>

unsigned long inet_addr(char *cp)
```

Parameter

Description

cp

A character string in standard dotted decimal (.) notation

Values specified in standard dotted decimal notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When a four-part address is specified, each part is interpreted as a byte of data and assigned, from left to right, to one of the four bytes of an internet address.

When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the two rightmost bytes of the network address. This makes the three-part address a good format for specifying Class B network addresses as `128.net.host`.

When a two-part address is specified, the last part is interpreted as a 24-bit quantity and placed in the three rightmost bytes of the network address. This makes the two-part address a good format for specifying Class A network addresses as `net.host`.

When a one-part address is specified, the value is stored directly in the network address space without any rearrangement of its bytes.

Numbers supplied as address parts in standard dotted decimal notation can be decimal, hexadecimal, or octal. Numbers are interpreted using C language syntax. A leading `0x` implies hexadecimal; a leading `0` implies octal. A number without a leading `0` implies decimal.

Return values

The internet address is returned in network byte order.

A value of -1 is returned as an error.

Related calls

`inet_lnaof()`, `inet_makeaddr()`, `inet_netof()`, `inet_network()`, `inet_ntoa()`

`inet_lnaof()`

The `inet_lnaof()` call breaks apart the existing internet host address, and returns the local network address portion.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>
#include <inet.h>

unsigned long inet_lnaof(struct in_addr in)
```

Parameter

Description

in

The host internet address

Return values

The local network address is returned in host byte order.

Related calls

`inet_addr()`, `inet_makeaddr()`, `inet_netof()`, `inet_network()`, `inet_ntoa()`

`inet_makeaddr()`

The `inet_makeaddr()` call combines an existing network number and a local network address to construct an internet address.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>
#include <inet.h>

struct in_addr
inet_makeaddr(unsigned long net, unsigned long lna)
```

Parameter

Description

net

The network number

lna

The local network address

Return values

The internet address is returned in network byte order.

Related calls

`inet_addr()`, `inet_lnaof()`, `inet_netof()`, `inet_network()`, `inet_ntoa()`

inet_netof()

The `inet_netof()` call breaks apart the existing internet host address and returns the network number portion.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>
#include <inet.h>

unsigned long inet_netof(struct in_addr in)
```

Parameter Description

in
The internet address in network byte order

Return values

The network number is returned in host byte order.

Related calls

`inet_addr()`, `inet_lnaof()`, `inet_makeaddr()`, `inet_ntoa()`

inet_network()

The `inet_network()` call interprets character strings representing addresses expressed in standard dotted decimal notation and returns numbers suitable for use as a network number.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>
#include <inet.h>

unsigned long inet_network(char *cp)
```

Parameter Description

cp
A character string in standard, dotted decimal (.) notation

Return values

The network number is returned in host byte order.

Related calls

`inet_addr()`, `inet_lnaof()`, `inet_makeaddr()`, `inet_ntoa()`

inet_ntoa()

The `inet_ntoa()` call returns a pointer to a string expressed in dotted decimal notation. The `inet_ntoa()` call accepts an internet address expressed as a 32-bit quantity in network byte order and returns a string expressed in dotted decimal notation.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>
#include <inet.h>

char *inet_ntoa(struct in_addr in)
```

Parameter	Description
-----------	-------------

in

The host internet address

Return values

Returns a pointer to the internet address expressed in dotted decimal notation

Related calls

`inet_addr()`, `inet_lnaof()`, `inet_makeaddr()`, `inet_network()`, `inet_ntoa()`

ioctl()

The operating characteristics of sockets can be controlled using the `ioctl()` call.

Rules:

- This call can be used only in the `AF_INET` domain.
- Only the `ioctl()` commands that are documented in this topic are supported by this API.

```
#include <manifest.h>
#include <socket.h>
#include <bsdtypes.h>
#include <ioctl.h>
#include <rtroute.h>
#include <if.h>
#include <ezbcmnc.h>
int ioctl(int s, unsigned long cmd, char *arg)
```

Parameter	Description
-----------	-------------

s

The socket descriptor.

cmd

The command to perform.

arg

Points to the data associated with *cmd*.

The operations to be controlled are determined by *cmd*. The *arg* parameter points to data associated with the particular command, and its format depends on the command being requested. The following keywords are valid `ioctl()` keywords:

Keyword	Description
---------	-------------

FIONBIO

Sets or clears nonblocking I/O for a socket. The variable *arg* points to an integer. If the integer is 0, nonblocking I/O on the socket is cleared; otherwise, the socket is set for nonblocking I/O.

FIONREAD

Gets for the socket the number of immediately readable bytes. The variable *arg* points to an integer.

SIOCADDRT

Adds a routing table entry. The variable *arg* points to a *rtentry* structure, as defined in RTRROUTE.H. The routing table entry, passed as an argument, is added to the routing tables.

SIOCATMARK

Queries whether the current location in the data input is pointing to out-of-band data. The variable *arg* points to an integer of 1 when the socket points to a mark in the data stream for out-of-band data; otherwise, it points to 0.

SIODELRT

Deletes a routing table entry. The variable *arg* points to a *rtentry* structure, as defined in RTRROUTE.H. If the structure exists, the routing table entry passed as an argument is deleted from the routing tables.

SIOCGIFADDR

Gets the network interface address. The variable *arg* points to an *ifreq* structure, as defined in IF.H. The interface address is returned in the argument.

SIOCGIFBRDADDR

Gets the network interface broadcast address. The variable *arg* points to an *ifreq* structure, as defined in IF.H. The interface broadcast address is returned in the argument.

SIOCGIFCONF

Gets the network interface configuration. The variable *arg* points to an *ifconf* structure, as defined in IF.H. The interface configuration is returned in the argument.

SIOCGIFDSTADDR

Gets the network interface destination address. The variable *arg* points to an *ifreq* structure, as defined in IF.H. The interface destination (point-to-point) address is returned in the argument.

SIOCGIFFLAGS

Gets the network interface flags. The variable *arg* points to an *ifreq* structure, as defined in IF.H. The interface flags are returned in the argument.

SIOCGIFMETRIC

Gets the network interface routing metric. The variable *arg* points to an *ifreq* structure, as defined in IF.H. The interface routing metric is returned in the argument.

SIOCGIFNETMASK

Gets the network interface network mask. The variable *arg* points to an *ifreq* structure, as defined in IF.H. The interface network mask is returned in the argument.

SIOSIFMETRIC

Sets the network interface routing metric. The variable *arg* points to an *ifreq* structure, as defined in IF.H. The interface routing metric is set to the value passed in the argument.

SIOCGMONDATA

Returns TCP/IP stack statistical counters. The variable *arg* points to a MonDataIn structure. The counters are returned in a MonDataOut structure. Both of these structures are defined in EZBZMONC in SEZANMAC.

Note: The ARP counter data provided differs depending on the type of device. See [z/OS Communications Server: IP Configuration Guide](#) for information about devices that support ARP Offload and what is supported for each device.

Return values

The value 0 indicates success; the value -1 indicates an error. Errno identifies the specific error.

Errno**Description****EBADF**

The *s* parameter is not a valid socket descriptor.

EINVAL

The request is not valid, or not supported.

EFAULT

The arg is a bad pointer.

Example

```
int s;
int dontblock;
int rc;
...
/* Place the socket into nonblocking mode */
dontblock = 1;
rc = ioctl(s, FIONBIO, (char *) &dontblock);
...
```

listen()

The `listen()` call applies only to stream sockets. It performs two tasks: it completes the binding necessary for a socket `s`, if `bind()` has not been called for `s`, and it creates a connection request queue of length *backlog* to queue incoming connection requests. When the queue is full, additional connection requests are ignored.

The `listen()` call indicates a readiness to accept client connection requests. It transforms an active socket into a passive socket. After called, `s` can never again be used as an active socket to initiate connection requests. Calling `listen()` is the third of four steps that a server performs when it accepts a connection. It is called after allocating a stream socket using `socket()`, and after binding a name to `s` using `bind()`. It must be called before calling `accept()`.

```
#include <manifest.h>
#include <socket.h>
int listen(int s, int backlog)
```

Parameter**Description****s**

Socket descriptor

backlog

Maximum queue length for pending connections

If the backlog is less than 0, *backlog* is set to 0. If the backlog is greater than `SOMAXCONN`, as defined in the `TCPIP.PROFILE` file, *backlog* is set to `SOMAXCONN`. There is a `SOMAXCONN` variable in the `SOCKET.H` file that is hardcoded at 10. If your C socket programs use this variable to determine the maximum `listen()` *backlog* queue length, remember to change the header file to reflect the value you specified for TCP/IP in `TCPIP.PROFILE`.

Return values

The value 0 indicates success; the value -1 indicates an error. `Errno` identifies the specific error.

Errno**Description****EBADF**

The `s` parameter is not a valid socket descriptor.

EOPNOTSUPP

The `s` parameter is not a socket descriptor that supports the `listen()` call.

Related calls

accept(), bind(), connect(), socket()

maxdesc()

The maxdesc() call reserves additional space in the TCP/IP address space to allow socket numbers to extend beyond the default range of 0 through 49. Socket numbers 0, 1, and 2 are never assigned, so the default maximum number of sockets is 47.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>

int maxdesc(int *totdesc, int *inetdesc)
```

Parameter Description

totdesc

Points to an integer containing a value one greater than the largest socket number desired. The maximum allowed value is 2000.

Note: If a totdesc value greater than 2000 is specified, the internal value is set to 2000. In all cases, use getdtablesize() to verify the value set by maxdesc().

inetdesc

Points to an integer containing a value one greater than the largest socket number desired. The maximum value, usable for AF_INET sockets, allowed is 2000.

Set the integer pointed to by *totdesc* to one more than the maximum socket number desired. If your program does not use AF_INET sockets, set the integer pointed to by *inetdesc* to 0. If your program uses AF_INET sockets, set the integer pointed to by *inetdesc* to the same value as *totdesc*; maxdesc() must be called before your program creates its first socket. Your program should use getdtablesize() to verify that the number of sockets has been changed.

Note: Increasing the size of the bit sets for the select() call must be done at compile time. To increase the size of the bit sets, before including BSDTYPES.H, define FD_SETSIZE to be the largest value of any socket. The default size of FD_SETSIZE is 255 sockets.

Return values

The value 0 indicates success. (Your application should check the integer pointed to by *inetdesc*. It might contain less than the original value, if there was insufficient storage available in the TCP/IP address space. In this case, the desired number of AF_INET sockets are not available.) The value -1 indicates an error. Errno identifies the specific error.

Errno Description

EFAULT

Using the *totdesc* or *inetdesc* parameters as specified results in an attempt to access storage outside of the caller address space, or storage not able to be modified by the caller.

EALREADY

Your program called maxdesc() after creating a socket, or after a previous call to maxdesc().

EINVAL

Indicates that **totdesc* is less than **inetdesc*; **totdesc* is less than or equal to 0; or **inetdesc* is less than 0.

ENOMEM

Your address space lacks sufficient storage.

Example

```
int totdesc, inetdesc;
totdesc = 100;
inetdesc = 0;
rc = maxdesc(&totdesc, &inetdesc)
```

If successful, your application can create 97 sockets, all of type AF_IUCV. Socket numbers run from 3–99.

```
int totdesc, inetdesc;
totdesc = 100;
inetdesc = 100;
rc = maxdesc(&totdesc, &inetdesc)
```

If successful, your application can create 97 sockets, each of which can be of type AF_INET or AF_IUCV. The socket numbers run from 3–99.

Related calls

select(), socket(), getdtablesize()

ntohl()

The ntohl() call translates a long integer from network byte order to host byte order.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>

unsigned long ntohl(unsigned long a)
```

Parameter

Description

a

The unsigned long integer to be put into host byte order

Return values

Returns the translated long integer

Related calls

htonl(), htons(), ntohs()

ntohs()

The ntohs() call translates a short integer from network byte order to host byte order.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>

unsigned short ntohs(unsigned short a)
```

Parameter

Description

a

The unsigned short integer to be put into host byte order

Return values

Returns the translated short integer

Related calls

ntohl(), htons(), htonl()

read()

The `read()` call reads data on a socket with descriptor `s` and stores it in a buffer. The `read()` call applies only to connected sockets. This call returns as many as `len` bytes of data. If fewer than the number of bytes requested is available, the call returns the number currently available. If data is not available for the socket `s`, and `s` is in blocking mode, the `read()` call blocks the caller until data arrives. If data is not available, and `s` is in nonblocking mode, `read()` returns a -1 and sets `errno` to `EWOULDBLOCK`. See “`ioctl()`” on page 129, or “`fcntl()`” on page 103 for a description of how to set nonblocking mode.

If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard extra bytes. Stream sockets act like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and Application A sends 1000 bytes, each call to this function can return 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, calling this function until all data has been received.

```
#include <manifest.h>
#include <socket.h>
int read(int s, char *buf, int len)
```

Parameter

Description

s

Socket descriptor.

buf

Points to the buffer that receives the data.

len

Length in bytes of the buffer pointed to by `buf`.

Return values

If successful, the number of bytes copied into the buffer is returned. The value 0 indicates that the connection is closed. The value -1 indicates an error. `Errno` identifies the specific error.

Errno

Description

EBADF

Indicates that `s` is not a valid socket descriptor.

EFAULT

Using the `buf` and `len` parameters would result in an attempt to access storage outside the caller address space.

EWOULDBLOCK

Indicates an unconnected socket (RAW).

Note: `ENOTCONN` is returned for TCP, and `EINVAL` is returned for UDP.

EMSGSIZE

For non-TCP sockets, this indicates that the length exceeds the maximum data size. This is determined by `getsockopt()` using `SO_SNDBUF` for the socket type (TCP, UDP, or RAW).

Related calls

connect(), fcntl(), getsockopt(), ioctl(), readv(), recv(), recvmsg(), recvfrom(), select(), selectex(), send(), sendmsg(), sendto(), setsockopt(), socket(), write(), writev()

readv()

The readv() call reads data on a socket with descriptor *s* and stores it in a set of buffers. The readv() call applies to connected sockets only.

```
#include <manifest.h>
#include <socket.h>
#include <bsdtypes.h>
#include <uio.h>

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

Parameter

Description

s

The socket descriptor.

iov

Points to an iovec structure.

iovcnt

The number of buffers pointed to by the *iov* parameter.

The data is scattered into the buffers specified by *iov*[0]...*iov*[*iovcnt*-1]. The *iovec* structure is defined in UIO.H and contains the following variables:

Variable

Description

iov_base

Points to the buffer.

iov_len

The length of the buffer.

The readv() call applies only to connected sockets.

This call returns up to *len* bytes of data. If less than the number of bytes requested is available, the call returns the number currently available. If data is not available for the socket *s*, and *s* is in blocking mode, the readv() call blocks the caller until data arrives. If data is not available and *s* is in nonblocking mode, readv() returns a -1 and sets *errno* to EWOULDBLOCK. See “fcntl() ” on page 103 or “ioctl() ” on page 129 for a description of how to set nonblocking mode. When a datagram packet is too long to fit in the supplied buffer, datagram sockets discard extra bytes. Stream sockets act like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and Application A sends 1000 bytes, each call to this function can return 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop and call this function until all data has been received.

Return values

If successful, the number of bytes read into the buffers is returned. The value 0 indicates that the connection is closed. The value -1 indicates an error. *Errno* identifies the specific error.

Errno

Description

EBADF

Indicates that *s* is not a valid socket descriptor.

EFAULT

Using *iov* and *iovcnt* would result in an attempt to access storage outside the caller address space.

EINVAL

Iovcnt was not valid, or one of the fields in the *iov* array was not valid. Also returned for a NULL *iov* pointer.

EWOULDBLOCK

Indicates that *s* is in nonblocking mode and data is not available to read.

Related calls

connect(), fcntl(), getsockopt(), ioctl(), read(), recv(), recvmsg(), recvfrom(), select(), selectex(), send(), sendmsg(), sendto(), setsockopt(), socket(), write(), writev()

recv()

The `recv()` call receives data on a socket with descriptor *s* and stores it in a buffer. The `recv()` call applies only to connected sockets.

This call returns the length of the incoming message or data. If data is not available for socket *s*, and *s* is in blocking mode, the `recv()` call blocks the caller until data arrives. If data is not available and *s* is in nonblocking mode, `recv()` returns a -1 and sets `errno` to `EWOULDBLOCK`. See [“fcntl\(\)” on page 103](#) or [“ioctl\(\)” on page 129](#) for a description of how to set nonblocking mode.

If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard extra bytes. Stream sockets act like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and Application A sends 1000 bytes, each call to this function can return 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop and call this function until all data has been received.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>

int recv(int s, char *buf, int len, int flags)
```

Parameter**Description*****s***

Socket descriptor.

buf

Points to the buffer that receives the data.

len

Length in bytes of the buffer pointed to by *buf*.

flags

Set the *flags* parameter by specifying one or more of the following flags. If more than one flag is specified, the logical OR operator (`|`) must be used to separate them. Setting this parameter is supported only for sockets in the `AF_INET` domain. Setting these flags is not supported for sockets in the `AF_IUCV` domain.

MSG_OOB

Reads any out-of-band data on the socket. This is valid for stream (TCP) sockets only.

MSG_PEEK

Peeks at the data present on the socket; the data is returned but not consumed, so that a subsequent receive operation sees the same data.

Return values

If successful, the byte length of the message or datagram is returned. The value -1 indicates an error. The value 0 indicates connection closed. Errno identifies the specific error.

Errno

Description

EBADF

Indicates that *s* is not a valid socket descriptor.

EFAULT

Using the *buf* and *len* parameters would result in an attempt to access storage outside the caller address space.

EWOULDBLOCK

Indicates that *s* is in nonblocking mode and data is not available to read.

ENOTCONN

Indicates an unconnected TCP socket.

EMSGSIZE

For non-TCP sockets, this indicates that length exceeds the maximum data size as determined by `getsockopt()` using `SO_SNDBUF` for the socket type, either TCP, UDP, or RAW.

Related calls

`connect()`, `fcntl()`, `getsockopt()`, `ioctl()`, `read()`, `readv()`, `recvfrom()`, `recvmsg()`, `select()`, `selectex()`, `send()`, `sendmsg()`, `sendto()`, `setsockopt()`, `socket()`, `write()`, `writev()`

recvfrom()

The `recvfrom()` call receives data on a socket by name with descriptor *s* and stores it in a buffer. The `recvfrom()` call applies to any datagram socket, whether connected or unconnected. For a datagram socket, when *name* is nonzero, the source address of the message is filled. Parameter *namelen* must first be initialized to the size of the buffer associated with *name*; then it is modified on return to indicate the actual size of the address stored there.

This call returns the length of the incoming message or data. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard extra bytes. If data is not available for the socket *s*, and *s* is in blocking mode, the `recvfrom()` call blocks the caller until data arrives. If data is not available, and *s* is in nonblocking mode, `recvfrom()` returns a -1 and sets `errno` to `EWOULDBLOCK`. See [“fcntl\(\)” on page 103](#) or [“ioctl\(\)” on page 129](#) to set nonblocking mode.

For datagram sockets, this call returns the entire datagram sent, providing the datagram can fit into the specified buffer. Stream sockets act like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket, and Application A sends 1000 bytes, each call to this function can return 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, calling this function until all data has been received.

For datagram protocols, `recvfrom()` returns the source address associated with each incoming datagram. For connection-oriented protocols like TCP, `getpeername()` returns the address associated with the remote end of the connection.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>

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

Parameter

Description

s

Socket descriptor.

buf

Pointer to the buffer to receive the data.

len

Length in bytes of the buffer pointed to by *buf*.

flags

A parameter that can be set to 0 or MSG_PEEK, or MSG_OOB. Setting this parameter is supported only for sockets in the AF_INET domain. Setting these flags is not supported for sockets in the AF_IUCV domain.

MSG_OOB

Reads any out-of-band data on the socket. This is valid for stream (TCP) sockets only.

MSG_PEEK

Peeks at the data present on the socket; the data is returned but not consumed, so that a subsequent receive operation sees the same data.

name

Points to a *socket address* structure from which data is received. If *name* is a nonzero value, the source address is returned (datagram sockets).

namelen

Points to the size of *name* in bytes.

Return values

If successful, the length of the message or datagram is returned in bytes. The value 0 indicates that the connection is closed. The value -1 indicates an error. Errno identifies the specific error.

Errno

Description

EBADF

Indicates that *s* is not a valid socket descriptor.

EFAULT

Using the *buf* and *len* parameters would result in an attempt to access storage outside the caller address space.

EWOULDBLOCK

Indicates that *s* is in nonblocking mode and data is not available to read.

ENOTCONN

Indicates an unconnected TCP socket.

EMSGSIZE

For non-TCP sockets, this indicates that length exceeds the maximum data size as determined by getsockopt() using SO_SNDBUF for the socket type, either TCP, UDP, or RAW.

EINVAL

Parameter *namelen* is not valid.

Related calls

fcntl(), getsockopt(), ioctl(), read(), readv(), recv(), recvmsg(), select(), selectex(), send(), sendmsg(), sendto(), setsockopt(), socket(), write(), writev()

recvmsg()

The recvmsg() call receives messages on the socket with descriptor *s* and stores the messages in an array of message headers.

For datagram protocols, `recvmsg()` returns the source address associated with each incoming datagram. For connection-oriented protocols like TCP, `getpeername()` returns the address associated with the remote end of the connection.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>

int recvmsg(int s, struct msghdr *msg, int flags)
```

Parameter	Description
-----------	-------------

s

Socket descriptor.

msg

Points to an `msghdr` structure.

flags

Set the *flags* parameter by specifying one or more of the following flags. If more than one flag is specified, the logical OR operator (`|`) must be used to separate them. Setting this parameter is supported only for sockets in the `AF_INET` domain. Setting these flags is not supported for sockets in the `AF_IUCV` domain.

MSG_OOB

Reads any out-of-band data on the socket. This is valid for stream (TCP) sockets only.

MSG_PEEK

Peeks at the data present on the socket; the data is returned but not consumed, so that a subsequent receive operation will see the same data.

A message header is defined by structure `msghdr`. The definition of this structure can be found in the `SOCKET.H` header file and contains the following elements:

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

msg_name

An optional pointer to a buffer where the sender address is stored for datagram sockets.

msg_namelen

The size of the address buffer.

msg_iov

An array of `iovec` buffers into which the message is placed. An `iovec` buffer contains the following variables:

iov_base

Points to the buffer.

iov_len

The length of the buffer.

msg_iovlen

The number of elements in the `msg_iov` array.

msg_accrights

The access rights received. This field is ignored.

msg_accrightslen

The length of access rights received. This field is ignored.

The `recvmsg()` call applies to sockets, regardless of whether they are in the connected state, except for TCP sockets, which must be connected.

This call returns the length of the data received. If data is not available for socket `s`, and `s` is in blocking mode, the `recvmsg()` call blocks the caller until data arrives. If data is not available, and `s` is in nonblocking mode, `recvmsg()` returns a -1 and sets `errno` to `EWOULDBLOCK`. See [“fcntl\(\)” on page 103](#) or [“ioctl\(\)” on page 129](#) to see how to set nonblocking mode.

If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard extra bytes. Stream sockets act like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket, and Application A sends 1000 bytes, each call to this function can return 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, and call this function until all data has been received.

Return values

If successful, the length of the message in bytes is returned. The value 0 indicates that the connection is closed. The value -1 indicates an error. Errno identifies the specific error.

Errno

Description

EBADF

Indicates that *s* is not a valid socket descriptor.

EFAULT

Using *msg* would result in an attempt to access storage outside the caller address space. Also returned when *msg_namelen* is not valid.

EWOULDBLOCK

Indicates that *s* is in nonblocking mode and data is not available to read.

ENOTCONN

Returned for an unconnected TCP socket.

EMSGSIZE

For non-TCP sockets, this indicates that length exceeds the maximum data size determined by `getsockopt()` using `SO_SNDBUF` for the socket type (TCP, UDP, or RAW).

Related calls

`connect()`, `fcntl()`, `getsockopt()`, `ioctl()`, `read()`, `readv()`, `recv()`, `recvfrom()`, `select()`, `selectex()`, `send()`, `sendmsg()`, `sendto()`, `setsockopt()`, `socket()`, `write()`, `writen()`

select()

The `select()` call monitors activity on a set of sockets looking for sockets ready for reading, writing, or with an exception condition pending.

```
#include <manifest.h>
#include <socket.h>
#include <bsdtypes.h>
#include <bsdtime.h>

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout)
```

Parameter

Description

nfds

The number of socket descriptors to be checked. This value should be one greater than the greatest number of sockets to be checked.

You can use the `select()` call to pass a bit set containing the socket descriptors for the sockets you want checked. The bit set is fixed in size using one bit for every possible socket. Use the *nfds* parameter to force `select()` to check only a subset of the allocated socket bit set.

If your application allocates sockets 3, 4, 5, 6, and 7, and you want to check all of your allocations, *nfds* should be set to 8, the highest socket descriptor you specified, plus 1. If your application checks sockets 3 and 4, *nfds* should be set to 5.

Socket numbers are assigned starting with number 3 because numbers 0, 1, and 2 are used by the C socket interface.

readfds

Points to a bit set of descriptors to check for reading.

writefds

Points to a bit set of descriptors to check for writing.

exceptfds

Points to a bit set of descriptors to check for exception conditions pending.

timeout

Points to the time to wait for select() to complete.

If *timeout* is not a NULL pointer, it specifies a maximum time to wait for the selection to complete. If *timeout* is a NULL pointer, the select() call blocks until a socket becomes ready. To poll the sockets and return immediately, *timeout* should be a non-NULL pointer to a *timeval* structure with a value of 0.

If you are using both AF_INET and AF_IUCV sockets in the socket descriptor sets, the timer value is ignored and processed as if *timeout* were a non-NULL pointer to a *timeval* structure with a value of 0.

To use select() as a timer in your program, do either of the following:

- Set the read, write, and except arrays to 0.
- Set nfds to be a NULL pointer.

To understand the implementation of the select() call, you must understand the difference between a socket and a port. TCP/IP defines ports to represent a certain process on a certain machine. A port represents the location of one process; it does not represent a connection between processes. In the MVS programming interface for TCP/IP, a socket describes an endpoint of communication. Therefore, a socket describes both a port and a machine. Like file descriptors, a socket is a small integer representing an index into a table of communication endpoints in a TCP/IP address space.

To test more than one socket at a time, place the sockets to be tested into a bit set of type FD_SET. A bit set is a string of bits that when X is an element of the set, the bit representing X is set to 1. If X is not an element of the set, the bit representing X is set to 0. For example, if Socket 33 is an element of a bit set, then bit 33 is set to 1. If Socket 33 is not an element of a bit set, then bit 33 is set to 0.

Because the bit sets contain a bit for every socket that a process can allocate, the size of the bit sets is constant. The function getdtablesize() returns the number of sockets that your program can allocate. If your program needs to allocate a large number of sockets, use getdtablesize() and maxdesc() to increase the number of sockets that can be allocated. Increasing the size of the bit sets must be done when you compile the program. To increase the size of the bit sets, define FD_SETSIZE before including BSDTYPES.H. The largest value of any socket is FD_SETSIZE, defined to be 255 in BSDTYPES.H.

The following macros can manipulate bit sets.

Macro**Description****FD_ZERO(&fdset)**

Sets all bits in bit set *fdset* to 0. After this operation, the bit set does not contain sockets as elements. This macro should be called to initialize the bit set before calling FD_SET() to set a socket as a member.

FD_SET(sock, &fdset)

Sets the bit for the socket *sock* to 1, making *sock* a member of bit set *fdset*.

FD_CLR(sock, &fdset)

Clears the bit for the socket *sock* in bit set *fdset*. This operation sets the appropriate bit to 0.

FD_ISSET(sock, &fdset)

Returns nonzero if *sock* is a member of the bit set *fdset*. Returns 0 if *sock* is not a member of *fdset*. (This operation returns the bit representing *sock*.)

A socket is ready to be read when incoming data is buffered for it, or when a connection request is pending. A call to accept(), read(), recv(), or recvfrom() does not block. To test whether any sockets are ready to be read, use FD_ZERO() to initialize the *readfds* bit set and invoke FD_SET() for each socket to be tested.

A socket is ready to be written if there is buffer space for outgoing data. A socket is ready for reading if there is data on the socket to be received. For a nonblocking stream socket in the process of connecting the `connect()` will return with a -1. The program needs to check the `errno`. If the `errno` is `EINPROGRESS` the socket is selected for write when the `connect()` completes. In the situation where the `errno` is not `EINPROGRESS`, the socket will still be selected for write which indicates that there is a pending error on the socket. A call to `write()`, `send()`, or `sendto()` does not block providing that the amount of data is less than the amount of buffer space. If a socket is selected for write, the amount of available buffer space is guaranteed to be at least as large as the size returned from using `SO_SNDBUF` with `getsockopt()`. To test whether any sockets are ready for writing, initialize *writefds* using `FD_ZERO()`, and use `FD_SET()` for each socket to be tested.

The `select()` call checks for a pending exception condition on the given socket to indicate that the target program has successfully called `takesocket()`. When `select()` indicates a pending exception condition, your program calls `close()` to close the given socket. A socket has exception conditions pending if it has received out-of-band data. A stream socket that was given using `givesocket()` is selected for exception when another application successfully takes the socket using `takesocket()`.

The programmer can pass `NULL` for any bit sets without sockets to test. For example, if a program need only check a socket for reading, it can pass `NULL` for both *writefds* and *exceptfds*.

Because the sets of sockets passed to `select()` are bit sets, the `select()` call must test each bit in each bit set before polling the socket for status. For efficiency, the *nfsd* parameter specifies the largest socket passed in any of the bit sets. The `select()` call then tests only sockets in the range 0 to *nfsd*-1. Variable *nfsd* can be the result of `getdtablesize()`, but if the application has only two sockets and *nfsd* is the result of `getdtablesize()`, `select()` tests every bit in each bit set.

Return values

The total number of ready sockets in all bit sets is returned. The value -1 indicates an error; check `errno`. The value 0 indicates an expired time limit. If the return value is greater than 0, the sockets that are ready in each bit set are set to 1. Sockets in each bit set that are not ready are set to 0. Use macro `FD_ISSET()` with each socket to test its status.

Errno

Description

EBADF

One of the bit sets specified an incorrect socket. [`FD_ZERO()` was probably not called before the sockets were set.]

EFAULT

One of the bit sets pointed to a value outside the caller address space.

EINVAL

One of the fields in the `timeval` structure is not valid.

Note: If the number of ready sockets is greater than 65535, only 65535 is reported.

Example

In the following example, `select()` is used to poll sockets for reading (socket *r*), writing (socket *w*), and exception (socket *e*) conditions.

```
/* sock_stats(r, w, e) - Print the status of sockets r, w, and e. */
int sock_stats(r, w, e)
int r, w, e;
{
    fd_set reading, writing, except;
    struct timeval timeout;
    int rc, max_sock;

    /* initialize the bit sets */
    FD_ZERO( &reading );
    FD_ZERO( &writing );
    FD_ZERO( &except );

    /* add r, w, and e to the appropriate bit set */
```

```

FD_SET( r, &reading );
FD_SET( w, &writing );
FD_SET( e, &except );

/* for efficiency, what's the maximum socket number? */
max_sock = MAX( r, w );
max_sock = MAX( max_sock, e );
max_sock ++;

/* make select poll by sending a 0 timeval */
memset( &timeout, 0, sizeof(timeout) );

/* poll */
rc = select( max_sock, &reading, &writing, &except, &timeout );

if ( rc < 0 ) {
    /* an error occurred during the select() */
    tcperror( "select" );
}
else if ( rc == 0 ) {
    /* none of the sockets were ready in our little poll */
    printf( "nobody is home.\n" );
}
else {
    /* at least one of the sockets is ready */
    printf("r is %s\n", FD_ISSET(r,&reading) ? "READY" : "NOT READY");
    printf("w is %s\n", FD_ISSET(w,&writing) ? "READY" : "NOT READY");
    printf("e is %s\n", FD_ISSET(e,&except) ? "READY" : "NOT READY");
}
}

```

Related calls

getdtablesize(), maxdesc(), selectex()

selectex()

The `selectex()` call provides an extension to the `select()` call by allowing you to use an ECB or ECB list that defines an event not described by *readfds*, *writefds*, or *exceptfds*.

The `selectex()` call monitors activity on a set of different sockets until a timeout expires to see whether any sockets are ready for reading or writing, or if any exception conditions are pending. See [“select\(\)”](#) on page 140 for more information about `selectex()`.

```

#include <manifest.h>
#include <socket.h>
#include <bsdtypes.h>
#include <bsdtime.h>

int selectex(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout, int *ecbptr)

```

Parameter

Description

nfds

The number of socket descriptors to be checked.

readfds

Points to a bit set of descriptors to be checked for reading.

writefds

Points to a bit set of descriptors to be checked for writing.

exceptfds

Points to a bit set of descriptors to be checked for exception pending conditions.

timeout

Points to the time to wait for `selectex()` to complete.

ecbptr

Points to the event control block (ECB) or ECB list. For an ECB list, the high-order bit must be turned on in *ecbptr*. The last entry in the ECB list must also have its high-order bit set to 1, signifying list end. The maximum ECBs allowed is 63.

Note: ECB list is supported only for AF_INET sockets.

Return values

The total number of ready sockets (in all bit sets) is returned. The returned value -1 indicates an error. The returned value of 0 indicates either an expired time limit or that the caller ECB has been posted. To determine which of these two conditions occurred, check the ECB value. If the value of the ECB is nonzero, then the ECB has been POSTed; otherwise, the time limit has expired. The caller must initialize the ECB value to 0 before issuing selectex(). If the caller's ECB has been POSTed, the caller descriptor sets are also set to 0. If the return value is greater than 0, the socket descriptors in each bit set that are ready are set to 1. All others are set to 0.

Errno

Description

EBADF

One of the descriptor sets specified an incorrect descriptor.

EFAULT

One of the parameters pointed to a value outside the caller address space.

EINVAL

One of the fields in the *timeval* structure is not valid.

Note: If the number of ready sockets is greater than 65535, only 65535 is reported.

Related calls

accept(), connect(), getdtablesize(), recv(), send(), select()

send()

The send() call sends datagrams on the socket with descriptor *s*. The send() call applies to all connected sockets.

If buffer space is not available to hold the socket data to be transmitted, and the socket is in blocking mode, send() blocks the caller until more buffer space becomes available. If the socket is in nonblocking mode, send() returns a -1 and sets errno to EWOULDBLOCK. See “fcntl() ” on page 103 or “ioctl() ” on page 129 to set nonblocking mode. See “select() ” on page 140 for additional information.

For datagram sockets, this call sends the entire datagram, providing the datagram can fit into the TCP/IP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application wants to send 1000 bytes, each call to this function can send 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop and call this function until all data has been sent.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>

int send(int s, char *msg, int len, int flags)
```

Parameter

Description

s

Socket descriptor.

msg

Points to the buffer containing the message to transmit.

len

Length of the message pointed to by *msg*.

flags

Set the *flags* parameter by specifying one or more of the following flags. If more than one flag is specified, the logical OR operator (`|`) must be used to separate them. Setting this parameter is supported only for sockets in the `AF_INET` domain. Setting these flags is not supported in the `AF_IUCV` domain.

MSG_OOB

Sends out-of-band data on sockets that support this function. Only `SOCK_STREAM` sockets created in the `AF_INET` address family support out-of-band data.

MSG_DONTROUTE

The `MSG_DONTROUTE` option is turned on for the duration of the operation. This is usually used only by diagnostic or routing programs.

Return values

No indication of failure to deliver is implicit in a `send()` routine. The value `-1` indicates locally detected errors. `Errno` identifies the specific error.

Errno**Description****EBADF**

Indicates that *s* is not a valid socket descriptor.

EFAULT

Using the *msg* and *len* parameters would result in an attempt to access storage outside the caller address space.

ENOBUFS

Buffer space is not available to send the message.

EWOULDBLOCK

Indicates that *s* is in nonblocking mode and there is not enough space in TCP/IP to accept the data.

Related calls

`connect()`, `fcntl()`, `getsockopt()`, `ioctl()`, `read()`, `readv()`, `recv()`, `recvfrom()`, `recvmsg()`, `select()`, `selectex()`, `sendmsg()`, `sendto()`, `socket()`, `write()`, `writew()`

sendmsg()

The `sendmsg()` call sends messages on a socket with descriptor *s* passed in an array of message headers.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>

int sendmsg(int s, struct msghdr *msg, int flags)
```

Parameter**Description*****s***

Socket descriptor.

msg

Points to an `msghdr` structure.

flags

Set the *flags* parameter by specifying one or more of the following flags. If more than one flag is specified, the logical OR operator (|) must be used to separate them. Setting this parameter is supported only for sockets in the AF_INET domain. Setting these flags is not supported in the AF_IUCV domain.

MSG_OOB

Sends out-of-band data on the socket.

MSG_DONTROUTE

The SO_DONTROUTE option is turned on for the duration of the operation; it is usually used by diagnostic or routing programs only.

A message header is defined by a msghdr. The definition of this structure can be found in the SOCKET.H header file and contains the following parameters.

Parameter

Description

msg_name

The pointer to the buffer containing the recipient address. This is required for datagram sockets where an explicit connect() has not been done.

msg_namelen

The size of the address buffer. This is required for datagram sockets where an explicit connect() has not been done.

msg_iov

An array of iovec buffers containing the message. The iovec buffer contains the following information:

iov_base

Points to the buffer.

iov_len

The length of the buffer.

msg_iovlen

The number of elements in the msg_iov array.

msg_accrights

The access rights sent. This field is ignored.

msg_accrightslen

The length of the access rights sent. This field is ignored.

The sendmsg() call applies to sockets regardless of whether they are in the connected state and returns the length of the data sent.

If there is not enough buffer space available to hold the socket data to be transmitted, and the socket is in blocking mode, sendmsg() blocks the caller until more buffer space becomes available. If the socket is in nonblocking mode, sendmsg() returns a -1 and sets errno to EWOULDBLOCK. See [“fcntl\(\)” on page 103](#) or [“ioctl\(\)” on page 129](#) to set nonblocking mode.

For datagram sockets, this call sends the entire datagram, providing the datagram can fit into the TCP/IP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application wants to send 1000 bytes, each call to this function can send 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, and call this function until all data has been sent.

Return values

If successful, the length of the message in bytes is returned. The value -1 indicates an error. Errno identifies the specific error.

Errno

Description

EBADF

Indicates that *s* is not a valid socket descriptor.

EFAULT

Using *msg* would result in an attempt to access storage outside the caller address space.

EINVAL

Indicates that *msg_namelen* is not the size of a valid address for the specified address family.

EMSGSIZE

The message was too big to be sent as a single datagram.

ENOBUFS

Buffer space is not available to send the message.

EWOULDBLOCK

Indicates that *s* is in nonblocking mode and there is not enough space in TCP/IP to accept the data.

Related calls

`connect()`, `fcntl()`, `getsockopt()`, `ioctl()`, `read()`, `readv()`, `recv()`, `recvfrom()`, `recvmsg()`, `select()`, `selectex()`, `send()`, `sendto()`, `setsockopt()`, `socket()`, `write()`, `writex()`

sendto()

The `sendto()` call sends datagrams on the socket with descriptor *s*. The `sendto()` call applies to any datagram socket, whether connected or unconnected.

If there is not enough available buffer space to hold the socket data to be transmitted, and the socket is in blocking mode, `sendto()` blocks the caller until more buffer space becomes available. If the socket is in nonblocking mode, `sendto()` returns a -1 and sets `errno` to `EWOULDBLOCK`. See [“fcntl\(\)” on page 103](#) or [“ioctl\(\)” on page 129](#) to set nonblocking mode.

For datagram sockets, this call sends the entire datagram, providing the datagram can fit into the TCP/IP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application wants to send 1000 bytes, each call to this function can send 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, and call this function until all data has been sent.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>

int sendto(int s, char *msg, int len, int flags, struct sockaddr *to,
           int tolen)
```

Parameter**Description**

s

Socket descriptor.

msg

Points to the buffer containing the message to be transmitted.

len

Length of the message in the buffer pointed to by *msg*.

flags

A parameter that can be set to 0 or `MSG_DONTROUTE`. Setting this parameter is supported only for sockets in the `AF_INET` domain. Setting these flags is not supported in the `AF_IUCV` domain.

MSG_DONTROUTE

The `SO_DONTROUTE` option is turned on for the duration of the operation. This is usually used by diagnostic or routing programs only.

to

Address of the target.

tolen

Size of the structure pointed to by *to*.

Return values

If successful, the number of characters sent is returned. The value -1 indicates an error. Errno identifies the specific error.

No indication of failure to deliver is implied in the return value of this call when used with datagram sockets.

Errno**Description****EBADF**

Indicates that *s* is not a valid socket descriptor.

EFAULT

Using the *msg* and *len* parameters would result in an attempt to access storage outside the caller address space.

EINVAL

Tolen is not the size of a valid address for the specified address family.

EMSGSIZE

The message was too big to be sent as a single datagram. The default is large-envelope-size.

ENOBUFS

Buffer space is not available to send the message.

EWOULDBLOCK

Indicates that *s* is in nonblocking mode and there is not enough space in TCP/IP to accept the data.

Related calls

`read()`, `readv()`, `recv()`, `recvfrom()`, `recvmsg()`, `send()`, `select()`, `selectex()`, `sendmsg()`, `socket()` `write()`, `writenv()`

sethostent()

The `sethostent()` call opens and caches the local host table contents for `gethostent()` calls. The `sethostent()` call is available only when `RESOLVE_VIA_LOOKUP` is defined before `MANIFEST.H` is included. See [z/OS Communications Server: IP Configuration Guide](#) for information about using local host tables.

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

int sethostent(int stayopen)
```

Parameter**Description*****stayopen***

A nonzero flag value prevents the cached local host table contents from being freed after an `endhostent()`.

Return values

The value 0 indicates success. The value -1 indicates an error. Errno identifies the specific error, returning the errno value of the `fopen()` call.

Related calls

endhostent(), endnetent(), gethostbyaddr(), gethostbyname(), gethostent()

setibmopt()

The setibmopt() call chooses the TCP/IP image with which to connect. It is used in conjunction with getibmopt(), which returns the number of TCP/IP images installed on a given MVS system and their names, versions, and states. With this information, the caller can dynamically choose the TCP/IP image with which to connect through the setibmopt() call.

Note: Images from pre-V3R2 releases of TCP/IP for MVS are excluded. The setibmopt() call is not meaningful for pre-V3R2 releases.

The setibmopt() call is optional. If setibmopt is not used, the standard method for determining the connecting TCP/IP image is followed. If setibmopt is used, it must be issued before any other socket calls that establish the connection to TCP/IP.

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

int setibmopt(int cmd, struct ibm_tcpimage *buf)

struct ibm_tcpimage {
    unsigned short status;
    unsigned short version;
    char name[8];
}
```

Parameter Description

cmd

The command to perform. For TCP/IP V3R2 for MVS, IBMTCP_IMAGE is supported.

buf

The address of the buffer to be used.

Parameter *buf* is the address of the struct ibm_tcpimage buffer containing the name and version of the TCP/IP image to which the caller wants to connect. The name must be left-justified and padded with blanks. The TCP/IP name is always the PROC name, left-justified and padded with blanks. The TCP/IP version and status are ignored. The caller is responsible to specify *name* before issuing the call. If setibmopt is not one of the active TCP/IP supported images on the system, subsequent socket calls will fail. This call checks the validity of the contents of the *name* field in the structure pointed to by *buf*. It checks the validity by verifying that the TCP/IP name is in the list generated by a getibmopt () call. It does not check the *status* or *version* fields. This call sets the image of the connection to be created on another call.

Typically, the caller issues getibmopt() to verify the choice for the TCP/IP image. On successful return, the caller's choice will be honored when attempting the connection to TCP/IP.

Return values

A 0 indicates success; the value -1 indicates an error. Errno identifies the specific error.

Errno

Description

EOPNOTSUPP

This is not supported in this release of TCP/IP.

EALREADY

Your program is already connected to a TCP/IP image.

EFAULT

Using *buf* would result in an attempt to copy the address into a portion of the caller address space into which information cannot be written.

setibmssockopt()

Like `setsockopt()` call, the `setibmssockopt()` call sets the options associated with a socket in the `AF_INET` domain. This call is for options specific to the IBM implementation of sockets.

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

int setibmssockopt(int s, int level, int optname, char *optval, int optlen)
```

Parameter

Description

s

Socket descriptor.

level

Level for which the option is being set. Only `SOL_SOCKET` is supported.

optname

The name of a specified socket option.

optval

Points to option data.

optlen

The length of the option data.

`SO_IGNOREINCOMINGPUSH` is another option to consider. This option is meaningful only for stream sockets. This option is effective only for connections established through an offload box. If *optval* points to 1, the option is set. If *optval* points to 0, the option is off.

The `SO_IGNOREINCOMINGPUSH` option causes a receive call to return when one of the following situations occurs:

- The requested length is reached.
- The internal TCP/IP length is reached.
- The peer application closes the connection.

The amount of data returned for each call is maximized and the amount of CPU time consumed by your program and TCP/IP is reduced.

This option is not appropriate to your operation if your program depends on receiving data before the connection is closed. For example, this option is appropriate for an FTP data connection, but not for a Telnet connection.

Return values

The value 0 indicates success; the value -1 indicates an error. `Errno` identifies the specific error.

Errno

Description

EBADF

The *s* parameter is not a valid socket descriptor.

EFAULT

Using *optval* and *optlen* parameters would result in an attempt to access storage outside the caller address space.

ENOPROTOOPT

The optname parameter is unrecognized, or the level parameter is not SOL_SOCKET.

Example

```
#include <manifest.h>
#include <socket.h>
#include <tcperror.h>

{ struct ibm_bulkmode_struct bulkstr;
  int optlen, rc;

  optlen = sizeof(bulkstr);
  rc = getibmssockopt(s, SOL_SOCKET, , (char *), &bulkstr, &optlen);
  if (rc < 0) {
    tcperror("on getibmssockopt()");
    exit(1);
  }
  fprintf(stream, "%d byte buffer available for outbound queue.\n",
    bulkstr.b_max_send_queue_size_avail);

  bulkstr.b_max_send_queue_size=bulkstr.b_max_send_queue_size_avail;
  bulkstr.b_onoff = 1;
  bulkstr.b_teststor = 0;
  bulkstr.b_move_data = 1;
  bulkstr.b_max_receive_queue_size = 65536;
  rc = setibmssockopt(s, SOL_SOCKET, , (char *), &bulkstr, optlen);
  if (rc < 0) {
    tcperror("on setibmssockopt()");
    exit(1);
  }
}
```

Related calls

getibmssockopt(), getsockopt(), ibmsflush(), setsockopt()

setnetent()

The setnetent() call opens and caches the local host table contents for getnetent() call. The setnetent() call is available only when RESOLVE_VIA_LOOKUP is defined before MANIFEST.H is included. See [z/OS Communications Server: IP Configuration Guide](#) for information about using local host tables.

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

int setnetent(int stayopen)
```

Parameter

Description

stayopen

A nonzero flag value prevents the cached local host table contents from being freed after an endnetent().

Return values

The value 0 indicates success; the value -1 indicates an error. Errno identifies the specific error, returning the errno value of the fopen() call.

Related calls

endnetent(), endhostent(), getnetbyaddr(), getnetbyname(), getnetent()

setprotoent()

The `setprotoent()` call opens the `hlq.ETC.PROTO` data set and sets it to the data set starting point. If the `stayopen` flag is nonzero, the `hlq.ETC.PROTO` data set remains open after every call.

Note: The `hlq.ETC.PROTO` data set is described in the [z/OS Communications Server: IP Configuration Reference](#).

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

int setprotoent(int stayopen)
```

Parameter	Description
-----------	-------------

<i>stayopen</i>	A flag that can be set to prevent data set <code>hlq.ETC.PROTO</code> closing after every call to <code>setprotoent()</code> .
------------------------	--

Return values

The value 0 indicates success; the value -1 indicates an error. Errno identifies the specific error, returning the errno value of the `fopen()` call.

Related calls

`endprotoent()`, `getprotobyname()`, `getprotobynumber()`, `getprotoent()`

setservent()

The `setservent()` call opens the `hlq.ETC.SERVICES` data set and resets it to its starting point. If the `stayopen` flag is nonzero, the `hlq.ETC.SERVICES` data set remains open after every call.

Note: The `hlq.ETC.SERVICES` data set is described in the [z/OS Communications Server: IP Configuration Reference](#).

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

int setservent(int stayopen)
```

Parameter	Description
-----------	-------------

<i>stayopen</i>	A flag that can be set to prevent data set <code>hlq.ETC.SERVICES</code> closing after each call to <code>setservent()</code> .
------------------------	---

Return values

The value 0 indicates success; the value -1 indicates an error. Errno identifies the specific error, returning the errno value of the `fopen()` call.

Related calls

`endservent()`, `getservbyname()`, `getservent()`

setsockopt()

The `setsockopt()` call sets options associated with a socket. It can be called only for sockets in the `AF_INET` domain. Options can exist at multiple protocol levels; they are always present at the highest socket level.

When manipulating socket options, you must specify the level at which the option resides and the name of the option. To manipulate options at the socket level, the *level* parameter must be set to `SOL_SOCKET`, as defined in `SOCKET.H`. To manipulate options at the TCP level, the *level* parameter must be set to `IPPROTO_TCP`, as defined in `SOCKET.H`. To manipulate options at any other level, such as the IP level, supply the appropriate protocol number for the protocol controlling the option. Currently, the `SOL_SOCKET`, `IPPROTO_TCP`, and `IPPROTO_IP` levels are supported. The `getprotobyname()` call can be used to return the protocol number for a named protocol.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>

int setsockopt(int s, int level, int optname, char *optval, int optlen)
```

Parameter Description

s

The socket descriptor

level

The level for which the option is being set

optname

The name of a specified socket option. See Appendix D, “GETSOCKOPT/SETSOCKOPT command values,” on page 767 for the numeric values of *optname*.

optval

The pointer to option data

optlen

The length of the option data

The *optval* and *optlen* parameters are used to pass data used by the particular set command. The *optval* parameter points to a buffer containing the data needed by the set command. The *optlen* parameter must be set to the size of the data pointed to by *optval*.

All of the socket level options except `SO_LINGER` expect *optval* to point to an integer and *optlen* to be set to the size of an integer. When the integer is nonzero, the option is enabled. For toggle type options, if the integer is nonzero, the option is enabled. If it is 0, the option is disabled. The `SO_LINGER` option expects *optval* to point to a *linger* structure, as defined in `SOCKET.H`. This structure is defined in the following example:

```
struct linger
{
    int    l_onoff;           /* option on/off */
    int    l_linger;         /* linger time */
};
```

The *l_onoff* field is set to 0 if the `SO_LINGER` option is disabled. A nonzero value enables the option. The *l_linger* field specifies the amount of time to wait on close. The units of *l_linger* are seconds.

The following option is recognized at the TCP level:

Option Description

TCP_NODELAY

Toggles the use of Nagle algorithm (RFC 896) for all data sent over the socket. This option is not supported for `AF_IUCV` sockets. Under most circumstances, TCP sends data when it is presented from the application.

However, when outstanding data has not yet been acknowledged, TCP will defer the transmission of any new data from the application until all of the outstanding data has been acknowledged. The Nagle

algorithm enforces this deferral, even in cases where the receiver's window is sufficiently open to accept the new data. For interactive applications, such as ones that send a stream of mouse events which receive no replies, this deferral of transmission might result in significant delays. For these types of applications, disabling Nagle algorithm would improve response time.

Notes:

1. When Nagle algorithm is enabled, TCP will wait to send small amounts of data until the acknowledgment for the previous data is received.
2. When Nagle algorithm is disabled, TCP will send small amounts of data even before the acknowledgment for previous data sent is received.

The following keywords are recognized at the socket level:

Keyword

Description

SO_RCVBUF

Sets the size of the data portion of the TCP/IP receive buffer in OPTVAL. The size of the data portion of the receive buffer is protocol-specific. If the requested size exceeds the allowed size, the following situation occurs:

- If the protocol is TCP, a return value of -1 and errno of ENOBUFS is set. The receive buffer size is unchanged.

For maximum values for the TCP protocol, see the TCPCONFIG TCPCVBUFRSIZE and TCPMAXRCVBUFSIZE parameters in the [z/OS Communications Server: IP Configuration Reference](#).

- If the protocol is UDP or RAW, a return value of 0 is returned and the buffer size is set to 65535.

SO_SNDBUF

Sets the size of the data portion of the TCP/IP send buffer in OPTVAL. The size of the data portion of the send buffer is protocol-specific. If the requested size exceeds the allowed size, the following situation occurs:

- If the protocol is TCP, a return value of -1 and errno of ENOBUFS is set. The send buffer size for the TCP connection is set to the maximum size. The value of the send buffer size can be retrieved by issuing GETSOCKOPT for SO_SNDBUF.

For maximum values for the TCP protocol, see the TCPCONFIG TCPSENDBUFRSIZE parameters in the [z/OS Communications Server: IP Configuration Reference](#).

- If the protocol is UDP or RAW, a return value of 0 is returned and the buffer size is set to 65535.

SO_BROADCAST

Toggles the ability to broadcast messages. The default is *disabled*. If this option is enabled, it allows the application to send broadcast messages over *s* when the interface specified in the destination supports broadcasting of packets. This option has no meaning for stream sockets.

SO_KEEPAIVE

Toggles the TCP keep alive mechanism for a stream socket. The default is *disabled*. When activated, the keep alive mechanism periodically sends a packet on an otherwise idle connection. If the remote TCP does not respond to the packet, or to retransmissions of the packet, the connection is ended with the error ETIMEDOUT.

SO_LINGER

Lingers on close if data is present. The default is *disabled*. When this option is enabled and there is unsent data present when close() is called, the calling application is blocked during the close() call until the data is transmitted, or the connection has timed out. If this option is disabled, the close() call returns without blocking the caller, and the TCP/IP address space still waits to try to send the data. Although the data transfer is usually successful, it cannot be guaranteed, because the TCP/IP address space waits a finite amount of time while trying to send the data. This option has meaning for stream sockets only.

SO_OOBINLINE

Toggles the reception of out-of-band data. The default is *disabled*. When this option is enabled, it causes out-of-band data to be placed in the normal data input queue as it is received, making it available to `recv()`, `recvfrom()`, and `recvmsg()` without having to specify the `MSG_OOB` flag in those calls. When this option is disabled, it causes out-of-band data to be placed in the priority data input queue as it is received, making it available to `recv()`, `recvfrom()`, and `recvmsg()` only by specifying the `MSG_OOB` flag in those calls. This option has meaning for stream sockets only.

SO_REUSEADDR

Toggles local address reuse. The default is disabled. This alters the normal algorithm used in the `bind()` call.

The normal `bind()` call algorithm allows each internet address and port combination to be bound only once. If the address and port have been bound already, a subsequent `bind()` will fail and return error `EADDRINUSE`.

After the 'SO_REUSEADDR' option is active, the following situations are supported:

- A server can `bind()` the same port multiple times as long as every invocation uses a different local IP address and the wildcard address `INADDR_ANY` is used only one time per port.
- A server with active client connections can be restarted and can bind to its port without having to close all of the client connections.
- For datagram sockets, multicasting is supported so multiple `bind()` calls can be made to the same class D address and port number.

The following options are recognized at the IP level (`IPPROTO_IP`):

Option

Description

IP_MULTICAST_TTL

Sets the IP time to live of outgoing multicast datagrams. The default value is 1 (multicast is available only to the local subnet).

IP_MULTICAST_LOOP

Enables or disables the loopback of outgoing multicast datagrams. The default value is enabled.

IP_MULTICAST_IF

Sets the interface for sending outbound multicast datagrams from the socket application.

Note: Multicast datagrams can be transmitted only on one interface at a time.

IP_ADD_MEMBERSHIP

Joins a multicast group on a specific interface. An interface has to be specified with this option. Only applications that want to receive multicast datagrams need to join multicast groups.

IP_DROP_MEMBERSHIP

Exits a multicast group.

Return values

The value 0 indicates success; the value -1 indicates an error. `Errno` identifies the specific error.

Errno

Description

EBADF

The `s` parameter is not a valid socket descriptor.

EFAULT

Using `optval` and `optlen` parameters would result in an attempt to access storage outside the caller address space.

ENOBUFS

No buffer space is available.

ENOPROTOOPT

The *optname* parameter is unrecognized, or the *level* parameter is not SOL_SOCKET.

Example

See “[getsockopt\(\)](#)” on page 119 to see how the getsockopt() options set is queried.

```
int rc;
int s;
int optval;
struct linger l;
int setsockopt(int s, int level, int optname, char *optval, int optlen);
:
/* I want out of band data in the normal inputqueue */
optval = 1;
rc = setsockopt(s, SOL_SOCKET, SO_OOBINLINE, (char *) &optval, sizeof(int));
:
/* I want to linger on close */
l.l_onoff = 1;
l.l_linger = 100;
rc = setsockopt(s, SOL_SOCKET, SO_LINGER, (char *) &l, sizeof(l));
```

Related calls

fcntl(), getprotobyname(), getsockopt(), ioctl(), socket()

shutdown()

The shutdown() call shuts down all or part of a duplex connection. Parameter *how* sets the condition for shutdown to the socket *s* connection.

If you issue a shutdown() for a socket that currently has outstanding socket calls pending, see [Table 3](#) on page 32 to determine the effects of this operation on the outstanding socket calls.

Note: Issue a shutdown() call before issuing a close() call for a socket.

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

int shutdown(int s, int how)
```

Parameter

Description

s

The socket descriptor.

how

The *how* condition can have a value of 0, 1, or 2, where:

- Zero ends further receive operations.
- One ends further send operations.
- Two ends further send and receive operations.

Return values

The value 0 indicates success; the value -1 indicates an error. Errno identifies the specific error.

Errno

Description

EBADF

Indicates that *s* is not a valid socket descriptor.

EINVAL

The *how* parameter was not set to a valid value: 0, 1, or 2.

sock_debug()

The `sock_debug()` call provides the socket library tracing facility. The *onoff* parameter can have a value of 0 or nonzero. If *onoff*=0 (the default), no socket library tracing is done. If *onoff*=nonzero, the system traces for socket library calls and interrupts.

Note: You can include the statement `SOCKDEBUG` in data set `TCPIP.DATA` as an alternative to calling `sock_debug()` with *onoff* not equal to 0.

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

void sock_debug(init onoff)
```

Parameter

Description

onoff

A parameter that can be set to 0 or nonzero

Related calls

`accept()`, `close()`, `connect()`, `socket()`

sock_do_teststor()

The `sock_do_teststor()` call is used to check for calls that attempt to access storage outside the caller address space.

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

void sock_do_teststor(int onoff)
```

Parameter

Description

onoff

A parameter that can be set to 0 or nonzero

If *onoff* is not 0 for either inbound or outbound sockets, both the address of the message buffer and the message buffer itself are checked for addressability at every socket call. The error condition, `EFAULT`, is set if there is an addressing problem. If *onoff* is set to 0, address checking is not done by the socket library program. If an error occurs when *onoff* is 0, normal runtime error handling reports the exception condition.

The default for *onoff* is 0. Addresses are not checked for addressability for parameters of C socket calls. While you are testing your program, you might find it useful to set *onoff* to a nonzero value.

Notes:

1. You can include the statement `SOCKNOTESTSTOR` in data set `TCPIP.DATA`, as an alternative to calling `sock_do_teststor()` with *onoff* equal to 0.
2. You can include the statement `SOCKTESTSTOR` in the data set `TCPIP.DATA` which is in the client's catalog when the socket program is started, as an alternative to calling `sock_do_teststor()` with *onoff* not equal to 0.

Restrictions

None

socket()

The `socket()` call creates an endpoint for communication and returns a socket descriptor representing that endpoint. Different types of sockets provide different communication services.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>

int socket(int domain, int type, int protocol)
```

Parameter Description

domain

The address domain requested. It is either `AF_INET` or `AF_IUCV`.

type

The type of socket created, either `SOCK_STREAM`, `SOCK_DGRAM`, or `SOCK_RAW`.

protocol

The protocol requested. Possible values are 0, `IPPROTO_UDP`, or `IPPROTO_TCP`.

The *domain* parameter specifies the communication domain within which communication is to take place. This parameter specifies the address family (format of addresses within a domain) to be used. The families supported are `AF_INET`, which is the internet domain, and `AF_IUCV`, which is the IUCV domain. These constants are defined in the `SOCKET.H` header file.

The *type* parameter specifies the type of socket created. The type is analogous to the communication requested. These socket type constants are defined in the `SOCKET.H` header file. The types supported are:

Socket Type Description

SOCK_STREAM

Provides sequenced, two-way byte streams that are reliable and connection-oriented. They support a mechanism for out-of-band data. This type is supported in both the `AF_INET` and `AF_IUCV` domains.

SOCK_DGRAM

Provides datagrams, which are connectionless messages, of a fixed maximum length whose reliability is not guaranteed. Datagrams can be corrupted, received out of order, lost, or delivered repeatedly. This type is supported in the `AF_INET` domain only.

SOCK_RAW

Provides the interface to internal protocols (such as IP and ICMP). This type is supported in the `AF_INET` domain only.

Note: To use raw sockets, the application must be APF-authorized.

The *protocol* parameter specifies the particular protocol to be used with the socket. In most cases, a single protocol exists to support a particular type of socket within a particular addressing family (not true with raw sockets). If the *protocol* parameter is set to 0, the system selects the default protocol number for the domain and socket type requested. Protocol numbers are found in the *hlq.ETC.PROTO* data set. Alternatively, the `getprotobyname()` call can be used to get the protocol number for a protocol with a known name. The *protocol* field must be set to 0 if the *domain* parameter is set to `AF_IUCV`. The *protocol* defaults are TCP for stream sockets and UDP for datagram sockets. There is no default for raw sockets.

`SOCK_STREAM` sockets model duplex byte streams. They provide reliable, flow-controlled connections between peer applications. Stream sockets are either active or passive. Active sockets are used by clients who initiate connection requests using `connect()`. By default, `socket()` creates active sockets. Passive

sockets are used by servers to accept connection requests from the `connect()` call. An active socket is transformed into a passive socket by binding a name to the socket using the `bind()` call, and by indicating a willingness to accept connections with the `listen()` call. If a socket is passive, it cannot be used to initiate connection requests.

In the `AF_INET` domain, the `bind()` call applied to a stream socket lets the application specify the networks from which it will accept connection requests. The application can fully specify the network interface by setting the *internet address* field in the *address* structure to the internet address of a network interface. Alternatively, the application can use a *wildcard* to specify that it wants to receive connection requests from any network. This is done by setting the *internet address* field within the *address* structure to the constant `INADDR_ANY`, as defined in the `SOCKET.H` header file.

After a connection has been established between stream sockets, any of the data transfer calls can be used: `read()`, `write()`, `send()`, `recv()`, `readv()`, `writew()`, `sendto()`, `recvfrom()`, `sendmsg()`, and `recvmsg()`. Usually, the read-write or send-recv pairs are used to send data on stream sockets. If out-of-band data is to be exchanged, the send-recv pair is normally used.

`SOCK_DGRAM` sockets model datagrams. They provide connectionless message-exchange without guarantee of reliability. Messages sent are limited in size. Datagram sockets are not supported in the `AF_IUCV` domain.

There is no active or passive analogy to stream sockets with datagram sockets. Servers must still call `bind()` to name a socket and to specify the network interface from which it wants to receive packets. Wildcard addressing, as described for stream sockets, applies to datagram sockets also. Because datagram sockets are connectionless, the `listen()` call has no meaning for them and must not be used with them.

After an application has received a datagram socket, it can exchange datagrams using the `sendto()` and `recvfrom()`, or `sendmsg()` and `recvmsg()` calls. If the application goes one step further by calling `connect()` and fully specifying the name of the peer with which all messages are to be exchanged, then the other data transfer calls of `read()`, `write()`, `readv()`, `writew()`, `send()`, and `recv()` can be used also. See [“connect\(\)” on page 99](#) for more information about placing a socket into the connected state.

Datagram sockets allow messages to be broadcast to multiple recipients. Setting the destination address to a broadcast address depends on the class of address, and whether subnets are used. The constant `INADDR_BROADCAST`, defined in `socket.h`, can be used to broadcast to the primary network when the primary network configured supports broadcast.

`SOCK_RAW` sockets give the application an interface to lower layer protocols, such as IP and ICMP. This interface is often used to bypass the transport layer when direct access to lower layer protocols is needed. Raw sockets are also used to test new protocols. Raw sockets are not supported in the `AF_IUCV` domain.

Raw sockets are connectionless and data transfer semantics are the same as those described previously for datagram sockets. The `connect()` call can be used similarly to specify the peer.

Outgoing packets have an IP header prefixed to them. IP options can be set and inspected using the `setsockopt()` and `getsockopt()` calls respectively. Incoming packets are received with the IP header and options intact.

Notes:

1. Sockets are deallocated using the `close()` call.
2. Only `SOCK_STREAM` sockets are supported in the `AF_IUCV` domain.
3. The `setsockopt()` and `getsockopt()` calls are not supported for sockets in the `AF_IUCV` domain.
4. The flags field in the `send()`, `recv()`, `sendto()`, `recvfrom()`, `sendmsg()`, and `recvmsg()` calls is not supported in the `AF_IUCV` domain.

Return values

A nonnegative socket descriptor indicates success. The value -1 indicates an error. `Errno` identifies the specific error.

Errno**Description****EPROTONOSUPPORT**

The *protocol* is not supported in this *domain* or this socket *type*.

EACCES

Access denied. The application is not an APF-authorized application.

EAFNOSUPPORT

The specified address family is not supported by this protocol family.

Example

```
int s;
struct protoent *p;
struct protoent *getprotobyname(char *name);
int socket(int domain, int type, int protocol);
...
/* Get stream socket in internetdomain with default protocol */
s = socket(AF_INET, SOCK_STREAM, 0);
...
/* Get stream socket in iucvdomain with default protocol */
s = socket(AF_IUCV, SOCK_STREAM, 0);
...
/* Get raw socket in internetdomain for ICMP protocol */
p = getprotobyname("iucv");
s = socket(AF_INET, SOCK_RAW, p->p_proto);
```

Related calls

accept(), bind(), close(), connect(), fcntl(), getprotobyname(), getsockname(), getsockopt(), ioctl(), maxdesc(), read(), readv(), recv(), recvfrom(), recvmsg(), select(), selectex(), send(), sendmsg(), sendto(), shutdown(), write(), writev()

takesocket()

The takesocket() call acquires a socket from another program. Typically, the other program passes its client ID and socket descriptor to your program through your program startup parameter list.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>

int takesocket(struct clientid *clientid, int hisdesc)
```

Parameter**Description*****clientid***

Points to the *clientid* of the application from which you are taking a socket.

hisdesc

Describes the socket to be taken.

The takesocket() call acquires a socket from another program. Typically, the other program passes its client ID and socket descriptor to your program through your program startup parameter list.

Return values

A nonnegative socket descriptor indicates success. The value -1 indicates an error. Errno identifies a specific error.

Errno**Description**

EACCES

The other application did not give the socket to your application.

EBADF

The *hisdesc* parameter does not specify a valid socket descriptor owned by the other application. The socket has already been taken.

EFAULT

Using the *clientid* parameter as specified would result in an attempt to access storage outside the caller address space.

EINVAL

The *clientid* parameter does not specify a valid client identifier.

EMFILE

The socket descriptor table is already full.

ENOBUFS

The operation cannot be performed because of a shortage of control blocks (SCB or SKCB) in the TCP/IP address space.

EPFNOSUPPORT

The domain field of the *clientid* parameter is not AF_INET.

Related calls

getclientid(), givesocket()

tcperror()

When a socket call produces an error, the call returns a negative value and the variable *errno* is set to an error value found in TCPERRNO.H. The `tcperror()` call prints a short error message describing the last error that occurred. If *s* is non-NULL, `tcperror()` prints the string *s* followed by a colon, followed by a space, followed by the error message, and terminating with a new-line character. If *s* is NULL or points to a NULL string, only the error message and the new-line character are output.

The `tcperror()` function is equivalent to the UNIX `error()` function.

```
#include <manifest.h>
#include <socket.h>
#include <tcperrno.h>

void tcperror(char *s)
```

Parameter**Description****s**

A NULL or NULL-terminated character string

Example 1

```
if ((s=socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
    tcperror("socket()");
    exit(2);
}
```

If the `socket()` call produces error ENOMEM, `socket()` returns a negative value and *errno* is set to ENOMEM. When `tcperror()` is called, it prints the string:

```
socket(): Not enough storage (ENOMEM)
```

Example 2

```
if ((s=socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    tcperror(NULL);
```

If the `socket()` call produces error `ENOMEM`, `socket()` returns a negative value and `errno` is set to `ENOMEM`. When `tcperror()` is called, it prints the string:

```
Not enough storage (ENOMEM)
```

write()

The `write()` call writes data from a buffer on a socket with descriptor `s`. The `write()` call applies only to connected sockets.

This call writes up to `len` bytes of data.

If there is not enough available buffer space to hold the socket data to be transmitted and the socket is in blocking mode, `write()` blocks the caller until more buffer space is available. If the socket is in nonblocking mode, `write()` returns a -1 and sets `errno` to `EWOULDBLOCK`. See [“fcntl\(\)” on page 103](#) or [“ioctl\(\)” on page 129](#) to set nonblocking mode.

For datagram sockets, this call sends the entire datagram, providing the datagram can fit into the TCP/IP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application wants to send 1000 bytes, each call to this function can send 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, and call this function until all data has been sent.

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

int write(int s, char *buf, int len)
```

Parameter Description

- s**
Socket descriptor.
- buf**
Points to the buffer holding the data to be written.
- len**
Length in bytes of *buf*.

Return values

If successful, the number of bytes written is returned. The value -1 indicates an error. `Errno` identifies the specific error.

Errno Description

EBADF
Indicates that `s` is not a valid socket descriptor.

EFAULT
Using the *buf* and *len* parameters would result in an attempt to access storage outside the caller address space.

ENOBUFS
Buffer space is not available to send the message.

EWOULDBLOCK
Indicates that `s` is in nonblocking mode and there is not enough space in TCP/IP to accept the data.

Related calls

connect(), fcntl(), getsockopt(), ioctl(), read(), readv(), recv(), recvfrom(), recvmsg(), select(), selectex(), send(), sendmsg(), sendto(), setsockopt(), socket(), writev()

writev()

The writev() call writes data from a set of buffers on a socket using descriptor *s*.

The writev() call applies only to connected sockets.

```
#include <manifest.h>
#include <socket.h>
#include <bsdtypes.h>
#include <uio.h>

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

Parameter	Description
-----------	-------------

<i>s</i>	Socket descriptor.
-----------------	--------------------

<i>iov</i>	Points to an array of iovec buffers.
-------------------	--------------------------------------

<i>iovcnt</i>	Number of buffers in the array.
----------------------	---------------------------------

The data is gathered from the buffers specified by *iov*[0]...*iov*[*iovcnt*-1]. The *iovec* structure is defined in UIO.H and contains the following fields:

Parameter	Description
-----------	-------------

<i>iov_base</i>	Points to the buffer.
------------------------	-----------------------

<i>iov_len</i>	The length of the buffer.
-----------------------	---------------------------

This call writes the sum of the *iov_len* bytes of data.

If buffer space is not available to hold the socket data to be transmitted and the socket is in blocking mode, writev() blocks the caller until additional buffer space becomes available. If the socket is in a nonblocking mode, writev() returns a -1 and sets *errno* to EWOULDBLOCK. For a description of how to set nonblocking mode, see [“fcntl\(\)” on page 103](#) or [“ioctl\(\)” on page 129](#).

For datagram sockets, this call sends the entire datagram, providing the datagram can fit into the TCP/IP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application wants to send 1000 bytes, each call to this function can send 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, calling this function until all data has been sent.

Return values

If successful, the number of bytes written from the buffers is returned. The value -1 indicates an error. Errno identifies the specific error.

Errno	Description
-------	-------------

EBADF	Indicates that <i>s</i> is not a valid socket descriptor.
--------------	---

EFAULT

Using the *iov* and *iovcnt* parameters would result in an attempt to access storage outside the caller address space.

ENOBUFS

Buffer space is not available to send the message.

EWOULDBLOCK

Indicates that *s* is in nonblocking mode and there is not enough space in TCP/IP to accept the data.

Related calls

connect(), fcntl(), getsockopt(), ioctl(), write(), read(), readv(), recv(), recvmsg(), recvfrom(), select(), selectex(), send(), sendmsg(), sendto(), setsockopt(), socket(), write()

Sample C socket programs

This topic contains sample C socket programs. The C source code can be found in the SEZAINST data set.

Following are the sample socket programs available:

Program	Description
TCPC	C socket TCP client
TCPS	C socket TCP server
UDPC	C socket UDP client
UDPS	C socket UDP server

For samples of the multitasking C programs in the following table, see [Appendix A, “Multitasking C socket sample program,”](#) on page 733.

Program	Description
MTCLNT	C socket MTC client
MTCSVR	C socket MTC server
MTCCSUB	C socket subtask MTCCSUB

Executing TCPS and TCPC modules

To start the TCPS server, execute TCPS 9999 on the other MVS address space (server).

To run the TCPC client, execute TCPC MVS13 9999. (MVS13 is the host name where the TCPS server is running, and 9999 is the port you have assigned.)

After executing the TCPC client, the following output is displayed on the server session:

```
Server Ended Successfully
```

Executing UDPS and UDPC modules

To start the UDPS server, execute UDPS on the other MVS address space (server). The following message is displayed:

```
Port assigned is 1028
```

To run the UDPC client, execute UDPC 9.67.60.10 1028. (Address 9.67.60.10 is the IP machine address where the UDPS server is running, and 1028 is the port assigned by the UDPS server.)

After executing the UDPC client, the following message is displayed:

```
Received Message Hello....
```


C socket TCP client

The following example shows a C socket TCP client (TGPC) program. The source code can be found in the TGPC member of the SEZAINST data set.

```
/** IBMCPYR *****/
/*
/* Part Name: TGPC
/*
/*
/* Copyright:    Licensed Materials - Property of IBM
/*
/*              "Restricted Materials of IBM"
/*
/*              5647-A01
/*
/*              (C) Copyright IBM Corp. 1977, 1998
/*
/*              US Government Users Restricted Rights -
/*              Use, duplication or disclosure restricted by
/*              GSA ADP Schedule Contract with IBM Corp.
/*
/* Status:      CSV2R6
/*
/* SMP/E Distribution Name: EZAEC01V
/*
/** IBMCPYR *****/

static char ibmcopyr[] =
    "TGPC    - Licensed Materials - Property of IBM. "
    "This module is \"Restricted Materials of IBM\" "
    "5647-A01 (C) Copyright IBM Corp. 1996. "
    "See IBM Copyright Instructions.";

/*
 * Include Files.
 */
#include <manifest.h>
#include <bsdtypes.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 & 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(2);
    }

    /*
     * 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)
    {
        tcperror("Socket()");
        exit(3);
    }

    /*
     * Connect to the server.
     */
    if (connect(s, (struct sockaddr *)&server, sizeof(server)) < 0)
    {
        tcperror("Connect()");
        exit(4);
    }

    if (send(s, buf, sizeof(buf), 0) < 0)
    {
        tcperror("Send()");
        exit(5);
    }

    /*
     * The server sends back the same message. Receive it into the
     * buffer.
     */
    if (recv(s, buf, sizeof(buf), 0) < 0)
    {
        tcperror("Recv()");
        exit(6);
    }

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

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

```

Figure 52. C socket TCP client sample

C socket TCP server

The following example shows a C socket TCP server (TCPS) program. The source code can be found in the TCPS member of the SEZAINST data set.

```

/**** IBMCOPYR *****/
/*
/* Component Name: TCPS
/*
/*
/* Copyright:      Licensed Materials - Property of IBM
/*
/*
/* "Restricted Materials of IBM"
/*

```

```

/*
/*          5647-A01
/*
/*          (C) Copyright IBM Corp. 1977, 1998
/*
/*          US Government Users Restricted Rights -
/*          Use, duplication or disclosure restricted by
/*          GSA ADP Schedule Contract with IBM Corp.
/*
/* Status:      CSV2R6
/*
/* SMP/E Distribution Name: EZAEC01X
/*
/*
/**** IBMCOPYR *****/

static char ibmcopyr[] =
    "TCPS - Licensed Materials - Property of IBM. "
    "This module is \"Restricted Materials of IBM\" "
    "5647-A01 (C) Copyright IBM Corp. 1996. "
    "See IBM Copyright Instructions.";

#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>
#include <netdb.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 & 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)
    {
        tcperror("Socket()");
        exit(2);
    }

    /*
     * 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, (struct sockaddr *)&server, sizeof(server)) < 0)
    {
        tcperror("Bind()");
        exit(3);
    }

    /*

```

```

    /* Listen for connections. Specify the backlog as 1.
    */
    if (listen(s, 1) != 0)
    {
        tcperror("Listen()");
        exit(4);
    }

    /*
    * Accept a connection.
    */
    namelen = sizeof(client);
    if ((ns = accept(s, (struct sockaddr *)&client, &namelen)) == -1)
    {
        tcperror("Accept()");
        exit(5);
    }

    /*
    * Receive the message on the newly connected socket.
    */
    if (recv(ns, buf, sizeof(buf), 0) == -1)
    {
        tcperror("Recv()");
        exit(6);
    }

    /*
    * Send the message back to the client.
    */
    if (send(ns, buf, sizeof(buf), 0) < 0)
    {
        tcperror("Send()");
        exit(7);
    }

    close(ns);
    close(s);

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

```

Figure 53. C socket TCP server sample

C socket UDP server

The following example shows a C socket UDP server (UDPS) program. The source code can be found in the UDPS member of the SEZAINST data set.

```

/**** IBMCOPYR *****/
/*
/* Component Name: UDPS
/*
/*
/* Copyright:    Licensed Materials - Property of IBM
/*
/*              "Restricted Materials of IBM"
/*
/*              5647-A01
/*
/*              (C) Copyright IBM Corp. 1977, 1998
/*
/*              US Government Users Restricted Rights -
/*              Use, duplication or disclosure restricted by
/*              GSA ADP Schedule Contract with IBM Corp.
/*
/* Status:      CSV2R6
/*
/* SMP/E Distribution Name: EZAEC021
/*
/**** IBMCOPYR *****/

static char ibmcopyr[] =
    "UDPS    - Licensed Materials - Property of IBM. "

```

```

"This module is \"Restricted Materials of IBM\" "
"5647-A01 (C) Copyright IBM Corp. 1992, 1996. "
"See IBM Copyright Instructions.";

#include <manifest.h>
#include <bsdtypes.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)
    {
        tcperror("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, (struct sockaddr *)&server, sizeof(server)) < 0)
    {
        tcperror("bind()");
        exit(2);
    }

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

```

Figure 54. C socket UDP server sample

C socket UDP client

The following example shows a C socket UDP (UDPC) client program. The source code can be found in the UDPC member of the SEZAINST data set.

```

/**** IBMCOPYR *****/
/*
/* Component Name: UDPC
/*
/*
/* Copyright:    Licensed Materials - Property of IBM
/*
/*              "Restricted Materials of IBM"
/*
/*              5647-A01
/*
/*              (C) Copyright IBM Corp. 1977, 1998
/*
/*              US Government Users Restricted Rights -
/*              Use, duplication or disclosure restricted by
/*              GSA ADP Schedule Contract with IBM Corp.
/*
/* Status:      CSV2R6
/*
/* SMP/E Distribution Name: EZAEC020
/*
/**** IBMCOPYR *****/

static char ibmcopyr[] =
    "UPDC - Licensed Materials - Property of IBM. "
    "This module is \"Restricted Materials of IBM\" "
    "5647-A01 (C) Copyright IBM Corp. 1992, 1996. "
    "See IBM Copyright Instructions.";

#include <manifest.h>
#include <bsdtypes.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)
{
    tcperror("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,
           (struct sockaddr *)&server, sizeof(server)) < 0)
{
    tcperror("sendto()");
    exit(2);
}

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

```

Figure 55. C socket UDP client sample

Chapter 11. X/Open Transport Interface

This topic describes the X/Open Transport Interface (XTI) IPv4 socket application program interface (API) and contains the following topics:

- Software requirements
- What is provided
- How XTI works in the z/OS environment
- Creating an application
- Coding XTI calls
- Compiling and linking XTI applications using cataloged procedures
- Understanding XTI sample programs

The XTI allows you to write applications in the z/OS environment to access the open transport interface.

Note: The XTI calls in this topic apply only to unconnected sessions.

For more information about the XTI protocol, see *CAE Specification: X/Open Transport Interface (XTI)*.

XTI software requirements

Application programs using the X/Open Transport Interface (XTI) require:

- SEZACMAC (macro library routines)
- SEZACMTX (executable modules)
- SEZALOAD (executable modules)
- SEZAINST (sample programs)
- Current z/OS Language Environment run-time library

What is provided with XTI

The XTI support provided with TCP/IP includes:

- The XTI library containing the XTI calls for C language programmers
- The XTI management services that allow you to include additional protocol mappers
- The RFC 1006 protocol mapping component that creates the protocol expected by the XTI interface

For more information about RFC1006, see [Appendix G, “Related protocol specifications,” on page 779](#).

How XTI works in the z/OS environment

The XTI is a network-transparent protocol. In the z/OS environment, XTI system support is a set of application calls to create the XTI protocol, as requested by your application. The services request is communicated to the XTI transport system using the RFC 1006 protocol mapper. RFC 1006 translates messages to transport class 0 service requests before passing them to the XTI.

[Figure 56 on page 174](#) is a high-level diagram to show how the XTI interface works in an z/OS environment.

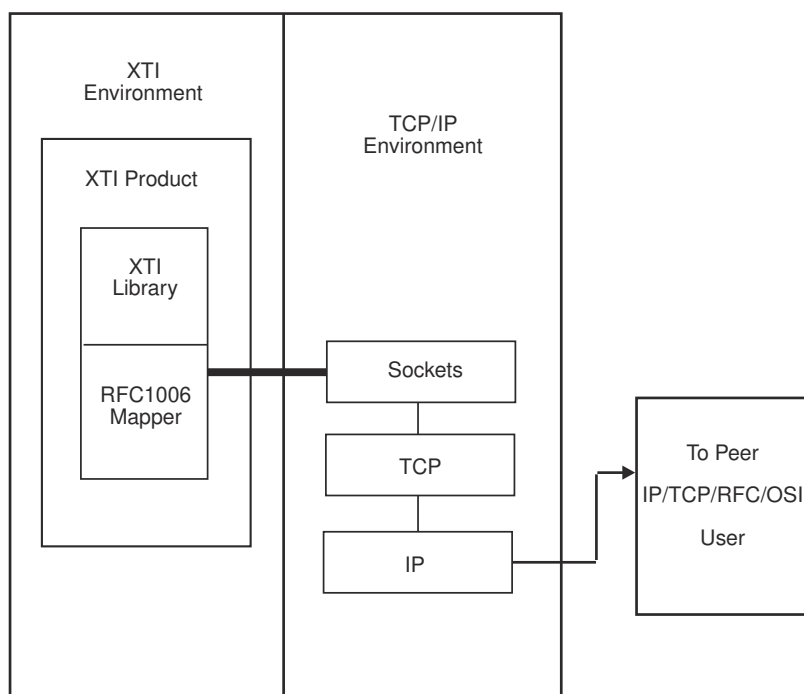


Figure 56. Using XTI with TCP/IP

In the z/OS environment, external names must be eight characters or fewer. If the XTI application program interface names exceed this limit, those names longer than eight characters are remapped to new names using the C compiler preprocessor. This name remapping is found in a file called X11GLUE.H, which is automatically included in your program when you include the header file called XLIB.H. When debugging your application, you can refer to the X11GLUE.H file to find the remapped names of the XTI programs.

Creating an application using the XTI protocol

To create an application that uses the XTI protocol, you should study the XTI application program interface in *CAE Specification: X/Open Transport Interface (XTI)*. In addition, both [“XTI socket client sample program”](#) on page 181, and [“XTI socket server sample program”](#) on page 186 illustrate programs that use the XTI interface. These programs are distributed with TCP/IP.

Coding XTI calls

The following tables list the call instructions supported by the XTI for TCP/IP. These call instructions are for unconnected sessions only, and are listed by type of service.

Coding XTI calls: Initializing a transport endpoint

Table 7 on page 174 lists the routines needed to initialize a transport endpoint. For more information, see *CAE Specification: X/Open Transport Interface (XTI)*.

Table 7. Initializing a call

Call	Description
t_bind()	Finds the endpoint for an address, and activates the endpoint.
t_open()	Creates a transport endpoint, and identifies the transport provided.

Coding XTI calls: Establishing a connection

Table 8 on page 175 lists the routines needed to establish a connection. For more information, see *CAE Specification: X/Open Transport Interface (XTI)*.

Table 8. Establishing a connection

Call	Description
t_accept()	Accepts a connection after a connect indication is received.
t_connect()	Requests connection to a transport user at a known destination.
t_listen()	Listens for connect information from other transport users.
t_rcvconnect()	Checks the status of a completed connect.

Coding XTI calls: Transferring data

Table 9 on page 175 lists the routines needed to transfer data. For more information, see *CAE Specification: X/Open Transport Interface (XTI)*.

Table 9. Transferring data

Routine	Description
t_rcv()	Receives normal or expedited data over a transport connection.
t_snd()	Sends normal or expedited data over a transport connection.

Coding XTI calls: Releasing a connection

Table 10 on page 175 lists the routines needed to release a connection. For more information, see *CAE Specification: X/Open Transport Interface (XTI)*.

Table 10. Releasing a connection

Call	Description
t_rcvdis()	Determines the reason for an abortive release or connection reject.
t_snddis()	Sends an abortive release or a connection reject.

Coding XTI calls: Disabling a connection

Table 11 on page 175 lists the routines needed to disable a connection. For more information, see *CAE Specification: X/Open Transport Interface (XTI)*.

Table 11. Disabling a connection

Call	Description
t_close()	Informs the XTI manager that you have finished with the endpoint, and frees any locally allocated resources assigned to endpoint.
t_unbind()	Resets the path to the transport endpoint. The connection is removed from the transport system, and requests for this path are denied.

Coding XTI calls: Managing events

Table 12 on page 176 lists the routines needed to manage events. Each XTI call handles one event at a time. Events are processed one at a time, and you can wait on only one event at a time. For more information, see *CAE Specification: X/Open Transport Interface (XTI)*.

Table 12. Managing events

Call	Description
t_look()	Returns the events current for a transport endpoint and notifies the calling program of an asynchronous event when the calling program is in synchronous mode.

Coding XTI calls: Using utility calls

Table 13 on page 176 lists utility routines that you can use to solve problems and monitor connections. For more information, see *CAE Specification: X/Open Transport Interface (XTI)*.

Table 13. Using utilities

Call	Description
t_error()	Returns the last error that occurred on a call to a transport function. You can add an identifying prefix to this call to aid in problem solving.
t_getinfo()	Returns information about the underlying transport protocol for the connection associated with file descriptor <i>fd</i> .
t_getstate()	Returns information about the state of the transport provider associated with file descriptor <i>fd</i> .

Coding XTI calls: Using system calls

Table 14 on page 176 lists system routines that you can use to manage your program. For more information see *CAE Specification: X/Open Transport Interface (XTI)*.

Table 14. System function calls

Call	Description
fcntl()	Controls the operating characteristics of sockets. For more information, see “selectex()” on page 143 .
select()	Checks descriptor sets to see if information is available for a read or a write. Select() also checks for pending exception conditions. For more information, see “select()” on page 140 .
selectex()	Extends the select() calls by allowing you to add an ECB to define extra events. For more information, see “selectex()” on page 143 .

Compiling and linking XTI applications using cataloged procedures

Several methods are available to compile, link-edit, and run your XTI program. This topic contains information about the data sets that you must include to run your XTI source program, using IBM-supplied cataloged procedures.

The following compile and link-edit sample procedures are supplied by IBM:

- XTICL is a sample compile and link-edit procedure.
- XTIC is a sample client execute procedure.
- XTIS is a sample server execute procedure.

Note: For more information about compiling and linking, see *IBM C/370 Programming Guide*.

XTICL

```
//XTICL JOB XTICLJOB
//*****
//*
//*      Communications Server IP
//*
//*      Licensed Materials - Program Property of IBM.
//*      This product contains "Restricted Materials of IBM"
//*      5694-A01 (C) Copyright IBM Corp. 1989, 2002
//*      US Government Users Restricted Rights -
//*      Use, duplication or disclosure restricted
//*      by GSA ADP Schedule Contract with IBM Corp.
//*      See IBM Copyright Instructions
//*
//* SMP/E Distribution Name: EZAEB02Z
//*
//*****
//*      SYMBOLIC DEFINITIONS
//*
//*      INSTLIB - TCPIP SEZAINST LIBRARY
//*      SEZALOD - TCPIP SEZALOAD LIBRARY
//*      SEZAMAC - TCPIP MACLIB
//*      SEZAMTX - TCPIP SEZACMTX LIBRARY
//*      SCEERUN - C/C++ SCEERUN LIBRARY
//*      SCBCCMP - C/C++ SCBCCMP
//*      CHEADRS - C/C++ HEADER LIBRARY
//*      CMSGGS - C/C++ MESSAGE LIBRARY
//*      SCEELKD - C/C++ SCEELKED LIBRARY
//*      INSTMEM - MEMBER TO COMPILE AND LINK
//*      OBJLIB - LIBRARY TO PUT INSTMEM OBJ DECK
//*      XTILOAD - LIBRARY FOR XTI LOAD MODULES
//*      SOUT - SYSOUT PARAMETER
//*
//*****
//CCOMP PROC REG='3072K',
//      CPARM='DEF(MVS),SOURCE,LIST,NOMARG,SEQ(73,80)',
//      INSTLIB=,
//      SEZALOD=,
//      SEZAMAC=,
//      SEZAMTX=,
//      SCEERUN=,
//      SCBCCMP=,
//      CHEADRS=,
//      CMSGGS=,
//      SCEELKD=,
//      OBJLIB=,
//      XTILOAD=,
//      DCB80='(RECFM=FB,LRECL=80,BLKSIZE=3200)',
//      DCB3200='(RECFM=FB,LRECL=3200,BLKSIZE=12800)',
//      SOUT=*
//*
//*****
//*      COMPILE STEP:
//*****
//*
//COMPILE EXEC PGM=CBCCRVR,
//      PARM=('&CPARM'),
//      REGION=&REG
//STEPLIB DD DSN=&SEZALOD,DISP=SHR
//      DD DSN=&SCEERUN,DISP=SHR
//      DD DSN=&SCBCCMP,DISP=SHR
//SYSLIB DD DSN=&SEZAMAC,DISP=SHR
//      DD DSN=&CHEADRS,DISP=SHR
//SYSIN DD DSN=&INSTLIB(&INSTMEM),DISP=SHR
//      DD DSN=&OBJLIB(&INSTMEM),DISP=SHR
//SYSMSG DD DUMMY,DSN=&CMSGGS,DISP=SHR
//SYSPRINT DD SYSOUT=&SOUT
//SYSPRT DD SYSOUT=&SOUT
//SYSTEM DD DUMMY
//SYSUT1 DD UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB80
//SYSUT5 DD UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT6 DD UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT7 DD UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT8 DD UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT9 DD UNIT=VIO,SPACE=(32000,(30,30)),
//      DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SYSUT10 DD SYSOUT=*
//*
```

```

//*****
//* LINKEDIT STEP:
//*****
//*
//LKED EXEC PGM=IEWL,COND=(1,LT),
// REGION=&REG
//OBJLIB DD DSN=&OBJLIB,DISP=SHR
//SYSLIB DD DSN=&SEZAMTX,DISP=SHR
// DD DSN=&SCEELKD,DISP=SHR
//SYSLMOD DD DSN=&XTILOAD,DISP=SHR
//SYSPRINT DD SYSOUT=&SOUT
//SYSUT1 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
// PEND
//*
//XTIC EXEC CCOMP,INSTMEM=XTICC
//LKED.SYSLIN DD *
// INCLUDE OBJLIB(XTICC)
// INCLUDE SYSLIB(XTI)
// MODE AMODE(31),RMODE(ANY)
// ENTRY CEESTART
// NAME XTIC(R)
//*
//XTIS EXEC CCOMP,INSTMEM=XTISC
//LKED.SYSLIN DD *
// INCLUDE OBJLIB(XTISC)
// INCLUDE SYSLIB(XTI)
// MODE AMODE(31),RMODE(ANY)
// ENTRY CEESTART
// NAME XTIS(R)
//*

```

Figure 57. Sample compile and link-edit job control procedure

XTIC

```
//XTICRUN JOB XTICJOB
//XTICPRC PROC P='XTIC',
//      IPADDR=XXX.XXX.XXX.XXX,
//      PORT='102',
//      REG=6M,
//      TIM=1440,
//      USERLIB='',
//      RUNLIB1='',
//      SOUT=*
//*****
//*                                     *
//*      Component Name: XTIC (alias EZAEC03S)      *
//*                                     *
//*      TCP/IP FOR CS z/OS      *
//*                                     *
//*      Licensed Materials - Program Property of IBM.      *
//*      This product contains "Restricted Materials of IBM"      *
//*      5694-A01 (C) Copyright IBM Corp. 1989,2005      *
//*      US Government Users Restricted Rights -      *
//*      Use, duplication or disclosure restricted      *
//*      by GSA ADP Schedule Contract with IBM Corp.      *
//*      See IBM Copyright Instructions      *
//*                                     *
//*      TCP/IP for CS z/OS      *
//*      SMP/E Distribution Name: EZAEC03S      *
//*                                     *
//*****
//*      SYMBOLIC DEFINITIONS      *
//*                                     *
//*      P      - PROGRAM TO EXECUTE      *
//*      IPADDR  - IP ADDRESS OF HOST THAT SERVER IS EXECUTING      *
//*      PORT    - PORT THE SERVER IS EXECUTING ON      *
//*      REG     - REGION SIZE      *
//*      TIM     - TIME PARAMETER      *
//*      USERLIB - LIBRARY WHERE THE XTIC EXECUTEABLE MODULE IS LOCATED *
//*      RUNLIB1 - C/C++ SCEERUN LIBRARY      *
//*      SOUT    - SYSOUT PARAMETER      *
//*                                     *
//*****
//XTIC EXEC PGM=&P,PARM='&PORT &IPADDR',
//      REGION=&REG,TIME=&TIM
//STEPLIB DD DSN=&USERLIB,DISP=SHR
//      DD DSN=&RUNLIB1,DISP=SHR
//SYSPRINT DD SYSOUT=&SOUT
//SYSERR DD SYSOUT=&SOUT
//SYSDEBUG DD SYSOUT=&SOUT
//LOG DD SYSOUT=&SOUT
//SYSABEND DD SYSOUT=&SOUT
//SYSUDUMP DD SYSOUT=&SOUT
//CEEDUMP DD SYSOUT=&SOUT
// PEND
// EXEC PROC=XTICPRC
```

Figure 58. Sample client execution job control procedure

XTIS

```
//XTISRUN JOB XTISJOB
//XTISPRC PROC P='XTIS',
//      REG=6M,
//      TIM=1440,
//      PORT='102',
//      USERLIB='',
//      RUNLIB1='',
//      SOUT=*
//*****
//*
//*      Component Name: XTIS (alias EZAEC03T)
//*
//*      TCP/IP FOR CS z/OS
//*
//*      Licensed Materials - Program Property of IBM.
//*      This product contains "Restricted Materials of IBM"
//*      5694-A01 (C) Copyright IBM Corp. 1989, 2005
//*      US Government Users Restricted Rights -
//*      Use, duplication or disclosure restricted
//*      by GSA ADP Schedule Contract with IBM Corp.
//*      See IBM Copyright Instructions
//*
//* TCP/IP for CS z/OS
//* SMP/E Distribution Name: EZAEC03T
//*
//*****
//*      SYMBOLIC DEFINITIONS
//*
//*      P      - PROGRAM TO EXECUTE
//*      REG     - REGION SIZE
//*      TIM     - TIME PARAMETER
//*      PORT    - PORT NUMBER TO START SERVER ON
//*      USERLIB - LIBRARY WHERE THE XTIS EXECUTEABLE MODULE IS LOCATED
//*      RUNLIB1 - C/C++ SCEERUN LIBRARY
//*      SOUT    - SYSOUT PARAMETER
//*
//*****
//XTIS EXEC PGM=&P,PARM='&PORT',
//      REGION=&REG,TIME=&TIM
//STEPLIB DD DSN=&USERLIB,DISP=SHR
//      DD DSN=&RUNLIB1,DISP=SHR
//SYSPRINT DD SYSOUT=&SOUT
//SYSERR DD SYSOUT=&SOUT
//SYSDEBUG DD SYSOUT=&SOUT
//LOG DD SYSOUT=&SOUT
//SYSABEND DD SYSOUT=&SOUT
//SYSUDUMP DD SYSOUT=&SOUT
//CEEDUMP DD SYSOUT=&SOUT
// PEND
// EXEC PROC=XTISPRC
```

Figure 59. Sample server execution job control procedure

Understanding XTI sample programs

This topic contains sample XTI socket programs. The XTI source code can be found in the SEZAINST data set.

Note: As with all TCP/IP applications, dynamic data set allocations are used unless explicitly overridden.

The following sample XTI socket programs are available:

Name when shipped	Alias name	Description
XTICC	EZAECOYL	XTI socket client sample program
XTISC	EZAECOYM	XTI socket server sample program

XTI socket client sample program

The following example shows an XTI socket client program.

```
/** IBMCOPYR *****/
/*
/* Component Name: XTICC.C (alias EZAEC0YL)
/*
/* Copyright:    Licensed Materials - Property of IBM
/*
/*              "Restricted Materials of IBM"
/*
/*              5647-A01
/*
/*              (C) Copyright IBM Corp. 1977, 1998
/*
/*              US Government Users Restricted Rights -
/*              Use, duplication or disclosure restricted by
/*              GSA ADP Schedule Contract with IBM Corp.
/*
/* Status:      CSV2R6
/*
/*
/* SMP/E Distribution Name: EZAEC0YL
/*
/*
/** IBMCOPYR *****/

static char ibmcopyr[] =
    "XTICC - Licensed Materials - Property of IBM. "
    "This module is \"Restricted Materials of IBM\" "
    "5647-A01 (C) Copyright IBM Corp. 1994. "
    "See IBM Copyright Instructions.";

/*****
/* XTIC Sample : Client
/*
/* Function:
/*
/* 1. Establishes an XTI endpoint (Asynchronous mode)
/* 2. Sends a connection request to an XTI server
/* 3. Receives the request
/* 4. Sends a block of data to the server
/* 5. Receives a block of data from the server
/* 6. Disconnects from the server
/* 7. Client stops
/*
/* Command line:
/*
/* XTIC hostname
/*
/*      hostname - name of the host that the server is running.
/*
*****/

#include "xti.h"
#include "xti1006.h"
#include "stdio.h"

/*
* bind request structure for t_bind()
*/

struct t_bind req,ret;

/*
* for client to make calls to server
*/

struct t_call scall,rcall;

/*
* store fd returned on open()
*/

int fd;

int tot_received;
```

```

char *hostname;

/*
 * data buffer
 */

char buf[25];

int looking;

/*
 * flags returned from t_rcv()
 */

int rflags,sflags;

/*
 * transport provider for t_open()
 */

char tprov[1][8] =
    { "RFC1006" } ;

/*
 * args that are optional
 */

int args;

int pnun = 102;
char *port = "102";
char *ctsel = "client";
char *stsel = "server";
unsigned int rqlen = 0;
struct xti1006tsap tsap, tsapret;
void cleanup(int);
void form_addr_1006(struct xti1006tsap *,int, char *, char*, int, int);

/*
 * MAIN line program starts here !!!
 */

main(argc,argv)
int argc;
char *argv[];
{
    /*
     * Check arguments. The host name is required. Host name is the
     * last parameter passed. Port can be changed by passing it as the
     * first parameter.
     */

    if ((argc > 3) | (argc < 2)) {
        fprintf(stderr,"Usage XTIC <port> <host>\n");
        exit(1);
    }

    if(argc==2)
        hostname = argv[1];
    else
    {
        hostname = argv[2];
        port = argv[1];
        pnun = (unsigned short) atoi(argv[1]);
    }

    /*
     * assume normal data
     */

    sflags = 0;

    /*
     * establish endpoint to t_listen() on
     */

    if ((fd = t_open(tprov[0],0_NONBLOCK,NULL)) < 0)
    {
        t_error("Error on t_open for FD");
        exit(t_errno);
    }
}

```

```

/*
 * compose req structure for t_bind() calls
 */

/*
 * length of tsap
 */

req.qlen = 0;
req.addr.len = sizeof(tsap);

/*
 * allocate the buffer to contain the
 * port and tsel to bind server to
 */

req.addr.buf = (char *)malloc(sizeof(tsap));

/*
 * fill address buffer with the address information
 */

form_addr_1006((struct xti1006tsap *)req.addr.buf, pnum, NULL, \
               ctset, fd, -1);

/*
 * now that we're done composing the req,
 * do the bind of fd to addr in req
 */

if (t_bind(fd, &req, NULL) != 0)
{
    t_error("ERROR ON BIND FOR FD");
    exit(t_errno);
}

/*
 * compose call structure for t_connect() call
 */

scall.addr.len = sizeof(tsap);
scall.addr.buf = (char *)malloc(sizeof(tsap));

/*
 * fill address buffer with the address information
 */

form_addr_1006((struct xti1006tsap *)scall.addr.buf, -1, hostname, \
               stset, fd, -1);

scall.opt.maxlen = 0;
scall.opt.len = 0;
scall.opt.buf = NULL;
scall.udata.len = 0;
scall.udata.buf = NULL;

rcall.addr.maxlen = sizeof(tsapret);
rcall.addr.buf = (char *)malloc(sizeof(tsapret));
rcall.opt.maxlen = 0;
rcall.udata.maxlen = 0;
rcall.udata.buf = NULL;

/*
 * issue connect request
 */

looking = t_connect(fd, &scall, &rcall);
if (looking < 0 & t_errno != TNODATA)
{
    t_error("ERROR ON CONNECT");
    cleanup(fd);
    exit(t_errno);
}

looking = 1;
while (looking)
{
    looking = t_look(fd);
    if (looking == T_CONNECT & looking > 0)
        looking = 0;
    else

```

```

        if (looking != 0)
        {
            t_error("ERROR ON LOOK");
            cleanup(fd);
            exit(t_errno);
        }
        else
            looking = 1;
    }

    /*
    * establish connection
    */

    looking = 1;
    while (looking)
        if (t_rcvconnect(fd,&rcall) == 0)
            looking = 0;
        else
            if (t_errno != TNODATA)
            {
                t_error("ERROR ON RCVCONNECT");
                cleanup(fd);
                exit(t_errno);
            }
            else
                looking = 1;

    /*
    * place message in buffer
    */

    memset(buf, 'B', 25);

    /*
    * send message to server
    */

    looking = 1;
    while (looking)
        if ((looking = t_snd(fd, buf, sizeof(buf), sflags)) < 0)
        {
            t_error("ERROR SENDING MESSAGE TO SERVER");
            cleanup(fd);
            exit(t_errno);
        }
        else
            if (looking == 0)
                looking = 1;
            else
                looking = 0;

    /*
    * receive data back from the server
    */

    looking = 1;
    while (looking)
    {
        if ((looking = t_rcv(fd, buf, sizeof(buf), &rflags)) > 0)
            looking = 0;
        else
        {
            if (looking < 0 & t_errno != TNODATA)
            {
                t_error("ERROR RECEIVING DATA FROM SERVER");
                cleanup(fd);
                exit(t_errno);
            }
            else
                looking = 1;
        }
    }

    /*
    * disconnect from server
    */

    looking = 1;
    while (looking)
        if (t_snddis(fd, NULL) == 0)
            looking = 0;
        else
            looking = 1;
    }

```

```

        t_error("ERROR DISCONNECTING FROM SERVER");
        cleanup(fd);
        exit(t_errno);
    }

    /*
    * if fd is an endpoint, try to close it
    */

    if (t_unbind(fd) != 0)
    {
        t_error("ERROR ON BIND FOR FD");
        exit(t_errno);
    }

    cleanup(fd);

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

/*****
void form_addr_1006(addrbuf1006,portnum,hostnmstr,tsselstr1006,fd1,fd2)

/*
* formats the provided address information
* into the buffer for RFC1006
*/

/*
* address buffer to be filled in
*/

struct xti1006tsap *addrbuf1006;
int    portnum;

/*
* hostnmstr represented as a string
*/

char    *hostnmstr;

/*
* tsel represented as a string
*/

char    *tselstr1006;

/*
* one possible endpoint to close if
* an error occurs in forming address
*/

int    fd1;

/*
* other possible endpoint to close
*/

int    fd2;
{

    /*
    * check validity of hostname
    * there's no way program can
    * continue without valid addr
    */

    if (strlen(hostnmstr) > 64)
    {
        fprintf(stderr,"hostname %s too long\n",hostnmstr);
        /*
        * don't want TADDRBUSY when you try to reuse the address
        */
        cleanup(fd1);
        cleanup(fd2);
        exit(TBADADDR);
    }
}

```

```

}

addrbuf1006->xti1006_hostnm_len = strlen(hostnmstr);
strcpy(addrbuf1006->xti1006_hostnm,hostnmstr);

/*
 * check validity of hostname
 * there's no way program can
 * continue without valid addr
 */

if (strlen(tselstr1006) > 64)
{
    fprintf(stderr,"tsel %s too long\n",tselstr1006);
    /*
     * don't want TADDRBUSY when you try to reuse the address
     */
    cleanup(fd1);
    cleanup(fd2);
    exit(TBADADDR);
}

addrbuf1006->xti1006_tsel_len = strlen(tselstr1006);
strcpy(addrbuf1006->xti1006_tsel,tselstr1006);

if (tselstr1006 == "Nulltsap")
{
    addrbuf1006->xti1006_tsel_len = 0;
    strcpy(addrbuf1006->xti1006_tsel,NULL);
}
else
{
    addrbuf1006->xti1006_tsel_len = strlen(tselstr1006);
    strcpy(addrbuf1006->xti1006_tsel,tselstr1006);
} /* endif */

if (portnum != -1)
    addrbuf1006->xti1006_tset = portnum;
}
/*****/

void cleanup(fd)

int fd;

{
    if (fd >= 0)
        if (t_close(fd) != 0)
        {
            fprintf(stderr,"unable to t_close() endpoint while");
            fprintf(stderr," cleaning up from error\n");
        }
}

```

Figure 60. Sample client code for XTI

XTI socket server sample program

As with all TCP/IP applications, dynamic dataset allocations are used unless explicitly overridden. For example, the TCPIP.DATA file can be specified using the SYSTCPD DD JCL statement. For more information, see [Chapter 10, “C Socket application programming interface,”](#) on page 85.

The following example shows an XTI socket server program.

```

/**** IBMCOPYR *****/
/*
/* Component Name: XTISC.C (alias EZAEC0YM)
/*
/* Copyright:    Licensed Materials - Property of IBM
/*
/*              "Restricted Materials of IBM"
/*
/*              5647-A01
/*
/*              (C) Copyright IBM Corp. 1977, 1998
/*
/*

```

```

/*          US Government Users Restricted Rights -          */
/*          Use, duplication or disclosure restricted by      */
/*          GSA ADP Schedule Contract with IBM Corp.        */
/*          */
/* Status:      CSV2R6                                     */
/*          */
/*          SMP/E Distribution Name: EZAEC0YM                */
/*          */
/*          */
/**** IBMCOPYR *****/

static char ibmcopyr[]=
    "XTISC - Licensed Materials - Property of IBM. "
    "This module is \"Restricted Materials of IBM\" "
    "5647-A01 (C) Copyright IBM Corp. 1994. "
    "See IBM Copyright Instructions.";

/*****
/* XTIS Sample : Server
/*
/* Function:
/*
/* 1. Establishes an XTI endpoint (Asynchronous mode)
/* 2. Listens for a connection request from an XTI client
/* 3. Accepts the connection request
/* 4. Receives a block of data from the client
/* 5. Echos the data back to the client
/* 6. Waits for the disconnect request from the XTI client
/* 7. Server stops
/*
/* Command line:
/*
/* XTIS      H
/*
*****/

#include "xti.h"
#include "xti1006.h"
#include "stdio.h"

/*
* bind request structure for t_bind()
*/

struct t_bind req,ret;

/*
* for server to listen for calls with
*/

struct t_call call;

/*
* descriptor to t_listen() on
*/

int fd;

/*
* descriptor to t_accept() on
*/

int resfd;

int tot_received;

/*
* data buffer
*/

char buf[25];

int looking;

/*
* flags returned from t_rcv()
*/

int rflags,sflags;

/*

```

```

* transport provider for t_open()
*/

char tprov[1][8] =
    { "RFC1006" } ;

/*
* args that are optional
*/

int args;

int tot_sent;
int pnum = 102;
char *port = "102";
char *hostnm;
char *stsel = "server";
unsigned int rqlen = 0;
struct xti1006tsap tsap, tsapret;
void cleanup(int);
void form_addr_1006(struct xti1006tsap *,int, char *, char*, int, int);

/*
* MAIN line program starts here !!!
*/

main(argc,argv)
int argc;
char *argv[];
{
    /*
    * Check arguments. No arguments should be passed to the server
    */

    if (argc > 2) {
        fprintf(stderr,"Usage : XTIS <port>\n");
        exit(1);
    }

    if(argc == 2)
    {
        pnum = (unsigned short) atoi(argv[1]);
        port = argv[1];
    }
    /*
    * assume normal data
    */

    sflags = 0;

    /*
    * establish endpoint to t_listen() on
    */

    if ((fd = t_open(tprov[0],O_NONBLOCK,NULL)) < 0)
    {
        t_error("Error on t_open");
        exit(t_errno);
    }

    /*
    * establish endpoint to t_accept() on
    */

    if ((resfd = t_open(tprov[0],O_NONBLOCK,NULL)) < 0)
    {
        t_error("Error on t_open");
        cleanup(fd);
        exit(t_errno);
    }

    /*
    * compose req structure for t_bind() calls
    */

    /*
    * length of tsap
    */

    req.addr.len = sizeof(tsap);

```



```

/*
 * allocate the buffer to contain the
 * port and tsel to bind server to
 */

req.addr.buf = (char *)malloc(sizeof(tsap));

/*
 * fill address buffer with the address information
 */

form_addr_1006((struct xti1006tsap *)req.addr.buf, \
               pnun, \
               NULL, \
               stsel, \
               fd, \
               resfd);

/*
 * length of tsap
 */

ret.addr.maxlen = sizeof(tsapret);
ret.addr.buf = (char *)malloc(sizeof(tsapret));

/*
 * listening endpoint needs qlen > 0,
 * ability to queue 10 requests
 */

req.qlen = 10;
ret.qlen = rqlen;

/*
 * now that we're done composing the req,
 * do the bind of fd to addr in req
 */

if (t_bind(fd,&req,&ret) != 0)
{
    t_error("Error on t_bind");
    cleanup(fd);
    cleanup(resfd);
    exit(t_errno);
}

/*
 * accepting endpoint with same addr needs qlen == 0
 */

req.qlen = 0;

/*
 * now that we're done composing the req,
 * do the bind of resfd to addr in req
 */

if (t_bind(resfd,&req,&ret) != 0)
{
    t_error("Error on t_bind");
    cleanup(fd);
    cleanup(resfd);
    exit(t_errno);
}

/*
 * initialize call receipt structure for t_listen()
 */

call.opt.maxlen = 0;
call.addr.len = 0;
call.opt.len = 0;
call.udata.len = 0;
call.opt.buf = NULL;

call.addr.maxlen = sizeof(tsapret); /* listen for return*/
call.addr.buf = (char *)malloc(sizeof(tsapret));

call.udata.maxlen = 0;
call.udata.buf = NULL;

/*

```

```

/* wait for connect req & get seq num in the call variable
*/

looking = 1;
while (looking)
    if (t_listen(fd,&call) == 0)
        looking = 0;
    else
        if (t_errno != TNODATA)
        {
            t_error("Error on t_accept");
            cleanup(fd);
            cleanup(resfd);
            exit(t_errno);
        }

/*
* accept the connection on the accepting endpoint
*/

if (t_accept(fd,resfd,&call) != 0)
{
    t_error("Error on t_accept");
    cleanup(fd);
    cleanup(resfd);
    exit(t_errno);
}

/*
* receive data from the client
*/

looking = 1;
while (looking)
    if (t_rcv(resfd,buf,sizeof(buf),&rflags) > 0)
        looking = 0;
    else
        if (t_errno != TNODATA)
        {
            t_error("Error on t_rcv");
            cleanup(fd);
            cleanup(resfd);
            exit(t_errno);
        }
    }

/*
* sent data back to the client
*/

strcpy(buf,"DATA FROM SERVER");

looking = 1;
while (looking)
    if (t_snd(resfd,buf,sizeof(buf),sflags) > 0)
        looking = 0;

/*
* wait for disconnect from the client
*/

looking = 1;
while (looking)
    if (t_look(resfd) == T_DISCONNECT)
        looking = 0;

/*
* receive the disconnect request
*/

looking = 1;
while (looking)
    if (t_rcvdis(resfd,NULL) == 0)
        looking = 0;

/*
* unbind the endpoints
*/

if (t_unbind(resfd) != 0)
{
    t_error("Error on t_unbind for resfd");
    cleanup(fd);

```

```

        cleanup(resfd);
        exit(t_errno);
    }

    if (t_unbind(fd) != 0)
    {
        t_error("Error on t_unbind for fd");
        cleanup(fd);
        cleanup(resfd);
        exit(t_errno);
    }

    /*
    * if fd is an endpoint, try to close it
    */

    cleanup(fd);

    /*
    * if resfd is an endpoint, try to close it
    */

    cleanup(resfd);

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

/*****
void form_addr_1006(addrbuf1006,portnum,hostnmstr,tsselstr1006,fd1,fd2)

/*
* formats the provided address information
* into the buffer for RFC1006
*/

/*
* address buffer to be filled in
*/

struct xti1006tsap *addrbuf1006;

int    portnum;

/*
* hostnmstr represented as a string
*/

char          *hostnmstr;

/*
* tsel represented as a string
*/

char          *tselstr1006;

/*
* one possible endpoint to close if
* an error occurs in forming address
*/

int    fd1;

/*
* other possible endpoint to close
*/

int    fd2;
{

    /*
    * check validity of hostname
    * there's no way program can
    * continue without valid addr
    */

    if (strlen(hostnmstr) > 64)
    {

```

```

        fprintf(stderr,"hostname %s too long\n",hostnmstr);
        /*
        * don't want TADDRBUSY when you try to reuse the address
        */
        cleanup(fd1);
        cleanup(fd2);
        exit(TBADADDR);
    }

    addrbuf1006->xti1006_hostnm_len = strlen(hostnmstr);
    strcpy(addrbuf1006->xti1006_hostnm,hostnmstr);

    /*
    * check validity of hostname
    * there's no way program can
    * continue without valid addr
    */

    if (strlen(tselstr1006) > 64)
    {
        fprintf(stderr,"tsel %s too long\n",tselstr1006);
        /*
        * don't want TADDRBUSY when you try to reuse the address
        */
        cleanup(fd1);
        cleanup(fd2);
        exit(TBADADDR);
    }

    addrbuf1006->xti1006_tsel_len = strlen(tselstr1006);
    strcpy(addrbuf1006->xti1006_tsel,tselstr1006);

    if (tselstr1006 == "Nulltsap")
    {
        addrbuf1006->xti1006_tsel_len = 0;
        strcpy(addrbuf1006->xti1006_tsel,NULL);
    }
    else
    {
        addrbuf1006->xti1006_tsel_len = strlen(tselstr1006);
        strcpy(addrbuf1006->xti1006_tsel,tselstr1006);
    } /* endif */

    if (portnum != -1)
        addrbuf1006->xti1006_tset = portnum;
}
/*****

void cleanup(fd)

int fd;

{
    if (fd >= 0)
        if (t_close(fd) != 0)
        {
            fprintf(stderr,"unable to t_close() endpoint while");
            fprintf(stderr," cleaning up from error\n");
        }
}

```

Figure 61. Sample server code for XTI

Chapter 12. Macro application programming interface

This information describes the macro API for IPv4 or IPv6 socket application programs written in z/OS assembler language. The macro interface can be used to produce reentrant modules and can be used in a multithread environment.

The following topics are included:

- Environmental restrictions and programming requirements
- Input register information
- Output register information
- Compatibility considerations
- Defining storage for the macro API
- Understanding common parameter descriptions
- Error messages and return codes
- Characteristics of sockets
- Task management and asynchronous function processing
- Using an unsolicited event exit routine
- Diagnosing problems in applications using the macro API
- Macros for assembler programs
- Macro interface assembler language sample programs

Sockets API environmental restrictions and programming requirements

The following restrictions apply to both the Macro Socket API and the Callable Socket API:

Function	Restriction
SRB mode	These APIs can only be invoked in TCB mode (task mode).
Cross-memory mode	These APIs can only be invoked in a non-cross-memory environment (PASN=SASN=HASN).
Functional Recovery Routine (FRR)	Do not invoke these APIs with an FRR set. This will cause system recovery routines to be bypassed and severely damage the system.
Locks	No locks should be held when issuing these calls.
INITAP and TERMAPI socket commands	The INITAPI and TERMAPI socket commands must be issued under the same task.
Storage	Storage acquired for the purpose of containing data returned from a socket call must be obtained in the same key as the application program status word (PSW) at the time of the socket call. This includes the ECB that is posted upon completion of an asynchronous EZASMI call that is issued after an EZASMI TYPE=INITAPI with the ASYNC=('ECB') option has been issued.

Function	Restriction
Nested socket API calls	You cannot issue nested API calls within the same task. That is, if a request block (RB) issues a socket API call and is interrupted by an interrupt request block (IRB) in an STIMER exit, any additional socket API calls that the IRB attempts to issue are detected and flagged as an error.
Addressability mode (Amode) considerations	The EZASMI interface can be invoked while the caller is in either 31-bit or 24-bit Amode. However, if the application is running in 24-bit addressability mode at the time of the call, all addresses of parameters passed by the application must be addressable in 31-bit Amode. This implies that even if the addresses being passed reside in storage below the 16 MB line (and therefore addressable by 24-bit Amode programs) the high-order byte of these addresses needs to be 0.
Use of z/OS UNIX System Services	Each z/OS UNIX process within an address space that uses the EZASMI API should not use any z/OS UNIX System Services socket API facilities such as z/OS UNIX Assembler Callable Services or Language Environment for z/OS C/C++. Doing so can yield unpredictable results.
Dynamic allocation	Socket calls should not be issued during START (initialization) processing for LOGONs, MOUNTs, or started tasks. This is because they require dynamic allocation that can fail during these times.

The EZASMI macro is located in SEZACMAC.

Sockets API input register information

Before invoking the sockets API, the general purpose registers (GPRs) need to contain the following registers:

Register Contents

0-1

N/A

2-12

N/A, unless referenced by a macro parameter

13

Pointer to a standard save area in the key of the caller

14-15

N/A

The contents of the access registers (ARs) on entry to the sockets API call are not used.

When control returns to the caller, the access registers (ARs) contain:

Register Contents

0-1

Used as work registers by the system

2-14

Unchanged

15

Used as a work register by the system

If a caller depends on register contents to remain the same before and after issuing a service, the caller must save the contents of a register before issuing the service and restore them after the system returns control.

Sockets API output register information

When control returns to the caller, the general purpose registers (GPRs) contain:

Register Contents

0-1

Used as work registers by the system

2-13

Unchanged

14

Used as a work register by the system

15

- For synchronous calls, it contains the entry point address of EZBSOH03.
- For asynchronous calls, see [“Task management and asynchronous function processing” on page 198](#).

When control returns to the caller, the access registers (ARs) contain:

Register Contents

0-1

Used as work registers by the system

2-14

Unchanged

15

Used as a work register by the system

If a caller depends on register contents to remain the same before and after issuing a service, the caller must save the contents of a register before issuing the service and restore them after the system returns control.

Sockets API compatibility considerations

Unless noted in [z/OS Communications Server: New Function Summary](#), an application program compiled and link edited on a release of z/OS Communications Server IP can be used on higher level releases. That is, the API is upward compatible.

Application programs that are compiled and link edited on a release of z/OS Communications Server IP cannot be used on older releases. That is, the API is not downward compatible.

Defining storage for the macro API

The macro API requires the definition of a task storage area.

The task storage area must be known to and addressable by all socket users communicating across a specified connection. A connection runs between the application and TCP/IP. The most common way to organize storage is to assign one connection to each MVS subtask. If there are multiple modules using sockets within a single task or connection, you must provide the address of the task storage to every user.

The following information describes how to define the address of the task storage:

- Code the instruction EZASMI TYPE=TASK with STORAGE=CSECT as part of the program code. This makes the program nonreentrant, but simplifies the code. The expansion of this instruction generates the equate field, TIELENTN, which is equal to the length of the storage area.
- Code the instruction EZASMI TYPE=TASK with STORAGE=DSECT as part of the program code. The expansion of this instruction generates the equate field, TIELENTN, which is equal to the length of the storage area. This can be used to issue an MVS GETMAIN to allocate the required storage.

Guideline: Clear the task storage prior to calling EZASMI TYPE=INIT. If the EZASMI TYPE=TASK definition is not named, you can use the EZASMI default storage name of EZASMTIE. Use the TIELENTN field to determine the length of the EZASMTIE storage to clear.

The defining program must make the address of this storage available to all other programs using this connection. Programs running in these tasks must define the storage mapping with an EZASMI TYPE=TASK with STORAGE=DSECT.

If the task storage is used to create a connection that is not deleted by a TERMAPI or termination of the task that created the connection, then the task storage can be in use by the macro API, even if the task storage is freed or cleared. If the application attempts to connect to TCP/IP by using a task storage area that is currently in use, the connection attempt fails with ERRNO 10335. The application cannot free or clear a task storage area that is currently in use. The application must first do a TERMAPI before freeing or clearing the task storage.

Restriction: The task storage area used on an INITAPI call cannot be moved or copied to other storage locations. Attempting to do so will result in an implicit INITAPI being performed with unexpected results.

The EZASMI TYPE=TASK macro generates only one parameter list for a connection. This can lead to overlay problems for programs using APITYPE=3 connections (multiple calls can be issued simultaneously). For more detail on APITYPE=3 connections, see [“Task management and asynchronous function processing”](#) on page 198. A program should use the following format to build unique parameter list storage areas if it will be issuing multiple calls simultaneously on one connection:

```

BINDPRML  EZASMI  MF=L  This will generate the storage used for
                        building the parm list in the following BIND call
                        EZASMI  TYPE=BIND,                                X
                        S=SOCKDESC,                                       X
                        NAME=NAMEID,                                       X
                        ERRNO=ERRNO,                                       X
                        RETCODE=RETCODE,                                   X
                        ECB=ECB1,                                          X
                        MF=(E,BINDPRML)

```

This example of an asynchronous BIND macro would use the MF=L macro to generate the parameter list. The fields that are common across all macro calls, for example, RETCODE and ERRNO, must be unique for each outstanding call.

You can create multiple connections to TCP/IP from a single task. Each of these connections functions independently of the other and is identified by its own task interface element (TIE). The TASK parameter can be used to explicitly reference a TIE. If you do not include the TASK parameter, the macro uses the TIE generated by the EZASMI TYPE=TASK macro.

TIE1	DS XL(TIELENTN)	Length of TIE
EZASMI	TYPE=INITAPI,	
	MAXSOC=MAX75,	X
	ERRNO=ERRNO,	X
	RETCODE=RETCODE,	X
	APITYPE=2,	X
	MAXSNO=MAXS,	X
	TASK=TIE1	
EZASMI	TYPE=SOCKET,	
	AF='INET',	X
	SOCTYPE='STREAM',	X
	ERRNO=ERRNO,	X
	RETCODE=RETCODE,	X
	TASK=TIE1	

In this example, the TIE TIE1 is used for the connection, not the TIE generated by the EZASMI TYPE=TASK macro.

Understanding common parameter descriptions

The following describes the parameters and concepts common to the macros described in this topic.

Parameter Description

address

The name of the field that contains the value of the parameter. The following example illustrates a BIND macro where SOCKNO is set to 2.

```
MVC    SOCKNO,=H'2'  
EZASMI TYPE=BIND,S=SOCKNO
```

**indaddr*

The name of the address field that contains the address of the field containing the parameter. The following example produces the same result as the example above.

```
MVC    SOCKNO,=H'2'  
LA      0,SOCKNO  
ST      0,SOCKADD  
EZASMI TYPE=BIND,S=*SOCKADD
```

(reg)

The name (equated to a number) or the number of a general purpose register. Do not use a register 0, 1, 14, or 15. The following example produces the same result as the previous examples.

```
MVC    SOCKNO,=H'2'  
LA      3,SOCKNO  
EZASMI TYPE=BIND,SOCKNO=(3)
```

'value'

A literal value for the parameter; for example, AF='INET'

Sockets API error messages and return codes

For information about error messages, see [z/OS Communications Server: IP Messages Volume 1 \(EZA\)](#).

For information about codes returned by TCP/IP, see [Appendix B, "Socket call error return codes,"](#) on [page 743](#).

Characteristics of sockets

For stream sockets, data is processed as streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and application A sends 1000 bytes, each call to the SEND function can return 1 byte, 10 bytes, or the entire 1000 bytes, with the number of bytes sent returned in the RETCODE call. Therefore, applications using stream sockets should place the READ call and the SEND call in a loop that repeats until all of the data has been sent or received.

PROTO specifies a particular protocol to be used with the socket. In most cases, a single protocol exists to support one type of socket in a domain (not true with raw sockets). If **PROTO** is set to 0, the system selects the default protocol number for the domain and socket type requested. The **PROTO** defaults are TCP for stream sockets and UDP for datagram sockets. There is no default for raw sockets.

SOCK_STREAM sockets model duplex byte streams. They provide reliable, flow-controlled connections between peer applications. Stream sockets are either active or passive. Active sockets are used by clients who initiate connection requests with CONNECT. By default, SOCKET creates active sockets. Passive sockets are used by servers to accept connection requests with the CONNECT macro. An active socket is transformed into a passive socket by binding a name to the socket with the BIND macro and by indicating

a willingness to accept connections with the LISTEN macro. If a socket is passive, it cannot be used to initiate connection requests.

In the AF_INET or AF_INET6 domain, the BIND macro, applied to a stream socket, lets the application specify the networks from which it is willing to accept connection requests. The application can fully specify the network interface by setting the Internet address field in the address structure to the Internet address of a network interface. Alternatively, the application can set the address in the name structure to zeros to indicate that it wants to receive connection requests from any network.

After a connection has been established between stream sockets, the data transfer macros READ, WRITE, SEND, RECV, SENDTO, and RECVFROM can be used. Usually, the READ-WRITE or SEND-RECV pairs are used for sending data on stream sockets.

SOCK_DGRAM sockets are used to model datagrams. They provide connectionless message exchange without guarantees of reliability. Messages sent have a maximum size. Datagram sockets are not supported in the AF_IUCV domain.

The active or passive concepts for stream sockets do not apply to datagram sockets. Servers must still call BIND to name a socket and to specify from which network interfaces it wants to receive datagrams. Wildcard addressing, as described for stream sockets, also applies to datagram sockets. Because datagram sockets are connectionless, the LISTEN macro has no meaning for them and must not be used.

After an application receives a datagram socket, it can exchange datagrams using the SENDTO and RECVFROM macros. If the application goes one step further by calling CONNECT and fully specifying the name of the peer with which all messages are exchanged, then the other data transfer macros READ, WRITE, SEND, and RECV can be used as well. For more information about placing a socket into the connected state, see [“CONNECT” on page 401](#).

Datagram sockets allow message broadcasting to multiple recipients. Setting the destination address to a broadcast address depends on the network interface (address class and whether subnets are used).

SOCK_RAW sockets supply an interface to lower layer protocols, such as IP. You can use this interface to bypass the transport layer when you need direct access to lower layer protocols. Raw sockets are also used to test new protocols. Raw sockets are not supported in the AF_IUCV domain.

Raw sockets are connectionless and data transfer is the same as for datagram sockets. You can also use the CONNECT macro to specify a peer socket in the same way that is previously described for datagram sockets.

Outgoing datagrams have an IP header prefixed to them. Your program receives incoming datagrams with the IP header intact. You can set and inspect IP options by using the SETSOCKOPT and GETSOCKOPT macros.

Use the CLOSE macro to deallocate sockets.

Regardless of the type of socket (SOCK_STREAM, SOCK_DGRAM or SOCK_RAW), all commands that pass a socket address must be consistent with the address family specified when the socket was opened. If the socket was opened with an address family of AF_INET, then any command for that socket that includes a socket address must use an AF_INET socket address. If the socket was opened with an address family of AF_INET6, then any command for that socket that includes a socket address must use an AF_INET6 socket address.

Task management and asynchronous function processing

The sockets extended interface allows asynchronous operation, although by default the task issuing a macro request is put into a WAIT state until the requested function completes. At that time, the issuing task resumes and continues execution.

If you do not want the issuing task to be placed into a WAIT while its request is processed, use asynchronous function processing.

Macro API asynchronous function processing: How it works

The macro API provides for asynchronous function processing in two forms. Both forms cause the system to return control to the application immediately after the function request has been sent to TCP/IP. The difference between the two forms is in how the application is notified when the function is completed:

ECB method

Enables you to pass an MVS event control block (ECB) on each socket call. The socket library returns control to the program immediately and posts the ECB when the call has completed.

EXIT method

Enables you to specify the entry point of an exit routine using the INITAPI() call. The individual socket calls immediately return control to the program and the socket library drives the specified exit routine when the socket call is complete.

In either case, the function is completed when the notification is delivered. Note that the notification can be delivered at any time, in some cases even before the application has received control back from the EZASMI macro call. It is therefore important that the application is ready to handle a notification as soon as it issues the EZASMI macro call.

Like nonblocking calls, asynchronous calls return control to your program immediately. But in this case, there is no need to reissue the call. When the requested event has taken place, an ECB is posted or an exit routine is driven.

Using the API macro, you can specify APITYPE=2 or APITYPE=3

APITYPE=2

Allows an asynchronous macro API program to have only one outstanding socket call per socket descriptor. An APITYPE=2 program can use macro API asynchronous calls, but synchronous calls are equally well supported.

APITYPE=3

Allows an asynchronous macro API program to have many outstanding socket calls per socket descriptor. Only the macro API supports APITYPE=3. An APITYPE=3 program must use macro API asynchronous calls with either an ECB or REQAREA parameter.

The REQAREA parameter is used in macros using the EXIT form. This parameter is mutually exclusive with the ECB parameter used with the ECB form.

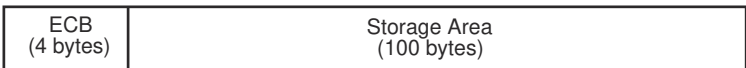


Figure 62. ECB input parameter

Like the ECB parameter, the REQAREA parameter points to an area that contains:

- A 4-byte token that is presented to your asynchronous exit routine when the response to this function request is complete
- A 100-byte storage area that is used by the interface to save the state information

Note: This storage must not be modified or freed until the macro function has completed and the ECB has been posted, or the asynchronous exit has been driven.

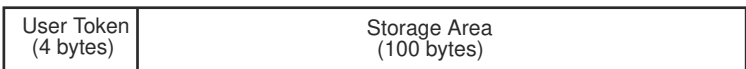


Figure 63. User token setting

Before you issue the macro, you must set the first word of the 104 bytes to a token of any value. The token is used by your asynchronous exit routine to determine the function completion event for which it is being invoked.

Asynchronous functions are processed in the following sequence:

1. The application must issue the EZASMI TYPE=INITAPI with ASYNC='ECB' or ASYNC=('EXIT', AEEXIT). The ASYNC parameter notifies the API that asynchronous processing is to be used for this connection. The API notes the type of asynchronous processing to be used, ECB or EXIT, and specifies the use of the asynchronous exit routine for this connection.
2. When a function request is issued by the application, the API takes one of the following actions:
 - If the type of asynchronous processing is ECB, and an ECB is supplied in the function request, the API returns control to the application. If Register 15 is 0, the ECB is posted when the function has completed. Note that the ECB might be posted prior to when control is returned to the application.
 - If the type of asynchronous processing is EXIT, and a REQAREA parameter is supplied in the function request, the API returns control to the application. If Register 15 is 0, the exit routine is invoked when the function has completed. Note that the exit can be invoked prior to when control is returned to the application.

In either case, Register 15 is used to inform the caller whether or not the ECB is posted or asynchronous exit driven. Therefore, you must not use Register 15 for the RETCODE parameter.

When the asynchronous exit routine is invoked, the following linkage conventions are used:

GPR0

Register Setting

0

Normal return

1

TCP/IP address space has terminated (TCPEND).

GPR1

Points to a doubleword field containing the following information:

WORD1

The token specified by the INITAPI macro

WORD2

The token specified by the functional request macro (First 4 bytes of the REQAREA storage)

GPR13

Points to standard MVS save area in the same key as the application PSW at the time of the INITAPI command.

GPR14

Return address

GPR15

Entry point of the exit routine

The following example shows how to code an asynchronous macro function:

```
*****
*   READ A BUFFER OF DATA FROM THE CONNECTION PEER. I MAY NEED TO   *
*   WAIT SO GIVE CONTROL BACK TO ME AND LET ME ISSUE MY OWN WAIT.    *
*   IT COULD BE PART OF A WAIT WHICH WOULD INCLUDE OTHER EVENTS.    *
*   SPECIFY ECB/STORAGE AREA FOR INTERFACE.                          *
*****
EZASMI TYPE=READ,                                     X
      S=SOCKNO,                                       X
      NBYTES=COUNT,                                X
      BUF=DATABUF,                                   X
      ERRNO=ERROR,                                    X
      RETCODE=RCODE,                                  X
      ECB=MYECB,                                      X
      ERROR=ERRORRTN
LTR    R15,R15      Was macro function passed to TCP/IP?
BNZ    BADRCODE     If no, ECB will not be posted
```

Asynchronous exit environmental and programming considerations

When utilizing the ASYNC=EXIT option of the EZASMI macro, the following requirements need to be considered:

- Asynchronous calls can only be issued from a single request block (RB) in a given task (TCB).

The first RB that issues an ASYNC EZASMI call under a given task is deemed as the target RB that is interrupted when an asynchronous exit needs to be driven. This means that after an asynchronous EZASMI macro call is invoked you should not invoke any services that cause the current RB to no longer be the top RB for this task (for example, a LINK call). If the target RB is no longer the top RB at the time that the exit needs to be driven, then the exit is deferred until the target RB becomes the top RB. One exception to this rule is that EZASMI calls can be issued under the asynchronous user exit.

- EZASMI macro calls within the asynchronous exits.

While running the asynchronous exit notification routine, an application can issue other EZASMI calls. However, the application should avoid issuing any blocking calls and should not enter into long delays. Doing so delays any additional exits from being driven and also blocks the TCB that made the original call. Note that TERMAPI should not be issued under the asynchronous exit.

- Linkage stack.

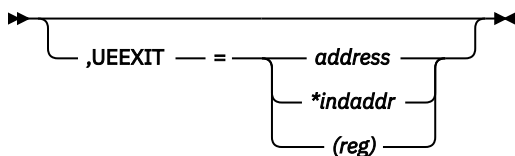
Applications issuing EZASMI macro asynchronous exit calls should not issue any PC instructions that cause the system linkage to be used. Doing so delays the asynchronous exits from being driven until the linkage stack entry is removed. If the linkage stack entry is not removed, the exit will not be driven.

- Asynchronous exits are given control in the same key as the program status word (PSW) key of the TCB from which the EZASMI call was issued.

Using an unsolicited event-exit routine

The unsolicited event-exit routine enables an application to specify an event exit routine that is invoked when an unsolicited event occurs. This exit routine can be a part of the program that specifies it, or it can be a separate module. The exit routine must be resident at the time that the EZASMI TYPE=INITAPI macro is issued, and it must stay resident until the EZASMI TYPE=TERMAPI macro is issued.

The user invokes this facility by the issuing the optional *UEEXIT* parameter that is used in the EZASMI TYPE=INITAPI macro, as shown in the following syntax fragment:



Keyword

Description

UEEXIT

A double word value that is composed of two positional parameters. The first parameter is the address of the event-exit routine that is invoked when an unsolicited event occurs. The second parameter is the address of the token that is passed to the exit routine. On entry to the unsolicited event-exit routine, the general purpose registers (GPRs) contain the following values:

GPR0

Register setting. The following values are supported:

0

TPC/IP is active

1

TCP/IP is inactive

GPR1

Address of the token that is specified in the INITAPI macro

GPR13

Pointer to a standard MVS save area. This save area is in the same key that the application program status word (PSW) was in when the EZASMI TYPE=INITAPI macro was issued.

GPR14

Return address

The following code example shows how the EZASMI TYPE=INITAPI macro can be used to specify an unsolicited event-exit routine (MYUEE). The MYUEE storage definition contains the address of the exit routine, MYUEE1, followed by the address of the user exit token, UETOKEN:

```

EZASMI TYPE=INITAPI,      Issue INITAPI Macro           X
    SUBTASK=SUBTASK,      SPECIFY SUBTASK IDENTIFIER    X
    MAXSOC=MAXSOC,        SPECIFY MAXIMUM NUMBER OF SOCKETS X
    MAXSNO=MAXSNO,        (HIGHEST SOCKET NUMBER ASSIGNED) X
    ERRNO=ERRNO,          (Specify ERRNO field)          X
    RETCODE=RETCODE,      (Specify RETCODE field)        X
    APITYPE=APITYPE,      (SPECIFY APITYPE FIELD)        X
    ERROR=ERROR,          ABEND IF ERROR ON MACRO        X
    UEEXIT=MYUEE,         X
    ASYNC=('EXIT',MYEXIT) (SPECIFY AN EXIT)              X

*-----*
*      UNSOLICITED EVENT EXIT
*-----*
MYUEE  DC  A(MYUEE1,UETOKEN)
UETOKEN DS  F
MYUEE1 SAVE (14,12),T,*
      LR  R2,R15
      USING MYUEE1,R2
UEKEY  WTO 'UEEXIT BEING DRIVEN'
      EZASMI TYPE=TERMAPI      Issue EZASMI Macro for Termap
      POST ECB,1
      RETURN (14,12),T,RC=0
      DROP  R2
```

Diagnosing problems in applications using the macro API

TCP/IP provides a trace facility that can be helpful in diagnosing problems in applications using the Macro API. The trace is implemented using the TCP/IP Component Trace (CTRACE) SOCKAPI trace option. The SOCKAPI trace option allows all Macro socket API calls issued by an application to be traced in the TCP/IP CTRACE. The SOCKAPI trace records include information such as the type of socket call, input, and output parameters and return codes. This trace can be helpful in isolating failing socket API calls and in determining the nature of the error or the history of socket API calls that might be the cause of an error. For more information about the SOCKAPI trace option, see [z/OS Communications Server: IP Diagnosis Guide](#).

Macros for assembler programs

This information contains the description, syntax, parameters, and other related information for every macro included in this API.

The EZASMI macro is located in the SEZACMAC library.

ACCEPT

The ACCEPT macro is issued when the server receives a connection request from a client. ACCEPT points to a socket that was created with a SOCKET macro and marked by a LISTEN macro. If a process waits for the completion of connection requests from several peer processes, a later ACCEPT macro can block until one of the CONNECT macros completes. To avoid this, issue a SELECT macro between the CONNECT and

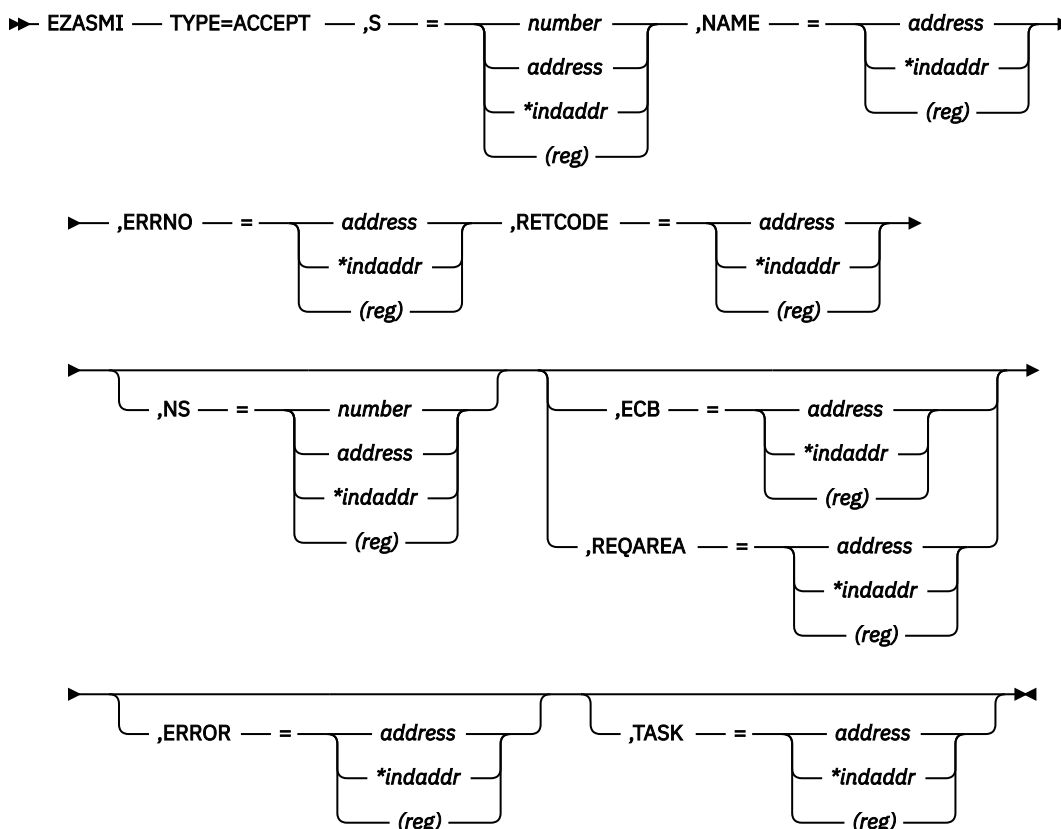
the ACCEPT macros. Concurrent server programs use the ACCEPT macro to pass connection requests to subtasks.

When issued, the ACCEPT macro takes the following actions:

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 macro and returns the address of the client for use by subsequent server macros. 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.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit.
	Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword	Description
---------	-------------

S	
----------	--

	Input parameter. A value or the address of a halfword binary number specifying the descriptor of the socket from which the connection is accepted.
--	--

NAME

	Output parameter. Initially, the IPv4 or IPv6 application provides a pointer to the IPv4 or IPv6 socket address structure, which is filled on completion of the call with the socket address of the connection peer. Include the SYS1.MACLIB(BPXYSOCK) macro to get the assembler mappings for the socket address structure. The socket address structure mappings begin at the SOCKADDR label. The AF_INET socket address structure fields start at the SOCK_SIN label. The AF_INET6 socket address structure fields start at the SOCK_SIN6 label.
--	---

The IPv4 socket address structure contains the following fields:

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

FAMILY	
---------------	--

	A halfword binary field specifying the IPv4 addressing family. For IPv4 the value is a decimal 2, indicating AF_INET.
--	---

PORT	
-------------	--

	A halfword binary field that is set to the client port number.
--	--

IPv4-ADDRESS	
---------------------	--

	A fullword binary field that is set to the 32-bit IPv4 Internet address, in network byte order, of the client host machine.
--	---

RESERVED	
-----------------	--

	Specifies eight bytes of binary zeros. This field is required, but not used.
--	--

The IPv6 socket address structure contains the following fields:

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

NAMELEN	
----------------	--

	A 1-byte binary field specifying the length of this IPv6 socket address structure. Any value specified by the use of this field is ignored when processed as input and the field is set to 0 when processed as output.
--	--

FAMILY	
---------------	--

	A 1-byte binary field specifying the IPv6 addressing family. For IPv6 the value is a decimal 19, indicating AF_INET6.
--	---

PORT	
-------------	--

	A halfword binary field that is set to the client port number.
--	--

FLOW-INFO	
------------------	--

	A fullword binary field specifying the traffic class and flow label. This value of this field is undefined.
--	---

IPv6-ADDRESS	
---------------------	--

	A 16-byte binary field that is set to the 128-bit IPv6 Internet address, in network byte order, of the client host machine.
--	---

SCOPE-ID	
-----------------	--

	A fullword binary field which identifies a set of interfaces as appropriate for the scope of the address carried in the IPv6-ADDRESS field. For a link scope IPv6-ADDRESS, SCOPE-ID contains the link index for the IPv6-ADDRESS. For all other address scopes, SCOPE-ID is undefined.
--	--

ERRNO	
--------------	--

	Output parameter. A fullword binary field. If RETCODE is negative, ERRNO contains a valid error number. Otherwise, ignore ERRNO .
--	--

See Appendix B, “Socket call error return codes,” on page 743 for information about **ERRNO** return codes.

RETCODE

Output parameter. If **RETCODE** is positive, **RETCODE** is the new socket number.

If **RETCODE** is negative, check **ERRNO** for an error number.

Value

Description

>0

Successful call.

-1

Check **ERRNO** for an error code.

NS

Input parameter. A value or the address of a halfword binary number specifying the descriptor number chosen for the new socket, which is the socket for the client at the time. If NS is not specified, the interface assigns it.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

BIND

In a server program, the BIND macro normally follows a SOCKET macro to complete the new socket creation process.

The BIND macro can 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 the socket address to use when issuing a CONNECT, SENDTO, or SENDMSG request.

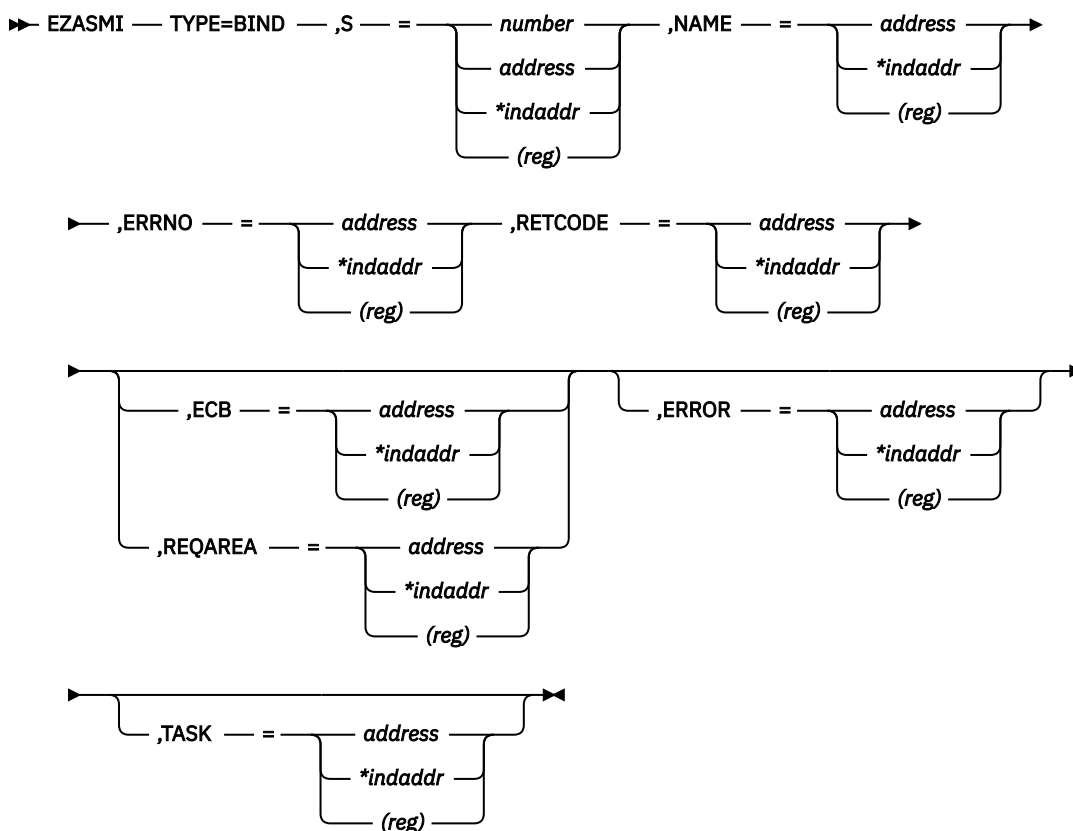
In addition to the port, the application also specifies an IP address on the BIND macro. Most applications typically specify a value of 0 for the IP address, which allows these applications to accept new TCP connections or receive UDP datagrams that arrive over any of the network interfaces of the local host. This enables client applications to contact the application using any of the IP addresses associated with the local host.

Alternatively, an application can indicate that it is interested in receiving new TCP connections or UDP datagrams that are targeted towards only a specific IP address associated with the local host. This can be accomplished by specifying the IP address in the appropriate field of the socket address structure passed on the NAME parameter.

Tip: Even if an application specifies the value 0 for the IP address on the BIND, the system administrator can override that value by specifying the BIND parameter on the PORT reservation statement in the TCP/IP profile. The effect of this override is similar to the effect of the application specifying an explicit IP address on the BIND macro. For more information, see [z/OS Communications Server: IP Configuration Reference](#).

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

S

Input parameter. A value or the address of a halfword binary number specifying the socket descriptor.

NAME

Input parameter. The IPv4 or IPv6 application provides a pointer to an IPv4 or IPv6 socket address structure. This structure specifies the port number and an IPv4 or IPv6 IP address from which the application can accept connections. Include the `SYS1.MACLIB(BPXYSOCK)` macro to get the assembler mappings for the socket address structure. The socket address structure mappings begin at the `SOCKADDR` label. The `AF_INET` socket address structure fields start at the `SOCK_SIN` label. The `AF_INET6` socket address structure fields start at the `SOCK_SIN6` label.

See Chapter 3, “Designing an iterative server program,” on page 23 for more information.

The IPv4 socket structure must specify the following fields:

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

FAMILY

A halfword binary field specifying the IPv4 addressing family. For IPv4 the value is a decimal 2, indicating `AF_INET`.

PORT

A halfword binary field set to the port number that binds to the socket. The application can call the `GETSOCKNAME` macro after the `BIND` macro to discover the assigned port number.

IPv4-ADDRESS

A fullword binary field that is set to the 32-bit IPv4 Internet address, in network byte order, of the host machine.

RESERVED

Specifies eight bytes of binary zeros. This field is required, but not used.

The IPv6 socket structure must specify the following fields:

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

NAMELEN

A 1-byte binary field specifying the length of this IPv6 socket address structure. Any value specified by the use of this field is ignored when processed as input and the field is set to 0 when processed as output.

FAMILY

A 1-byte binary field specifying the IPv6 addressing family. For IPv6 the value is a decimal 19, indicating `AF_INET6`.

PORT

A halfword binary field set to the port number that binds to the socket. The application can call the `GETSOCKNAME` macro after the `BIND` macro to discover the assigned port number.

FLOW-INFO

A fullword binary field specifying the traffic class and flow label. This field must be set to 0.

IPv6-ADDRESS

A 16-byte binary field that is set to the 128-bit IPv6 Internet address, in network byte order, of the host machine.

SCOPE-ID

A fullword binary field which identifies a set of interfaces as appropriate for the scope of the address carried in the `IPv6-ADDRESS` field. A value of 0 indicates that the `SCOPE-ID` field does not identify the set of interfaces to be used, and can be specified for any address types and scopes. For a link scope `IPv6-ADDRESS`, `SCOPE-ID` can specify a link index which identifies a set of interfaces. For all other address scopes, `SCOPE-ID` must be set to 0.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, **ERRNO** contains a valid error number. Otherwise, ignore **ERRNO**.

See Appendix B, “Socket call error return codes,” on page 743 for information about **ERRNO** return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check **ERRNO** for an error code.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

BIND2ADDRSEL

The BIND2ADDRSEL macro binds a socket to the local IP address that would be selected by the stack to communicate with the input destination IP address.

Use the BIND2ADDRSEL macro when the application must verify that the local IP address assigned by the stack meets its address selection criteria as specified by the IPV6_ADDR_PREFERENCES socket option before the stack sends any packets to the remote host. In a TCP or UDP application, the BIND2ADDRSEL macro usually follows the SETSOCKOPT macro with option IPV6_ADDR_PREFERENCES and precedes any communication with a remote host.

Result: The stack attempts to select a local IP address according to your application preferences. However, a successful BIND2ADDRSEL macro does not guarantee that all of your selection preferences of source IP address were met.

Guidelines:

- Use the SETSOCKOPT macro to set the IPV6_ADDR_PREFERENCES option to indicate your selection preferences of source IP address before binding the socket and before allowing an implicit bind of the socket to occur.

Result: If a socket has not been explicitly bound to a local IP address with a BIND or BIND2ADDRSEL macro when a CONNECT, SENDTO, or SENDMSG macro is issued, an implicit bind occurs. The stack chooses the local IP address used for outbound packets.

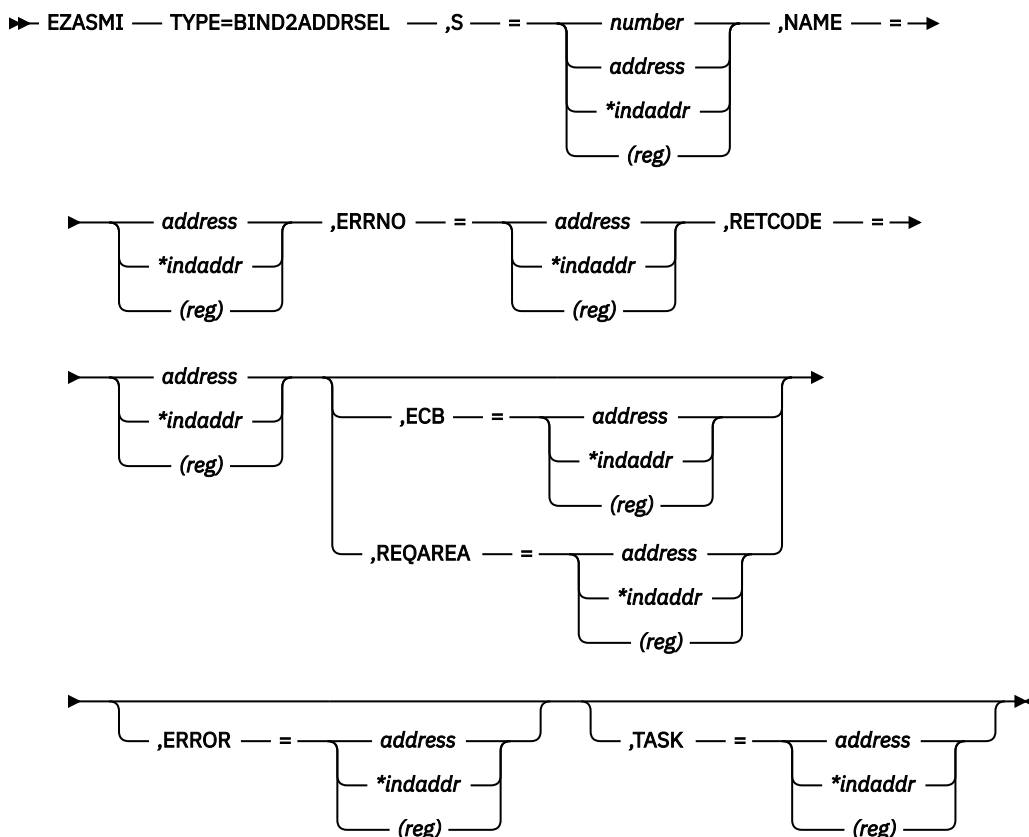
Requirement: When your application is using stream sockets, and must prevent the stack from sending any packets whatsoever (such as SYN) to the remote host before it can verify that the local IP address meets the values specified for the IPV6_ADDR_PREFERENCES option, do not allow the

CONNECT macro to implicitly bind the socket to a local IP address. Instead, bind the socket with the BIND2ADDRSEL macro and test the local IP address assigned with the INET6_IS_SRCADDR macro. If the assigned local IP address is satisfactory, you can then use the CONNECT macro to establish communication with the remote host.

- After you successfully issue the BIND2ADDRSEL macro, use the GETSOCKNAME macro to obtain the local IP address that is bound to the socket. When the local IP address is obtained, use the INET6_IS_SRCADDR macro to verify that the local IP address meets your address selection criteria.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

S

Input parameter. A value or the address of a halfword binary number that specifies the socket descriptor.

Requirements: The socket must be an AF_INET6 socket. The type can be SOCK_STREAM or SOCK_DGRAM.

NAME

Input parameter. The application provides a pointer to an AF_INET6 socket address structure. You can specify an IPv4 address by using its IPv4-mapped IPv6 format.

Guidelines:

- Include the SYS1.MACLIB(BPXYSOCK) macro to get the assembler mappings for the socket address structure.
- Begin the socket address structure mappings at the SOCKADDR label.
- Begin the AF_INET6 socket address structure fields at the SOCK_SIN6 label.

The IPv6 socket structure must specify the following fields:

Field

Description

FAMILY

A halfword binary field that specifies the IPv6 addressing family. For IPv6 the value is the decimal value 19, indicating AF_INET6.

PORT

A halfword binary field. This field is ignored by BIND2ADDRSEL processing.

Guideline: The application can call the GETSOCKNAME macro after the BIND2ADDRSEL macro to discover the assigned port number.

FLOW-INFO

A fullword binary field that specifies the traffic class and flow label. This field is ignored by BIND2ADDRSEL processing.

IPv6-ADDRESS

A 16-byte binary field that is set to the 128-bit IPv6 Internet address, in network byte order, of the remote host machine that the application communicates with.

Rule: Specify an IPv4 address by using its IPv4-mapped IPv6 format.

SCOPE-ID

A fullword binary field that identifies a set of interfaces as appropriate for the scope of the address that is specified in the **IPv6-ADDRESS** field. The value 0 indicates that the **SCOPE-ID** field does not identify the set of interfaces to be used.

Requirement: The **SCOPE-ID** field must be nonzero if the address is a link-local address. For all other address scopes, the **SCOPE-ID** value must be set to 0.

ERRNO

Output parameter. A fullword binary field. If the **RETCODE** value is negative, the **ERRNO** field contains a valid error number; otherwise, ignore **ERRNO**.

See [Appendix B, "Socket call error return codes," on page 743](#) for information about **ERRNO** return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check **ERRNO** for an error code.

ECB or REQAREA

Input parameter. This parameter is required if you use APITYPE=3. This parameter points to a 104-byte field containing one of the following values:

For ECB

A 4-byte **ECB** that is posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field that is used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

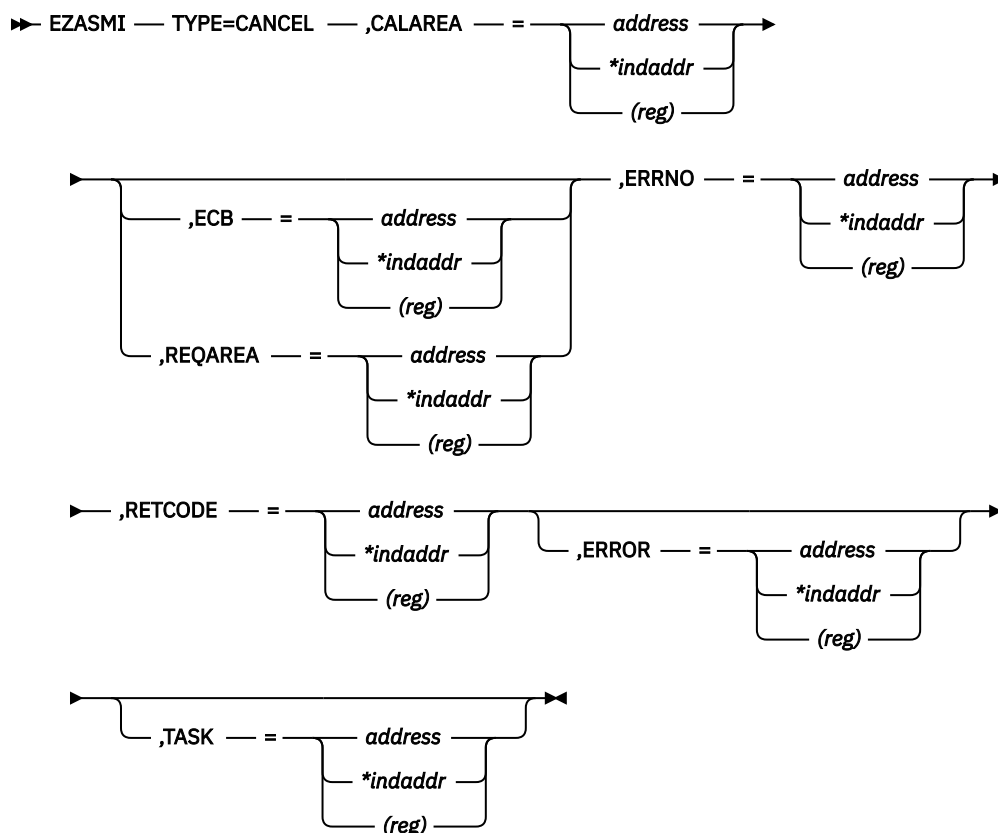
Input parameter. The location of the task storage area in your program.

CANCEL

The CANCEL function terminates a call in progress. The call being canceled must have specified **ECB** or **REQAREA**.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

CALAREA

Input parameter. The **ECB** or **REQAREA** specified in the call being canceled.

Note: To be compatible with TCP/IP for MVS V3R1, **CALAREA** can be specified as **CALLAREA**.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted, or the asynchronous exit has been driven.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, this contains an error number.

RETCODE

Output parameter. A fullword binary field. If **RETCODE** is 0, the CANCEL was successful.

Value

Description

0

Successful call.

-1

Check **ERRNO** for an error code.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

CLOSE

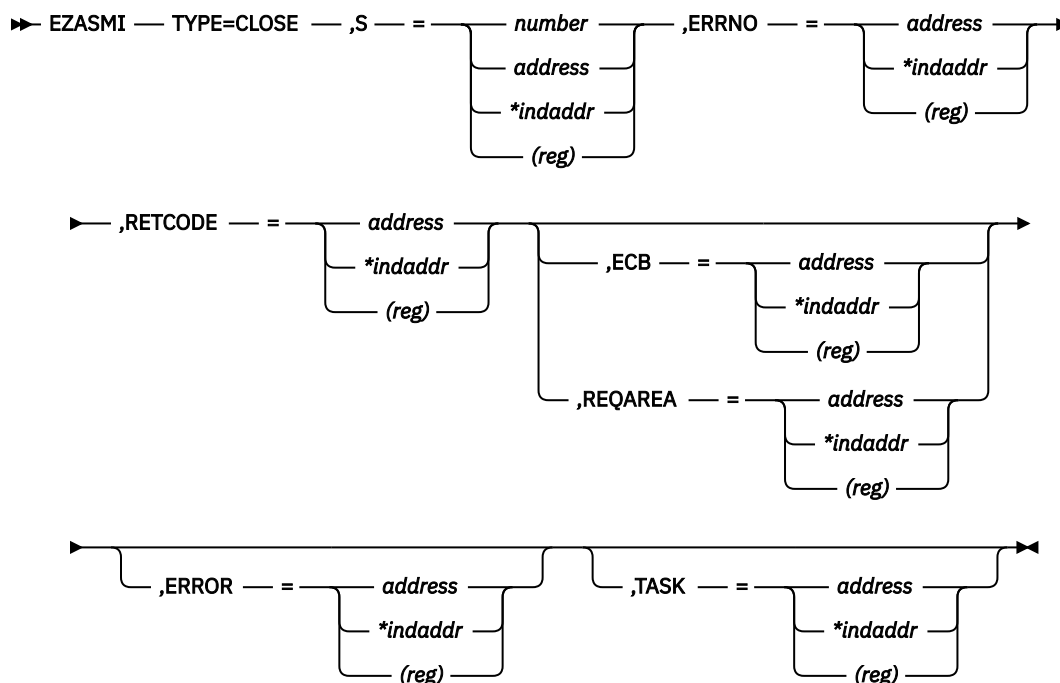
The CLOSE macro shuts down the socket and frees the resources that are allocated to the socket. Issue the SHUTDOWN macro before you issue the CLOSE macro.

CLOSE can also be issued by a concurrent server after it gives a socket to a subtask program. After issuing GIVESOCKET and receiving notification that the client child has successfully issued TAKESOCKET, the concurrent server issues the CLOSE macro to complete the transfer of ownership.

Note: If a stream socket is closed while input or output data is queued, the stream connection is reset and data transmission can be incomplete. SETSOCKOPT can be used to set a SO_LINGER condition, in which TCP/IP continues to send data for a specified period of time after the CLOSE macro is issued. For information about SO_LINGER, see [“SETSOCKOPT” on page 497](#).

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389 .
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

S

Input parameter. A value or the address of a halfword binary number specifying the socket to be closed.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, **ERRNO** contains a valid error number. Otherwise, ignore **ERRNO** field.

See [Appendix B, "Socket call error return codes," on page 743](#) for information about **ERRNO** return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check **ERRNO** for an error code.

ECB or REQAREA

Input parameter. This parameter is required if you are using **APITYPE=3**. It points to a 104-byte field containing:

For ECB

A 4-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

CONNECT

The CONNECT macro is used by a client to establish a connection between a local socket and a remote socket.

For stream sockets, the CONNECT macro:

- Completes the binding process for a stream socket if BIND has not been previously issued.
- Attempts connection to a remote socket. This connection must be completed before data can be transferred.

For datagram sockets, CONNECT is not essential, but you can use it to send messages without specifying the destination.

For both types of sockets, the following CONNECT macro sequence applies:

1. The server issues BIND and LISTEN (stream sockets only) to create a passive open socket.
2. The client issues CONNECT to request a connection.
3. The server creates a new connected socket by accepting the connection on the passive open socket.

If the socket is in blocking mode, CONNECT blocks the calling program until the connection is established or until an error is received.

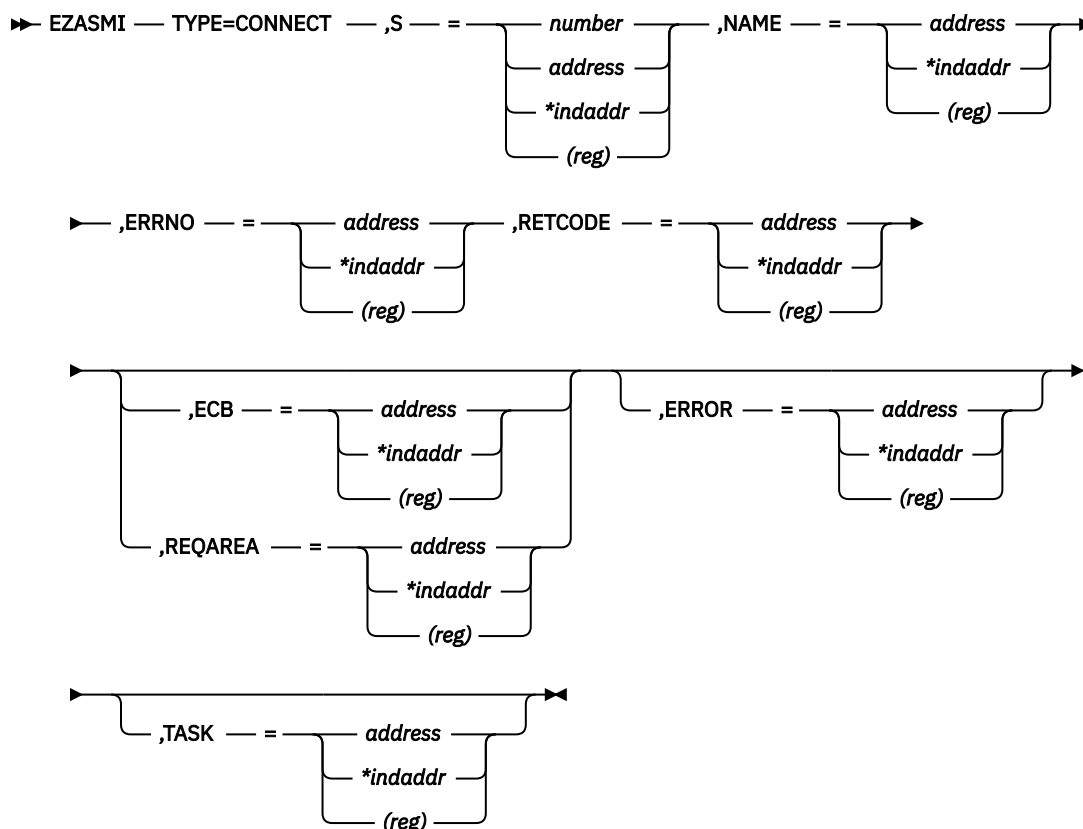
If the socket is in nonblocking mode, the return code indicates the success of the connection request.

- A 0 RETCODE indicates that the connection was completed.
- A nonzero RETCODE with an ERRNO of 36 (EINPROGRESS) indicates that the connection could not be completed, but since the socket is nonblocking, the CONNECT macro completes its processing.

The caller must test the completion of the connection setup by calling SELECT and testing for the ability to write to the socket. The completion cannot be checked by issuing a second CONNECT.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

S

Input parameter. A value or the address of a halfword binary number specifying the socket descriptor.

NAME

Input parameter. The NAME parameter for CONNECT specifies the IPv4 or IPv6 socket address of the IPv4 or IPv6 IP connection peer. Include the SYS1.MACLIB(BPXYSOCK) macro to get the assembler mappings for the socket address structure. The socket address structure mappings begin at the SOCKADDR label. The AF_INET socket address structure fields start at the SOCK_SIN label. The AF_INET6 socket address structure fields start at the SOCK_SIN6 label.

The IPv4 socket structure must specify the following fields:

Field Description

FAMILY

A halfword binary field specifying the IPv4 addressing family. For IPv4 the value is always a decimal 2, indicating AF_INET.

PORT

A halfword binary field that is set to the server port number in network byte order. For example, if the port number is 5000 in decimal, it is set to X'1388'.

IPv4-ADDRESS

A fullword binary field specifying the 32-bit IPv4 Internet Protocol address, in network byte order, of the server host machine. For example, if the Internet Protocol address is 129.4.5.12 in dotted decimal notation, it is set to X'8104050C'.

RESERVED

Specifies 8 bytes of binary zeros. This field is required, but not used.

The IPv6 socket structure must specify the following fields:

Field

Description

NAMELEN

A 1-byte binary field specifying the length of this IPv6 socket address structure. Any value specified by the use of this field is ignored when processed as input and the field is set to 0 when processed as output.

FAMILY

A 1-byte binary field specifying the IPv6 addressing family. For IPv6 the value is a decimal 19, indicating AF_INET6.

PORT

A halfword binary field that is set to the port number in network byte order. For example, if the port number is 5000 in decimal, it is set to X'1388'.

FLOW-INFO

A fullword binary field specifying the traffic class and flow label. This field must be set to 0.

IPv6-ADDRESS

A 16-byte binary field that is set to the 128-bit IPv6 Internet Protocol address, in network byte order, of the client host machine. For example, if the IPv6 Internet Protocol address is 12ab:0:0:cd30:123:4567:89AB:cedf in colon-hexadecimal notation, it is set to X'12AB00000000CD300123456789ABCDEF'.

SCOPE-ID

A fullword binary field which identifies a set of interfaces as appropriate for the scope of the address carried in the IPv6-ADDRESS field. A value of 0 indicates the SCOPE-ID field does not identify the set of interfaces to be used, and can be specified for any address types and scopes. For a link scope IPv6-ADDRESS, SCOPE-ID can specify a link index which identifies a set of interfaces. For all other address scopes, SCOPE-ID must be set to 0.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, **ERRNO** contains a valid error number. Otherwise, ignore **ERRNO**.

See [Appendix B, "Socket call error return codes," on page 743](#) for information about **ERRNO** return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check **ERRNO** for an error code.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

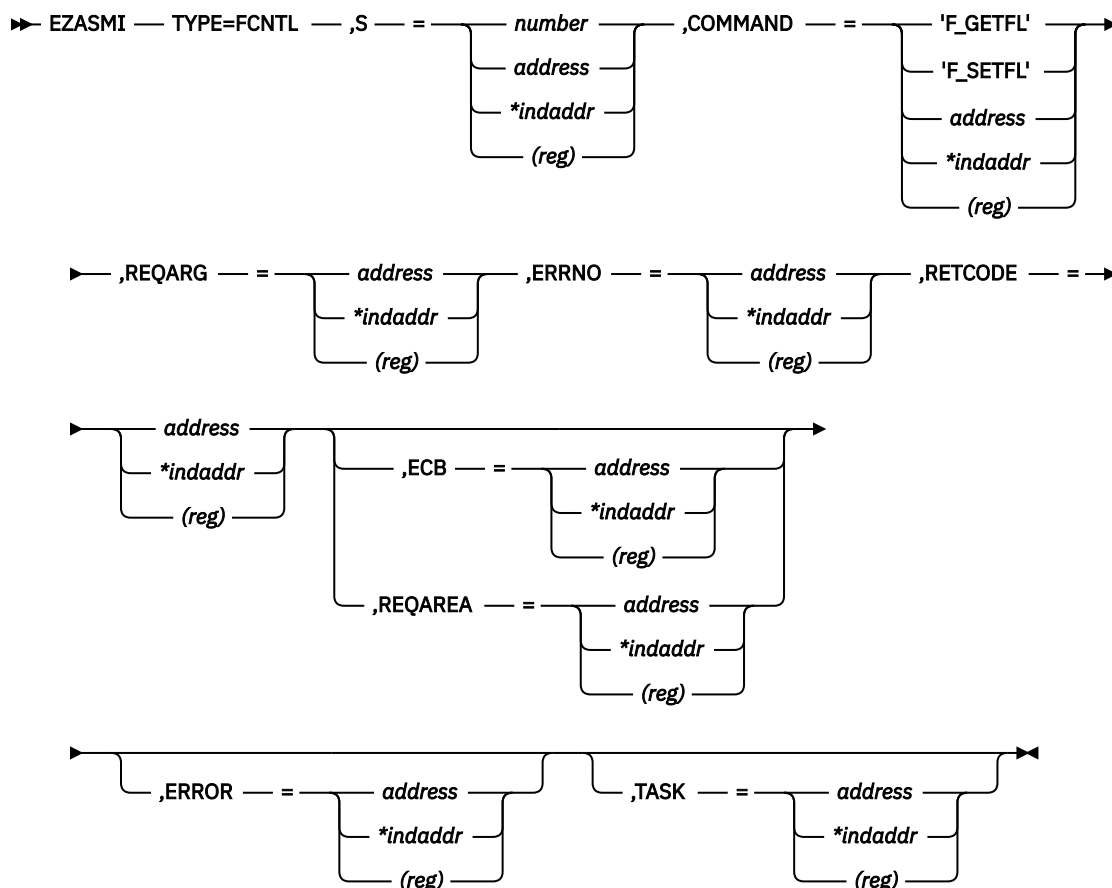
FCNTL

The blocking mode for a socket can be queried or set to nonblocking using the FNDELAY flag. You can query or set the FNDELAY flag even though it is not defined in your program.

See [“IOCTL” on page 455](#) for another way to control socket blocking.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389 .
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

S

Input parameter. A value or the address of a halfword binary number specifying the socket descriptor for the socket that you want to unblock or query.

COMMAND

Input parameter. A fullword binary field or a literal that sets the FNDELAY flag to one of the following values:

Value

Description

3 or 'F_GETFL'

Query the blocking mode for the socket.

4 or 'F_SETFL'

Set the mode to nonblocking for the socket. **REQARG** is set by TCP/IP.

The FNDELAY flag sets the nonblocking mode for the socket. If data is not present on calls that can block (READ, READV, and RECV), the call returns a -1, and **ERRNO** is set to 35 (EWOULDBLOCK).

Note: Values for **COMMAND** that are supported by the z/OS UNIX System Services FCNTL callable service are supported also. see [z/OS UNIX System Services Programming: Assembler Callable Services Reference](#) for more information.

REQARG

A fullword binary field containing a mask that TCP/IP uses to set the FNDELAY flag.

- If **COMMAND** is set to 3 (query), the **REQARG** field should be set to 0.
- If **COMMAND** is set to 4 (set),
 - Set **REQARG** to 4 to turn the FNDELAY flag on. This places the socket in nonblocking mode.

- Set **REQARG** to 0 to turn the FNDELAY flag off. This places the socket in blocking mode.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, **ERRNO** contains a valid error number. Otherwise, ignore the **ERRNO** field.

See Appendix B, “Socket call error return codes,” on page 743 for information about **ERRNO** return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

- If **COMMAND** was set to 3 (query), a bit string is returned.
 - If **RETCODE** contains X'00000004', the socket is nonblocking. The FNDELAY flag is on.
 - If **RETCODE** contains X'00000000', the socket is blocking. The FNDELAY flag is off.
- If the **COMMAND** field was 4 (set), a successful call returns 0 in **RETCODE**. For either **COMMAND**, a **RETCODE** of -1 indicates an error. Check **ERRNO** for the error number.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

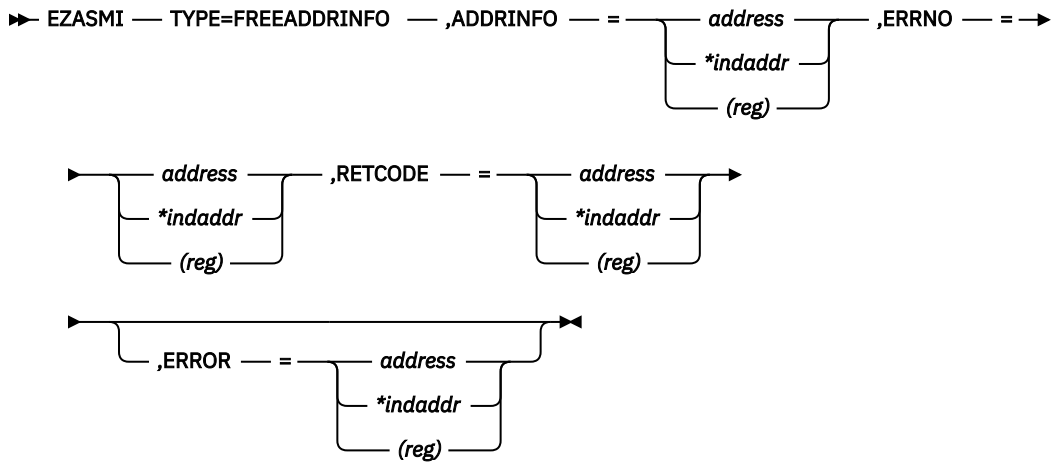
FREEADDRINFO

The FREEADDRINFO macro frees all the address information structures returned by GETADDRINFO in the RES parameter.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.

Control parameters:	All parameters must be addressable by the caller and in the primary address space.
---------------------	--



Keyword
Description

ADDRINFO

Input parameter. The address of a set of address information structures returned by **TYPE=GETADDRINFO RES** argument.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, **ERRNO** contains a valid error number. Otherwise, ignore the **ERRNO** field.

See [Appendix B, “Socket call error return codes,” on page 743](#) for information about **ERRNO** return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value
Description

0

Successful call.

-1

Check **ERRNO** for an error code.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

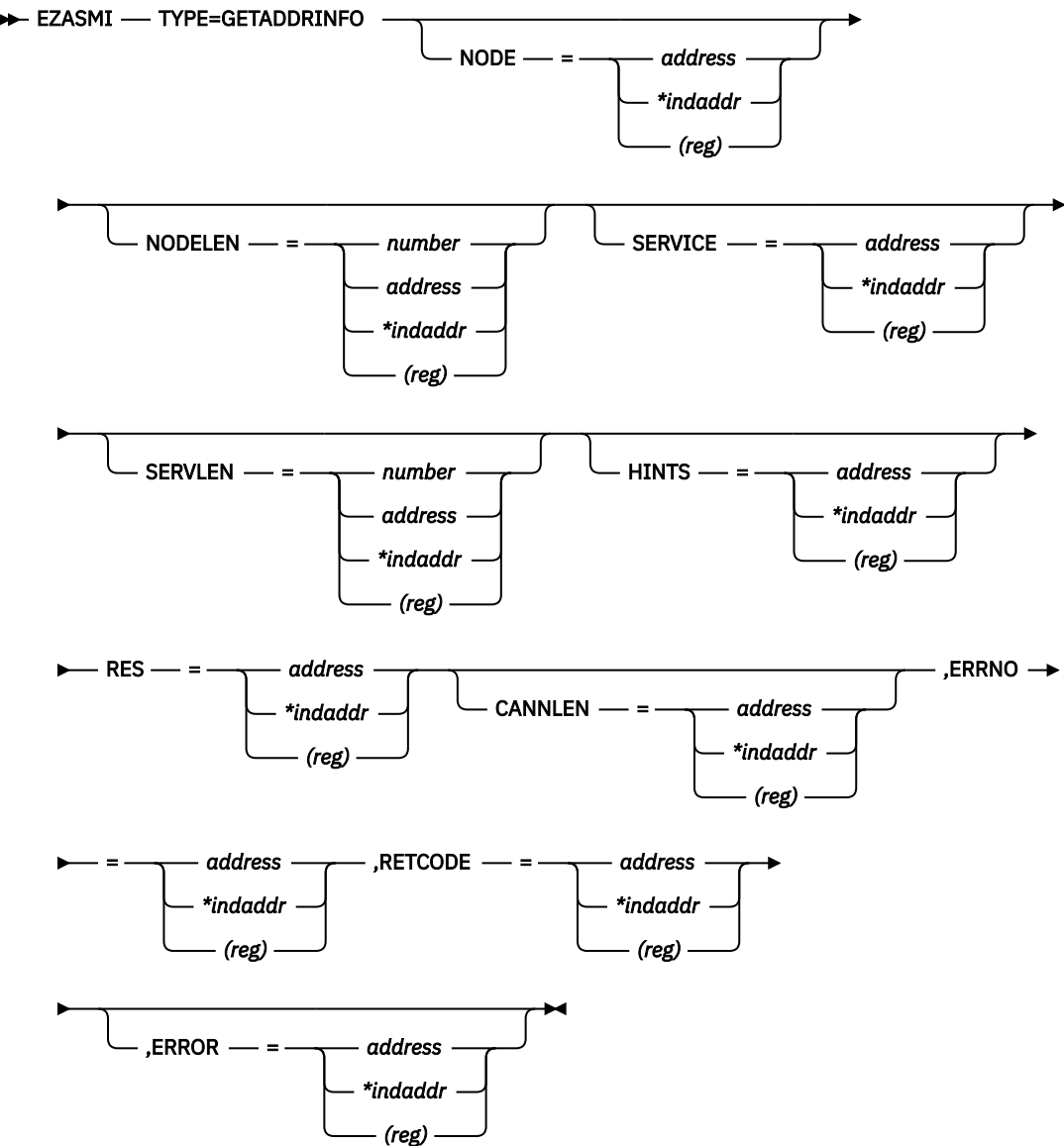
GETADDRINFO

The GETADDRINFO macro translates either the name of a service location (for example, a host name), a service name, or both, and returns a set of socket addresses and associated information to be used in creating a socket with which to address the specified service or sending a datagram to the specified service.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.

Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword
Description

NODE
An input parameter. Storage up to 255 bytes long that contains the host name being queried. If the AI_NUMERICHOST flag is specified in the storage pointed to by the HINTS operand, then NODE should contain the queried host's IP address in network byte order presentation form. This is an optional

field, but if specified you must also code NODELEN. The NODE name being queried consists of up to NODELEN or up to the first binary zero.

You can append scope information to the host name by using the format *node%scope information*. The combined information must be 255 bytes or less. For more information, see [z/OS Communications Server: IPv6 Network and Appl Design Guide](#).

NODELEN

An input parameter. A fullword binary field set to the length of the host name specified in the NODE field and should not include extraneous blanks. This is an optional field, but if specified you must also code NODE.

SERVICE

An input parameter. Storage up to 32 bytes long that contains the service name being queried. If the AI_NUMERICSERV flag is specified in the storage pointed to by the HINTS operand, then SERVICE should contain the queried port number in presentation form. This is an optional field, but if specified you must also code SERVLN. The SERVICE name being queried consists of up to SERVLN or up to the first binary zero.

SERVLN

An input parameter. A fullword binary field set to the length of the service name specified in the SERVICE field and should not include extraneous blanks. This is an optional field but if specified you must also code SERVICE.

HINTS

An input parameter. If the HINTS argument is specified, then it contains the address of an addrinfo structure containing input values that can direct the operation by providing options and limiting the returned information to a specific socket type, address family, or protocol. If the HINTS argument is not specified, then the information returned is as if it referred to a structure containing the value 0 for the FLAGS, SOCTYPE and PROTO fields, and AF_UNSPEC for the AF field. Include the EZBREHST Resolver macro to enable your program to contain the assembler mappings for the ADDR_INFO structure.

This is an optional field.

The address information structure has the following fields:

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

FLAGS

A fullword binary field. Must have the value of 0 or the bitwise OR of one or more of the following values:

AI_PASSIVE (X'00000001')

- Specifies how to specify the **NAME** pointed to in the returned **RES**. If this flag is specified, then the returned address information is suitable for use in binding a socket for accepting incoming connections for the specified service (that is the **TYPE=BIND** call). In this case, if the **NODE** argument is not specified, then the IP address portion of the socket address structure pointed to by the returned **RES** is set to **INADDR_ANY** for an IPv4 address or to the IPv6 unspecified address (in6addr_any) for an IPv6 address.
- If this flag is not set, the returned address information is suitable for the **TYPE=CONNECT** call (for a connection-mode protocol) or for a **TYPE=CONNECT, TYPE=SENDTO**, or **TYPE=SENDMSG** call (for a connectionless protocol). In this case, if the **NODE** argument is not specified, then the IP address portion of the socket address structure pointed to by the returned **RES** is set to the default loopback address for an IPv4 address (127.0.0.1) or the default loopback address for an IPv6 address (::1).
- This flag is ignored if the **NODE** argument is specified.

AI_CANONNAMEOK (X'00000002')

- If this flag is specified and the **NODE** argument is specified, then the **TYPE=GETADDRINFO** call attempts to determine the canonical name corresponding to the **NODE** argument.

AI_NUMERICHOST (X'00000004')

- If this flag is specified then the **NODE** argument must be a numeric host address in presentation form. Otherwise, an error of host not found [**EAI_NONAME**] is returned.

AI_NUMERICSERV (X'00000008')

- If this flag is specified then the **SERVICE** argument must be a numeric port in presentation form. Otherwise, an error [**EAI_NONAME**] is returned.

AI_V4MAPPED (X'00000010')

- If this flag is specified along with the **AF** field with the value of **AF_INET6**, or a value of **AF_UNSPEC** when IPv6 is supported on the system, then the caller accepts IPv4-mapped IPv6 addresses.
 - If the **AF** field is **AF_INET6**, a query for IPv4 addresses is made if the **AI_ALL** flag is specified or if no IPv6 addresses are found. Any IPv4 addresses that are found are returned as IPv4-mapped IPv6 addresses.
 - If the **AF** field is **AF_UNSPEC**, queries are made for both IPv6 and IPv4 addresses. If IPv4 addresses are found and IPv6 is supported, the IPv4 addresses are returned as IPv4-mapped IPv6 addresses.
- Otherwise, this flag is ignored.

AI_ALL (X'00000020')

- If the **AF** field has a value of **AF_INET6** and **AI_ALL** is set, the **AI_V4MAPPED** flag must also be set to indicate that the caller accepts all addresses: IPv6 and IPv4-mapped IPv6 addresses.
- If the **AF** field has a value of **AF_UNSPEC**, **AI_ALL** is accepted, but has no impact on the processing. No matter if **AI_ALL** is specified or not, the caller accepts both IPv6 and IPv4 addresses. A query is first made for IPv6 addresses and if successful, the IPv6 addresses are returned. Another query is then made for IPv4 addresses:
 - If the **AI_V4MAPPED** flag is also specified and the system supports IPv6, the IPv4 addresses are returned as IPv4-mapped IPv6 addresses.
 - If the **AI_V4MAPPED** flag is not specified or the system does not support IPv6, the IPv4 addresses are returned as IPv4 addresses.
- Otherwise, the flag is ignored.

AI_ADDRCONFIG (X'00000040')

If this flag is specified, then a query for IPv6 on the **NODE** will occur if the Resolver determines whether either of the following conditions is true:

- If the system is IPv6 enabled and has at least one IPv6 interface, the Resolver makes a query for IPv6 (AAAA or A6 DNS) records.
- If the system is IPv4 enabled and has at least one IPv4 interface, the Resolver makes a query for IPv4 (A DNS) records.

The loopback address is not considered in this case as a valid interface.

AI_EXTFLAGS (X'00000080') or a decimal value of 128

Requests the extended form of the getaddrinfo function. The extended form allows for additional hints to be passed to the resolver for determining the order of destination addresses that is returned. If this flag is specified, the **EFLAGS** field is required.

AF

A fullword binary field. Used to limit the returned information to a specific address family. The value of **AF_UNSPEC** means that the caller accepts any protocol family. The value of a decimal 0 indicates **AF_UNSPEC**. The value of a decimal 2 indicates **AF_INET**, and the value of a decimal 19 indicates **AF_INET6**.

SOCTYPE

A fullword binary field. Used to limit the returned information to a specific socket type. A value of 0 means that the caller accepts any socket type. If a specific socket type is not given (for example, a value of 0), then information on all supported socket types is returned.

The following table shows the acceptable socket types:

Type name	Decimal value	Description
SOCK_STREAM	1	for stream socket
SOCK_DGRAM	2	for datagram socket
SOCK_RAW	3	for raw-protocol interface

Anything else will fail with return code **EAI_SOCKTYPE**. Note that although **SOCK_RAW** is accepted, it is valid only when **SERVICE** is numeric (for example, **SERVICE=23**). A lookup for a **SERVICE** name will never occur in the appropriate services file (for example, **hlq.ETC.SERVICES**) using any protocol value other than SOCK_STREAM or SOCK_DGRAM.

If **PROTO** is not 0 and **SOCTYPE** is 0, then the only acceptable input values for **PROTO** are **IPPROTO_TCP** and **IPPROTO_UDP**. Otherwise, the **TYPE=GETADDRINFO** fails with return code **EAI_BADFLAGS**.

If **SOCTYPE** and **PROTO** are both specified as 0 then **TYPE=GETADDRINFO** proceeds as follows:

- If **SERVICE** is null, or if **SERVICE** is numeric, then any returned addrinfos defaults to a specification of **SOCTYPE** as **SOCK_STREAM**.
- If **SERVICE** is specified as a service name (for example, **SERVICE=FTP**), then **TYPE=GETADDRINFO** searches the appropriate services file (such as, **hlq.ETC.SERVICES**) twice. The first search uses SOCK_STREAM as the protocol, and the second search uses SOCK_DGRAM as the protocol. No default socket type is provided in this case.

If both **SOCTYPE** and **PROTO** are specified as a value other than 0 then they should be compatible, regardless of the value specified by **SERVICE**. In this context, *compatible* can have one of the following meanings:

- **SOCTYPE=SOCK_STREAM** and **PROTO=IPPROTO_TCP**
- **SOCTYPE=SOCK_DGRAM** and **PROTO=IPPROTO_UDP**
- **SOCTYPE=SOCK_RAW** and **PROTO** can be anything

PROTO

A fullword binary field. Used to limit the returned information to a specific protocol. A value of 0 means that the caller accepts any protocol.

The following table shows the acceptable protocols:

Protocol name	Decimal value	Description
IPPROTO_TCP	6	TCP
IPPROTO_UDP	17	user datagram

If **SOCKETTYPE** is 0 and **PROTO** is not 0, then the only acceptable input values for **PROTO** are **IPPROTO_TCP** and **IPPROTO_UDP**. Otherwise, the **TYPE=GETADDRINFO** is failed with return code **EAI_BADFLAGS**.

If **PROTO** and **SOCKETTYPE** are both specified as 0, then **TYPE=GETADDRINFO** proceeds as follows:

- If **SERVICE** is null, or if **SERVICE** is numeric, then any returned addrinfos default to a specification of **SOCKETTYPE** as **SOCK_STREAM**.
- If **SERVICE** is specified as a service name (for example, **SERVICE=FTP**), then **TYPE=GETADDRINFO** searches the appropriate services file (such as, **hlq.ETC.SERVICES**) twice.

The first search uses SOCK_STREAM as the protocol, and the second search uses SOCK_DGRAM as the protocol. No default socket type is provided in this case.

If both PROTO and SOCKTYPE are specified as nonzero, then they should be compatible, regardless of the value specified by SERVICE. In this context, compatible can have one of the following meanings:

- SOCKTYPE=SOCK_STREAM and PROTO=IPPROTO_TCP
- SOCKTYPE=SOCK_DGRAM and PROTO=IPPROTO_UDP
- SOCKTYPE=SOCK_RAW and PROTO can be anything

If the lookup for the value specified in SERVICE fails [that is, the service name does not appear in the appropriate services file (for example, **hlq.ETC.SERVICES**) using the input protocol], then the TYPE=GETADDRINFO fails with return code EAI_SERVICE.

NAMELEN

A fullword binary field followed by 8 padding bytes. This field must be 0.

CANONNAME

A fullword binary field followed by 4 padding bytes. This field must be 0.

NAME

A fullword binary field followed by 4 padding bytes. This field must be 0.

NEXT

A fullword binary field. This field must be 0.

EFLAGS

A fullword binary field that specifies the source IPv6 address selection preferences. This field is required if AI_EXTFLAGS is specified in the FLAGS field.

This field must contain the value 0 or the bitwise OR of one or more of the following values:

IPv6_PREFER_SRC_HOME (X'00000001') or the decimal value 1

Indicates that home source IPv6 addresses are preferred over care-of source IPv6 addresses.

IPv6_PREFER_SRC_COA (X'00000002') or the decimal value 2

Indicates that care-of source IPv6 addresses are preferred over home source IPv6 addresses.

IPv6_PREFER_SRC_TMP (X'00000004') or the decimal value 4

Indicates that temporary source IPv6 addresses are preferred over public source IPv6 addresses.

IPv6_PREFER_SRC_PUBLIC (X'00000008') or the decimal value 8

Indicates that public source IPv6 addresses are preferred over temporary source IPv6 addresses.

IPv6_PREFER_SRC_CGA (X'00000010') or the decimal value 16

Indicates that cryptographically generated source IPv6 addresses are preferred over non-cryptographically generated source IPv6 addresses.

IPv6_PREFER_SRC_NONCGA (X'00000020') or the decimal value 32

Indicates that non-cryptographically generated source IPv6 addresses are preferred over cryptographically generated source IPv6 addresses.

If contradictory EFLAGS (for example, IPv6_PREFER_SRC_TMP and IPv6_PREFER_SRC_PUBLIC) or invalid EFLAGS (for example, X'00000040' or a decimal value 64) are specified, then the GETADDRINFO call fails with RETCODE -1 and ERRNO EAI_BADEXTFLAGS (decimal value 11).

RES

Initially a fullword binary field. On a successful return, this field contains a pointer to a chain of one or more address information structures. Use the EZBREHST macro to establish address information mapping. The structures are allocated in the key of the calling application. The structures that are returned by a TYPE=GETADDRINFO call are serially reusable storage for the z/OS UNIX process. They can be used or referenced between process threads, but should not be used or referenced between processes. When you finish using the structures, explicitly release their storage by specifying the returned pointer on a TYPE=FREEADDRINFO call.

The address information structure contains the following fields. All fields in this structure that are not specified with an explicit value are set to 0:

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

FLAGS

A fullword binary field that is not used as output.

AF

A fullword binary field. The value returned in this field can be used as the AF= argument on the TYPE=SOCKET macro to create a socket suitable for use with the returned address NAME.

SOCTYPE

A fullword binary field. The value returned in this field can be used as the SOCTYPE= argument on the TYPE=SOCKET macro to create a socket suitable for use with the returned address NAME.

PROTO

A fullword binary field. The value returned in this field can be used as the PROTO= argument on the TYPE=SOCKET macro to create a socket suitable for use with the returned address NAME.

NAMELEN

A fullword binary field followed by 8 padding bytes. The length of the NAME socket address structure. The value returned in this field can be used as the arguments for the TYPE=CONNECT or TYPE=BIND macros with such a socket, according to the AI_PASSIVE flag.

CANONNAME

A fullword binary field followed by 4 padding bytes. The address of storage containing the canonical name for the value specified by NODE. Initially, this field must be 0. If the NODE argument is specified, and if the AI_CANONNAMEOK flag was specified by the HINTS argument, then the CANONNAME field in the first returned address information structure contains the address of storage containing the canonical name corresponding to the input NODE argument. If the canonical name is not available, then the CANONNAME field refers to the NODE argument or a string with the same contents. The CANNLEN field contains the length of the returned canonical name.

NAME

A fullword binary field followed by 4 padding bytes. The address of the returned socket address structure. The value returned in this field can be used as the arguments for the TYPE=CONNECT or TYPE=BIND macros with such a socket, according to the AI_PASSIVE flag.

NEXT

A fullword binary field. Contains the address of the next address information structure on the list, or 0's if it is the last structure on the list.

EFLAGS

A fullword binary field that is not used as output.

CANNLEN

Initially an input parameter. A fullword binary field used to contain the length of the canonical name returned by the RES CANONNAME field. This is an optional field.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, **ERRNO** contains a valid error number. Otherwise, ignore the **ERRNO** field.

See [Appendix B, "Socket call error return codes," on page 743](#) for information about **ERRNO** return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value	Description
-------	-------------

0	Successful call.
---	------------------

-1

Check **ERRNO** for an error code.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

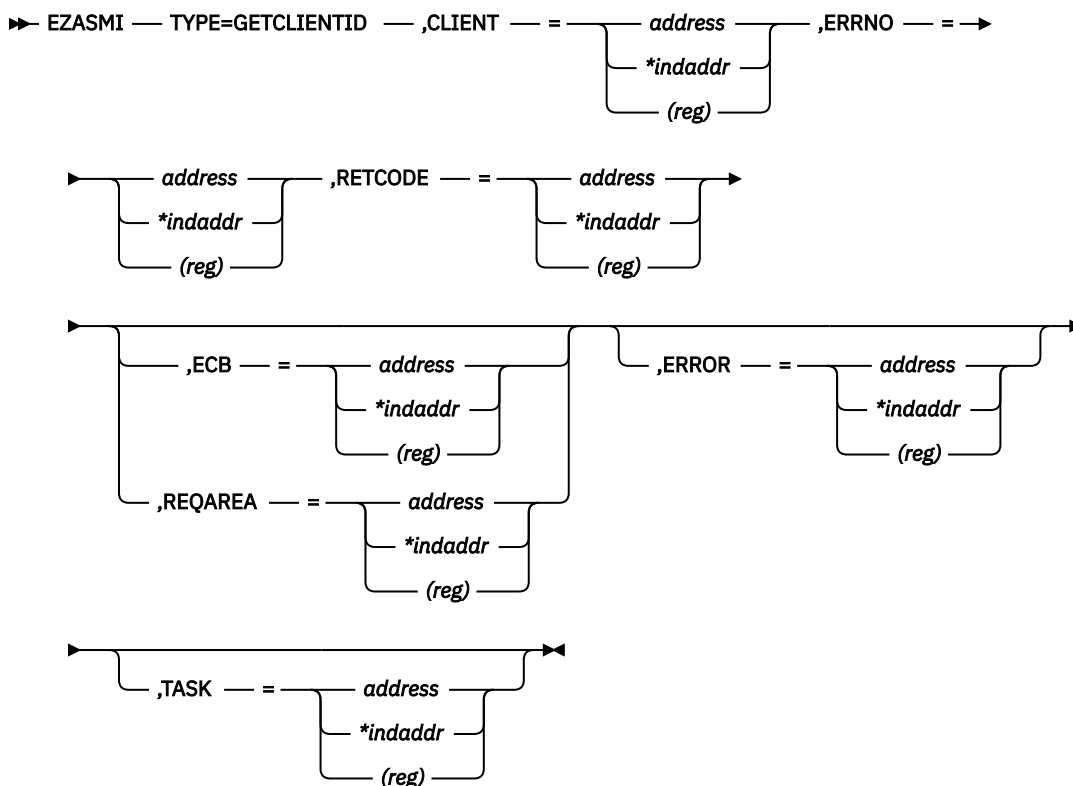
GETCLIENTID

The GETCLIENTID macro returns the identifier by which the calling application is known to the TCP/IP address space. The client ID structure returned is used by the GIVESOCKET and TAKESOCKET macros.

When GETCLIENTID is called by a server or client, the identifier of the calling application is returned.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword
Description

CLIENT

Input/Output parameter. A client ID structure describing the identifier for your application, regardless whether a server or client.

Field
Description

DOMAIN

A fullword binary number specifying the domain of the client. On input, this is an optional parameter for AF_INET, and a required parameter for AF_INET6 to specify the domain of the client. For TCP/IP, the value is a decimal 2 indicating AF_INET, or decimal 19 indicating AF_INET6. On output, this is the returned domain of the client.

NAME

An 8-byte character field that is filled, on completion of the call, with the client address space identifier.

TASK

Output parameter. An 8-byte field set to the client task identifier.

RESERVED

Output parameter. Specifies 20 bytes of binary zeros. This field is required, but it is not used.

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, ERRNO contains a valid error number. Otherwise, ignore ERRNO.

See Appendix B, “Socket call error return codes,” on page 743 for information about ERRNO return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value
Description

0

Successful call.

-1

Check ERRNO for an error code.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte ECB posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the ECB has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

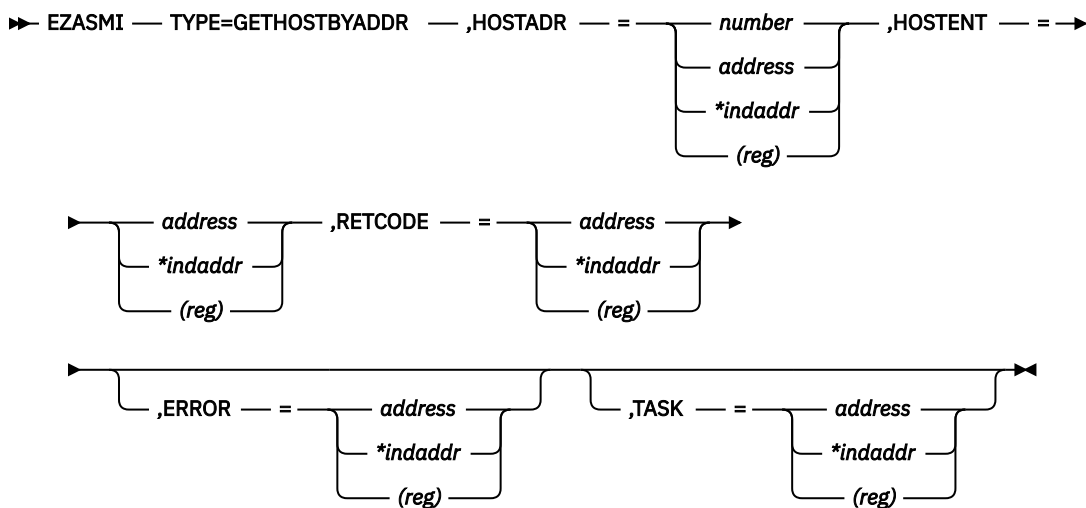
Input parameter. The location of the task storage area in your program.

GETHOSTBYADDR

The GETHOSTBYADDR macro returns domain and alias names of the host whose IPv4 Internet address is specified by the macro. A TCP/IP host can have multiple alias names and host IPv4 Internet addresses.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state. The PSW key must match the key in which the MVS application task was attached.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389 .
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Note: The storage for the HOSTENT structure returned by this call is released during TERMAPI processing; therefore, the application program must not use the HOSTENT storage after the TERMAPI.

Keyword

Description

HOSTADR

Input parameter. A fullword unsigned binary field set to the Internet address of the host whose name you want to find.

HOSTENT

Input parameter. A fullword containing the address of the HOSTENT structure returned by the macro. For information about the HOSTENT structure, see Figure 64 on page 231.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value	Description
>0	Successful call.
-1	An error occurred.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

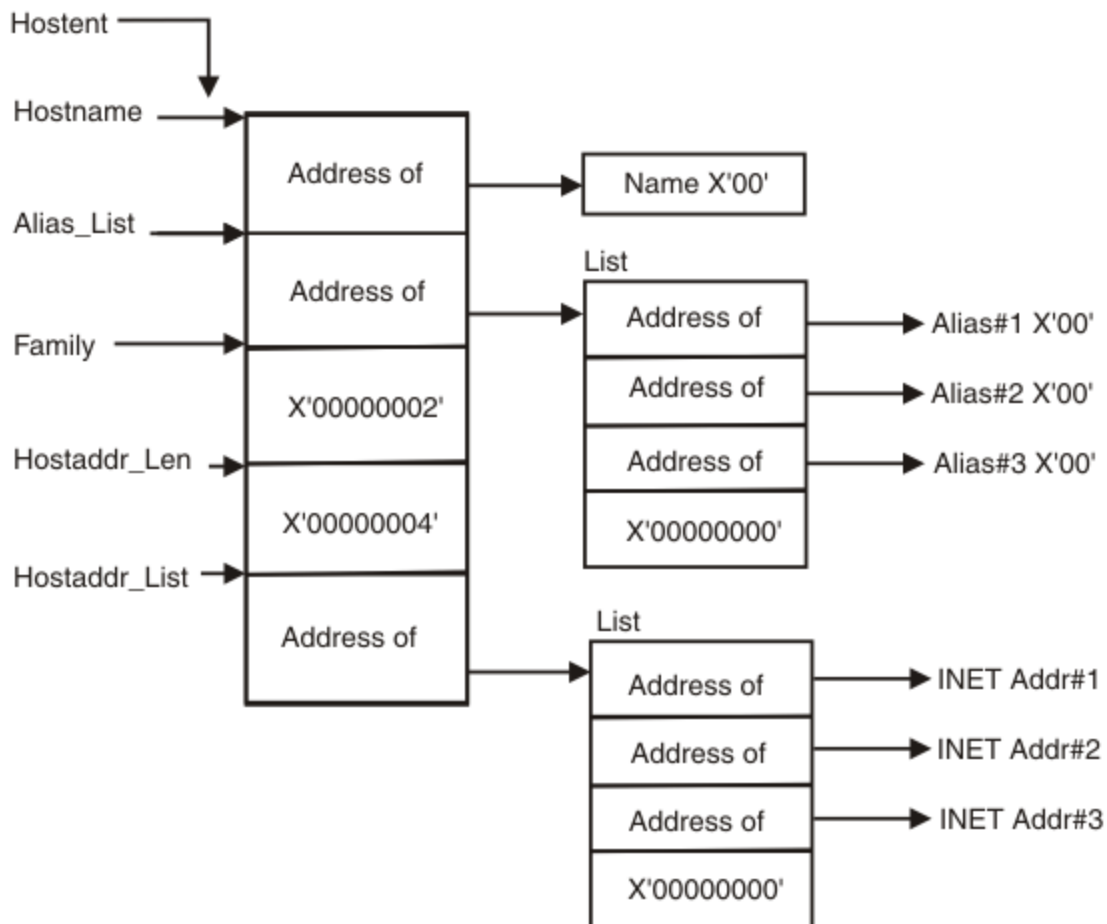


Figure 64. HOSTENT structure returned by the GETHOSTBYADDR macro

GETHOSTBYADDR returns the HOSTENT structure shown in Figure 64 on page 231. The HOSTENT structure is a task's serially reusable storage area. It should not be used or referenced between MVS tasks. The storage is freed when the task terminates. The assembler mapping of the structure is defined in macro EZBREHST, which is installed in the data set specified on your SMP/E DDDEF for MACLIB. This structure contains:

- The address of the host name returned by the macro. The name length is variable and is ended by X'00'.
- The address of a list of addresses that point to the alias names returned by the GETHOSTBYADDR. This list is ended by the pointer X'00000000'. Each alias name is a variable length field ended by X'00'.
- The value returned in the FAMILY field is always 2 to signify AF_INET.

- The length of the host Internet address returned in the HOSTADDR_LEN field is always 4 to signify AF_INET.
- The address of a list of addresses that point to the host Internet addresses returned by the macro. The list is ended by the pointer X'00000000'.

The HOSTENT structure uses indirect addressing to return a variable number of alias names and Internet addresses.

GETHOSTBYNAME

The GETHOSTBYNAME macro returns the alias names and the IPv4 Internet addresses of a host whose domain name is specified in the macro.

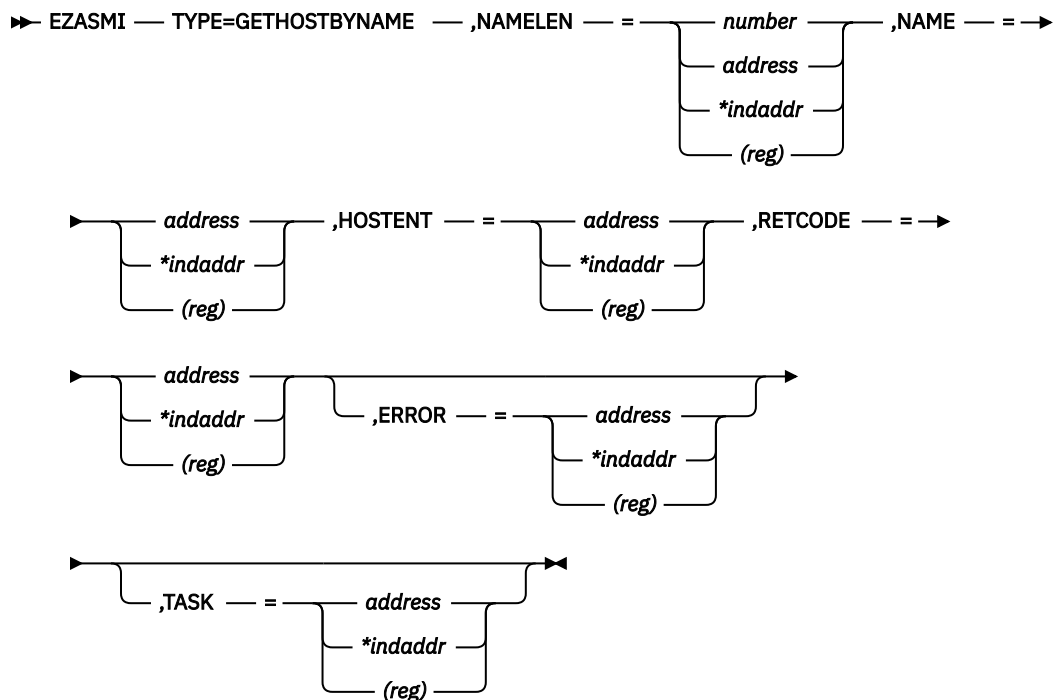
The name resolution attempted depends on how the resolver is configured and if any local host tables exist. See [z/OS Communications Server: IP Configuration Guide](#) for information about configuring the resolver and using local host tables.

If the host name is not found, the return code is -1.

Important: Repeated use of GETHOSTBYNAME before calls which implicitly or explicitly invoke INITAPI can result in the allocation of unreleased storage.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state. The PSW key must match the key in which the MVS application task was attached.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Note: The storage for the HOSTENT structure returned by this call is released during TERMAPI processing; therefore, the application program must not use the HOSTENT storage after the TERMAPI.

Keyword

Description

NAMELEN

Input parameter. A value or the address of a fullword binary field specifying the length of the name and alias fields. This length has a maximum value of 255 bytes.

NAME

A character string, up to 255 characters, set to a host name. This call returns the address of HOSTENT for this name. Any trailing blanks are removed from the specified name prior to trying to resolve it to an IP address.

HOSTENT

Output parameter. A fullword word containing the address of HOSTENT returned by the macro. For information about the HOSTENT structure, see [Figure 65 on page 234](#).

RETCODE

A fullword binary field that returns one of the following:

Value

Description

0

Successful call.

-1

An error occurred.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

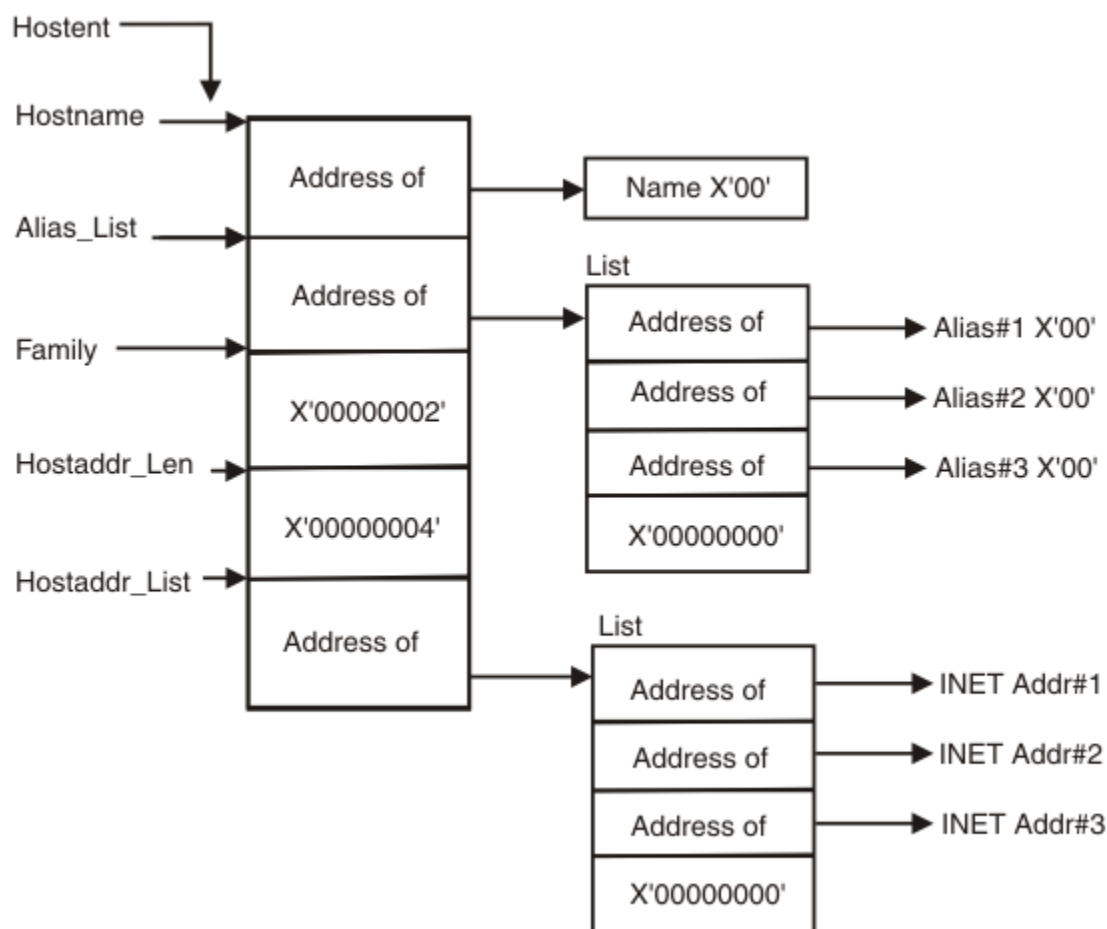


Figure 65. HOSTENT structure returned by the GETHOSTBYNAME macro

GETHOSTBYNAME returns the HOSTENT structure shown in Figure 65 on page 234. The HOSTENT structure is a task's serially reusable storage area. It should not be used or referenced between MVS tasks. The storage is freed when the task terminates. The assembler mapping of the structure is defined in macro EZBREHST, which is installed in the data set specified on your SMP/E DDDEF for MACLIB. This structure contains:

- The address of the host name returned by the macro. The name length is variable and is ended by X'00'.
- The address of a list of addresses that point to the alias names returned by GETHOSTBYNAME. This list is ended by the pointer X'00000000'. Each alias name is a variable length field ended by X'00'.
- The value returned in the FAMILY field is always 2 to signify AF_INET.
- The length of the host Internet address returned in the HOSTADDR_LEN field is always 4 to signify AF_INET.
- The address of a list of addresses that point to the host Internet addresses returned by the macro. The list is ended by the pointer X'00000000'.

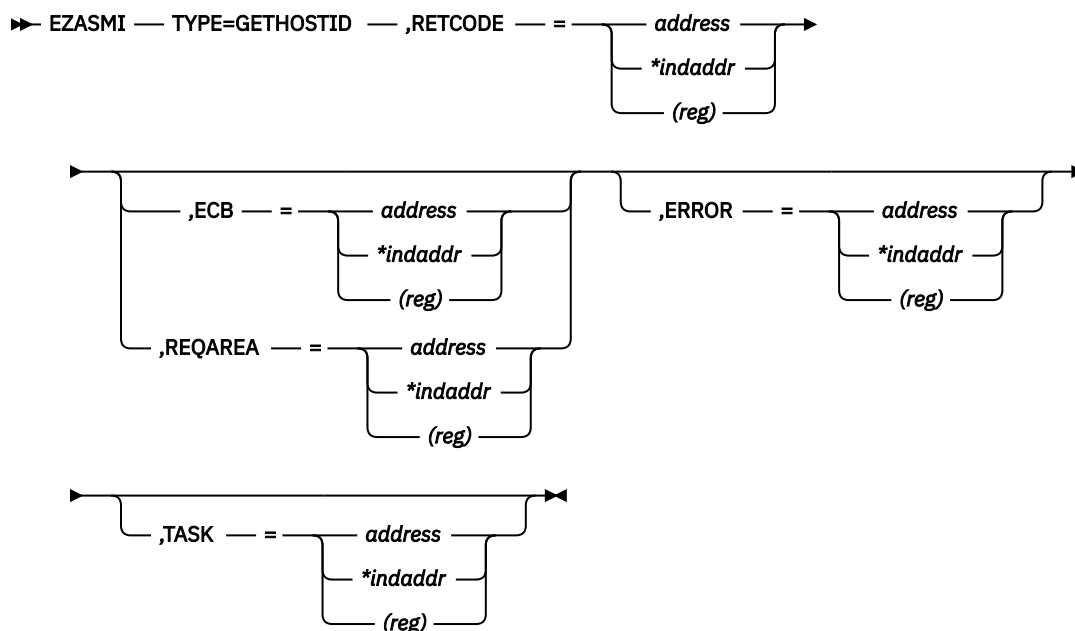
The HOSTENT structure uses indirect addressing to return a variable number of alias names and Internet addresses.

GETHOSTID

The GETHOSTID macro returns the 32-bit identifier for the current host. This value is the default home Internet address.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

RETCODE

Output parameter. Returns a fullword binary field containing the 32-bit Internet address of the host. A -1 in RETCODE indicates an error. There is no ERRNO parameter for this macro.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte ECB posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the ECB has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

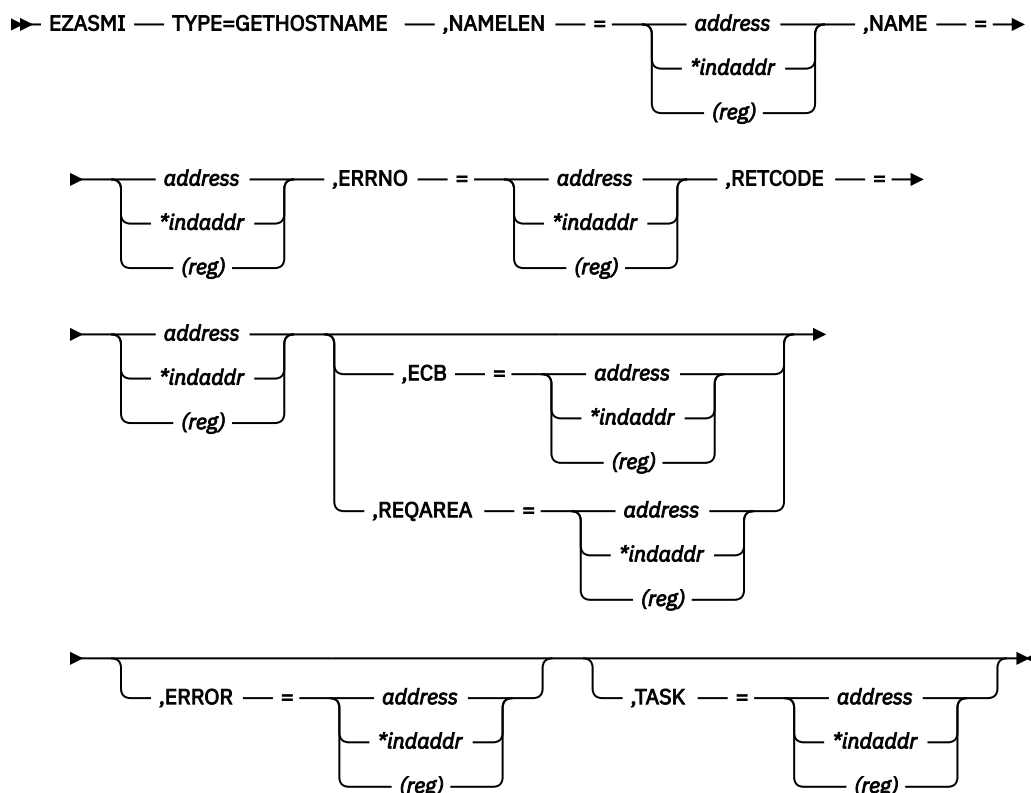
GETHOSTNAME

The GETHOSTNAME macro returns the name of the host processor on which the program is running. As many as NAMELEN characters are copied into the NAME field.

Note: The host name that is returned is the host name the TCPIP stack learned at startup from the TCPIP.DATA file that was found. For more information about host name, see [HOSTNAME statement in z/OS Communications Server: IP Configuration Reference](#).

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

NAMELEN

Input parameter. A fullword set to a value or the address of a fullword binary field set to the length of the name field. The minimum length of the NAME field is 1 character. The maximum length of the NAME field is 255 characters.

NAME

The application provides a pointer to a receiving field for the host name. If the host name is shorter than the NAMELEN value, the NAME field is filled with binary zeros after the host name. If the host name is longer than the NAMELEN value, the name is truncated.

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, ERRNO contains a valid error number. Otherwise, ignore ERRNO.

See [Appendix B, "Socket call error return codes," on page 743](#) for information about ERRNO return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

An error occurred.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte ECB posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the ECB has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

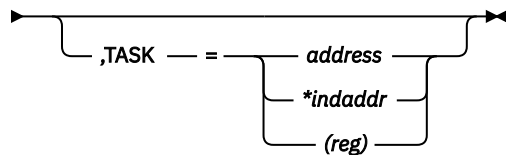
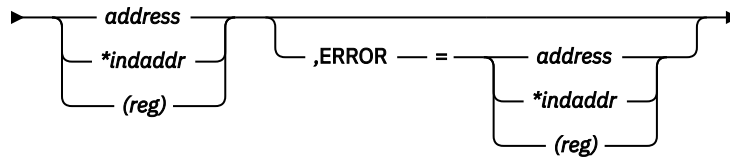
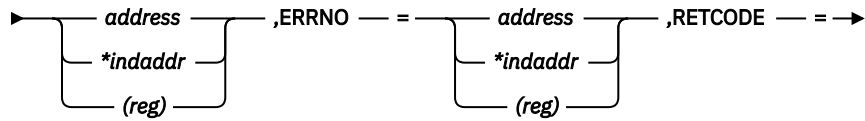
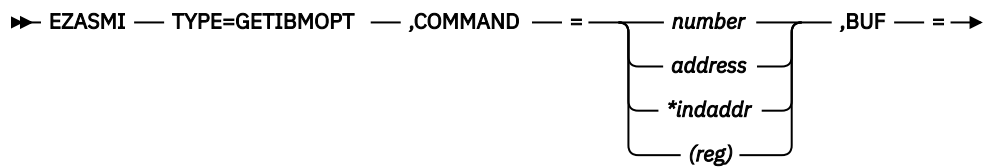
GETIBMOPT

The GETIBMOPT macro returns the number of TCP/IP images installed on a given MVS system and the status, version, and name of each image. With this information, the caller can dynamically choose the TCP/IP image with which to connect, using the INITAPI macro. The GETIBMOPT macro is optional. If you do not use this macro, follow the standard method to determine the connecting TCP/IP image:

1. Connect to the TCP/IP specified by TCPIPJOBNAME in the active TCPIP.DATA file.
2. Locate TCPIP.DATA using the search order described in the [z/OS Communications Server: IP Configuration Reference](#).

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

COMMAND

Input parameter. A value or the address of a fullword binary number specifying the command to be processed. The only valid value is 1.

BUF

Output parameter. A 100-byte buffer into which each active TCP/IP image status, version, and name are placed.

On successful return, these buffer entries contain the status, name and version of up to eight active TCP/IP images. The following layout shows BUF upon completion of the call.

NUM_IMAGES (4 bytes)		
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)

Figure 66. NUM_IMAGES field settings

The NUM_IMAGES field indicates how many entries of TCP_IMAGE are included in the total BUF field. If the NUM_IMAGES returned is 0, there are no TCP/IP images present.

The status field can combine the following information:

**Status Field
Meaning**

X'8xxx'

Active

X'4xxx'

Terminating

X'2xxx'

Down

X'1xxx'

Stopped or stopping

Note: In the above status fields, xxx is reserved for IBM use and can contain any value.

When the status field returns Down and Stopped, TCP/IP abended. Stopped, returned alone, indicates that TCP/IP was stopped. The following table shows the examples that are returned on version field:

Version	Field
TCP/IP z/OS Communications Server V1R13	X'061D'
TCP/IP z/OS Communications Server V2R1	X'0621'

The name field is the PROC name, left-aligned, and padded with blanks.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, **ERRNO** contains a valid error number. Otherwise, ignore **ERRNO**.

See [Appendix B, “Socket call error return codes,”](#) on page 743 for information about **ERRNO** return codes.

RETCODE

Output parameter. A fullword binary field with the following values:

Value

Description

-1

Call returned error. See **ERRNO**.

>=0

Successful call.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

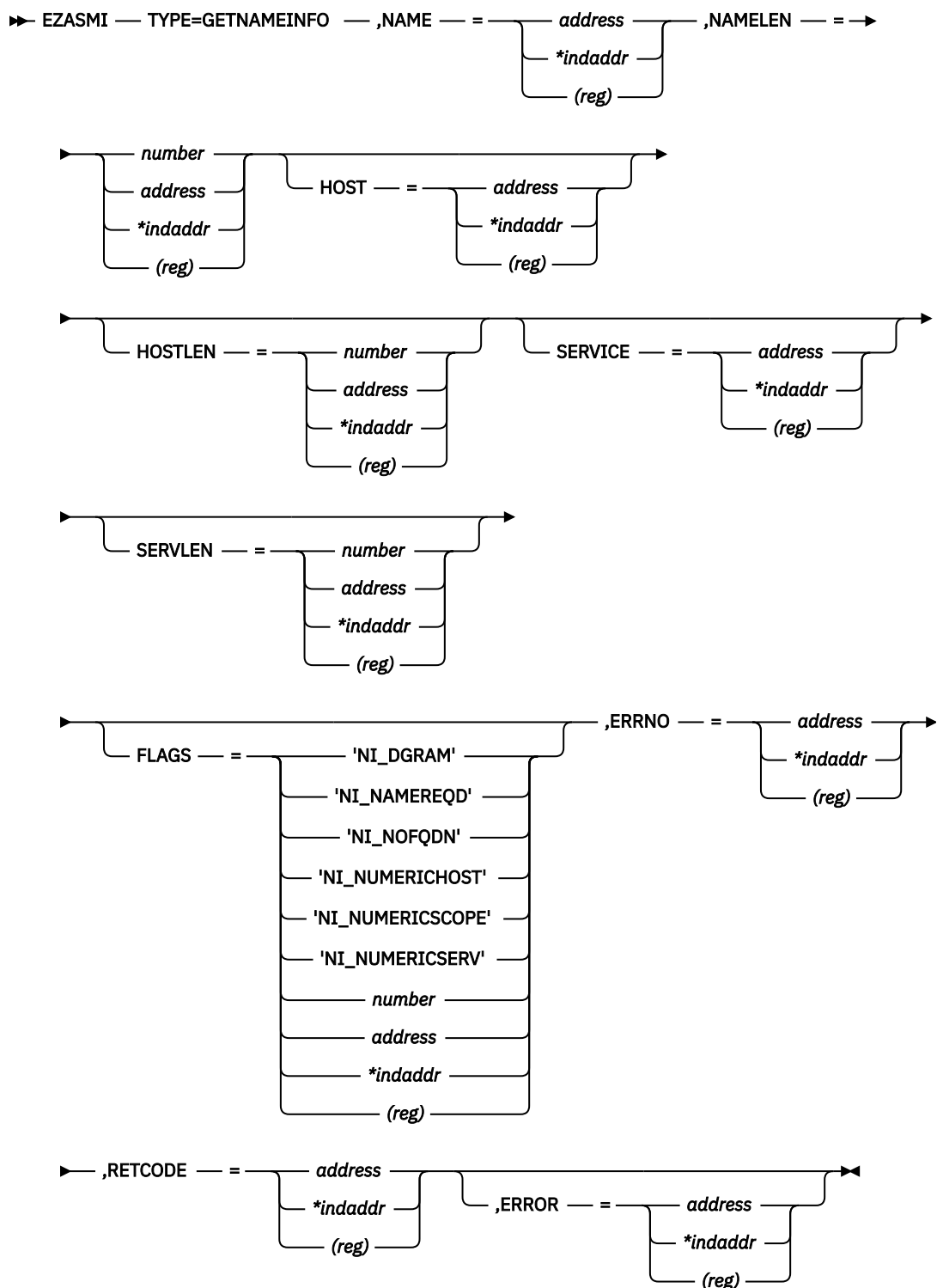
Input parameter. The location of the task storage area in your program.

GETNAMEINFO

The GETNAMEINFO macro returns the node name and service location of a socket address that is specified in the macro. On successful completion, GETNAMEINFO returns the node and service named, if requested, in the buffers provided.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword
Description
NAME

An input parameter. An IPv4 or IPv6 socket address structure to be translated. Include the SYS1.MACLIB(BPXYSOCK) macro to get the assembler mappings for the socket address structure. The socket address structure mappings begin at the SOCKADDR label. The AF_INET socket address

structure fields start at the SOCK_SIN label. The AF_INET6 socket address structure fields start at the SOCK_SIN6 label.

The IPv4 socket address structure must specify the following fields:

Field

Description

FAMILY

A halfword binary number specifying the IPv4 addressing family. For TCP/IP the value is a decimal 2, indicating AF_INET.

PORT

A halfword binary number specifying the port number.

IPv4-ADDRESS

A fullword binary number specifying the 32-bit IPv4 Internet address.

RESERVED

An 8-byte reserved field. This field is required, but is not used.

The IPv6 socket address structure specifies the following fields:

Field

Description

NAMELEN

A 1-byte binary field that specifies the length of the IPv6 socket address structure. Any value specified by the use of this field is ignored when processed as input. The field is set to 0 when processed as output.

FAMILY

A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT

A halfword binary number that specifies the port number.

FLOW-INFO

This field is ignored by the TYPE=GETNAMEINFO macro.

IPv6-ADDRESS

A 16-byte binary field that specifies the 128-bit IPv6 Internet address, in network byte order.

SCOPE-ID

A fullword binary field that specifies the scope for an IPv6 address as an interface index. The resolver ignores the SCOPE_ID field, unless the address in IPv6-ADDRESS is a link-local address and the HOST parameter also is specified.

NAMELEN

An input parameter. A fullword binary field. The length of the socket address structure pointed to by the NAME argument.

HOST

On input, storage capable of holding the returned resolved host name, which can be up to 255 bytes long, for the input socket address. If inadequate storage is specified to contain the resolved host name, then the resolver returns the host name up to the storage specified and truncation might occur. If the host's name cannot be located, the numeric form of the host's address is returned instead of its name. However, if the NI_NAMEREQD option is specified and no host name is located, an error is returned. One or both of the following groups of parameters are required:

- The HOST and HOSTLEN parameters
- The SERVICE and SERVLLEN parameters

Otherwise, an error occurs. The HOST name being queried consists of up to HOSTLEN or up to the first binary 0.

If the IPv6-ADDRESS value is a link-local address, and the SCOPE_ID interface index is nonzero, scope information is appended to the resolved host name using the format *host%scope information*.

The scope information can be the numeric form of the SCOPE_ID interface index or the interface name that is associated with the SCOPE_ID interface index. Use the NI_NUMERICSCOPE option to select which form is returned. The combined host name and scope information is 255 bytes or less. For more information about scope information and TYPE=GETNAMEINFO processing, see [z/OS Communications Server: IPv6 Network and Appl Design Guide](#).

HOSTLEN

Initially an input parameter. A fullword binary field that contains the length of the host storage that is used to contain the returned resolved host name. If HOSTLEN is 0 on input, then the resolved host name is not returned. The HOSTLEN value must be equal to or greater than the length of the longest host name, or hostname and scope information combination, to be returned. The TYPE=GETNAMEINFO returns the host name, or host name and scope information combination, up to the length specified by the HOSTLEN value. On output, HOSTLEN contains the length of the returned resolved host name, or host name and scope information combination. This is an optional field, but if you specify this field, you also must code the HOST value. One or both of the following groups of parameters are required:

- The HOST and HOSTLEN parameters
- The SERVICE and SERVLLEN parameters

Otherwise, an error occurs.

SERVICE

On input, storage capable of holding the returned resolved service name, which can be up to 32 bytes long, for the input socket address. If inadequate storage is specified to contain the resolved service name, then the resolver returns the service name up to the storage specified and truncation might occur. If the service name cannot be located, or if NI_NUMERICSERV was specified in the FLAGS operand, then the presentation form of the service address is returned instead of its name. This is an optional field, but if you specify this field, you also must code the SERVLLEN parameter. The SERVICE name being queried consists of up to SERVLLEN or up to the first binary zero. One or both of the following groups of parameters are required:

- The HOST and HOSTLEN parameters
- The SERVICE and SERVLLEN parameters

Otherwise, an error occurs.

SERVLLEN

Initially an input parameter. A fullword binary field that contains the length of the SERVICE storage used to contain the returned resolved service name. If SERVLLEN is 0 on input, then the service name information is not returned. SERVLLEN must be equal to or greater than the length of the longest service name to be returned. The TYPE=GETNAMEINFO returns the service name up to the length specified by SERVLLEN. On output, SERVLLEN contains the length of the returned resolved service name. This is an optional field, but if you specify it, you also must code the SERVICE parameter. One or both of the following groups of parameters are required:

- The HOST and HOSTLEN parameters
- The SERVICE and SERVLLEN parameters

Otherwise, an error occurs.

FLAGS

An input parameter. A fullword binary field. This is an optional field. A resultant binary value can represent multiple flags. If this parameter is not specified, the fully qualified host name is returned. The FLAGS argument can be a literal value or a fullword binary field:

Literal Value	Binary Value	Decimal Value	Description
'NI_NOFQDN'	X'00000001'	1	Return the NAME portion of the fully qualified domain name.
'NI_NUMERICHOST'	X'00000002'	2	Only return the numeric form of host's address.

Literal Value	Binary Value	Decimal Value	Description
'NI_NAMEREQD'	X'00000004'	4	Return an error if the host's name cannot be located.
'NI_NUMERICSERV'	X'00000008'	8	Only return the numeric form of the service address.
'NI_DGRAM'	X'00000010'	16	Indicates that the service is a datagram service. The default behavior is to assume that the service is a stream service.
'NI_NUMERICSCOPE'	X'00000020'	32	Only return the numeric form of the SCOPE-ID interface index, if applicable.

Do not specify both the NI_NUMERICHOST and the NI_NAMEREQD flags; otherwise, you get the EAI_FAIL (3) error. See [Appendix F, “GETNAMEINFO flags and returned information examples,”](#) on page 777 for examples of returned information when various combinations of flags are used.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, **ERRNO** contains a valid error number. Otherwise, ignore the **ERRNO** field.

See [Appendix B, “Socket call error return codes,”](#) on page 743 for information about **ERRNO** return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check **ERRNO** for an error code.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

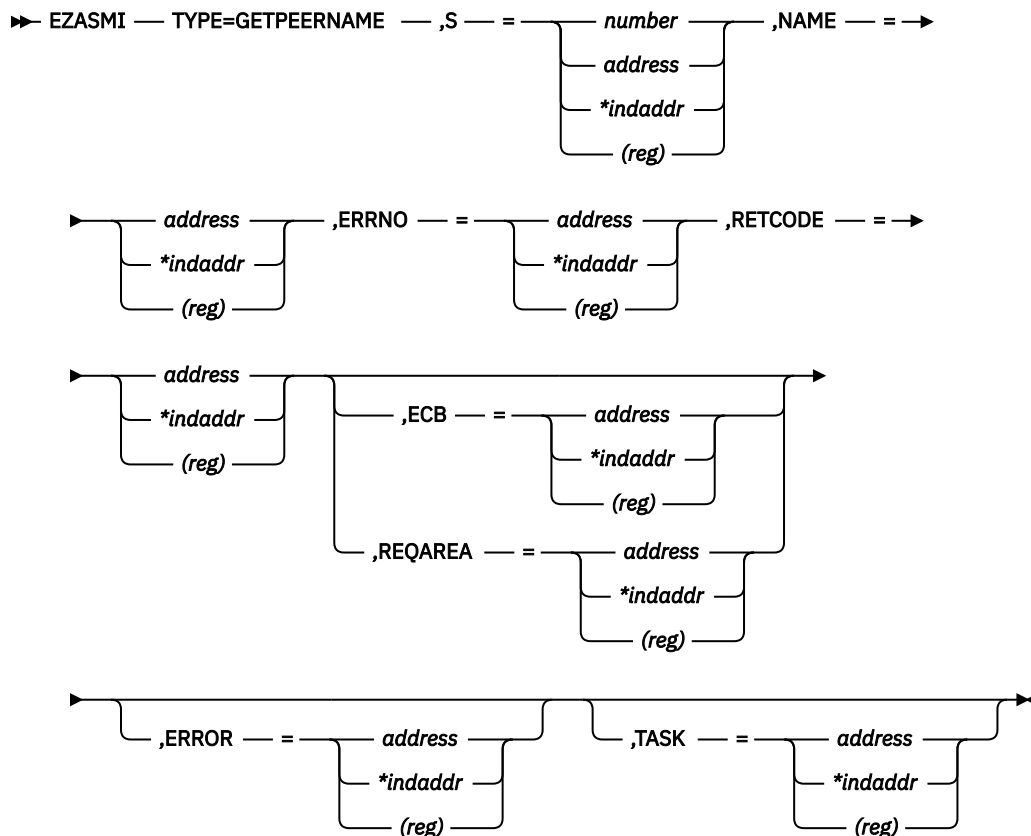
GETPEERNAME

The GETPEERNAME macro returns the name of the remote socket to which the local socket is connected.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.

Control parameters:	All parameters must be addressable by the caller and in the primary address space.
---------------------	--



Keyword Description

S

A value or the address of a halfword binary number specifying the local socket connected to the remote peer whose address is required.

NAME

Initially points to the peer name structure. It is filled when the call completes with the IPv4 or IPv6 address structure for the remote socket connected to the local socket, specified by S. Include the `SYS1.MACLIB(BPXYSOCK)` macro to get the assembler mappings for the socket address structure. The socket address structure mappings begin at the `SOCKADDR` label. The `AF_INET` socket address structure fields start at the `SOCK_SIN` label. The `AF_INET6` socket address structure fields start at the `SOCK_SIN6` label.

The IPv4 socket address structure must specify the following fields:

Field Description

FAMILY

A halfword binary field set to the connection peer IPv4 addressing family. The IPv4 value is always a decimal 2, indicating `AF_INET`.

PORT

A halfword binary field set to the connection peer port number.

IPv4-ADDRESS

A fullword binary field set to the 32-bit IPv4 Internet address of the connection peer host machine.

RESERVED

Input parameter. Specifies an 8-byte reserved field. This field is required, but not used.

The IPv6 socket structure must specify the following fields:

Field**Description****NAMELEN**

A 1-byte binary field specifying the length of this IPv6 socket address structure. Any value specified by the use of this field is ignored when processed as input and the field is set to 0 when processed as output.

FAMILY

A 1-byte binary field specifying the IPv6 addressing family. For IPv6 the value is a decimal 19, indicating AF_INET6.

PORT

A halfword binary field set to the connection peer port number.

FLOW-INFO

A fullword binary field specifying the traffic class and flow label. This value of this field is undefined.

IPv6-ADDRESS

A 16-byte binary field that is set to the 128-bit IPv6 Internet address, in network byte order, of the connection peer host machine.

SCOPE-ID

A fullword binary field which identifies a set of interfaces as appropriate for the scope of the address carried in the IPv6-ADDRESS field. For a link scope IPv6-ADDRESS, SCOPE-ID contains the link index for the IPv6-ADDRESS. For all other address scopes, SCOPE-ID is undefined.

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, this field contains an error number. See [Appendix B, "Socket call error return codes," on page 743](#) for information about ERRNO return codes.

RETCODE

Output parameter. A fullword binary field.

Value**Description**

0

Successful call.

-1

An error occurred.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte ECB posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the ECB has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

GETSOCKNAME

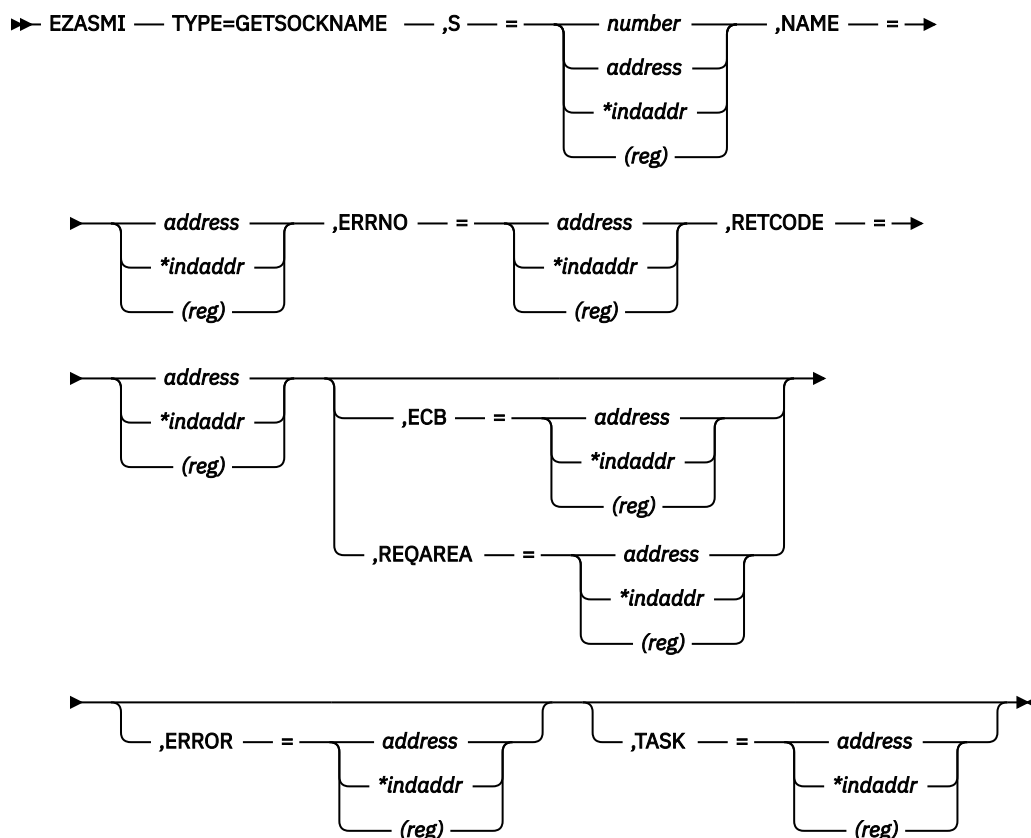
The GETSOCKNAME macro stores the name of the socket into the structure pointed to by NAME and returns the address to the socket that has been bound. If the socket is not bound to an address, the macro returns with the FAMILY field completed and the rest of the structure set to zeros.

Stream sockets are not assigned a name until after a successful call to BIND, CONNECT, or ACCEPT.

Use the GETSOCKNAME macro to determine the port assigned to a socket after that socket has been implicitly bound to a port. If an application calls CONNECT without previously calling BIND, the CONNECT macro completes the binding necessary by assigning a port to the socket. You can determine the port assigned to the socket by issuing GETSOCKNAME.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

S

Input parameter. A value or the address of a halfword binary number specifying the socket descriptor.

NAME

Initially, the application provides a pointer to the IPv4 or IPv6 socket address structure, which is specified on completion of the call with the socket name. Include the SYS1.MACLIB(BPXYSOCK) macro to get the assembler mappings for the socket address structure. The socket address structure mappings begin at the SOCKADDR label. The AF_INET socket address structure fields start at the SOCK_SIN label. The AF_INET6 socket address structure fields start at the SOCK_SIN6 label.

The IPv4 socket address structure must specify the following fields:

Field Description

FAMILY

Output parameter. A halfword binary field containing the IPv4 addressing family. The value for IPv4 socket descriptor (S parameter) is a decimal 2, indicating AF_INET.

PORT

Output parameter. A halfword binary field set to the port number bound to this socket. If the socket is not bound, a 0 is returned.

IPv4-ADDRESS

Output parameter. A fullword binary field set to the 32-bit IPv4 Internet address of the local host machine.

RESERVED

Output parameter. Specifies 8 bytes of binary zeros. This field is required, but not used.

The IPv6 socket structure must specify the following fields:

Field	
Description	

NAMELEN

Output parameter. A 1-byte binary field specifying the length of this IPv6 socket address structure. Any value specified by the use of this field is ignored when processed as input and the field is set to 0 when processed as output.

FAMILY

Output parameter. A 1-byte binary field specifying the IPv6 addressing family. The value for IPv6 socket descriptor (S parameter) is a decimal 19, indicating AF_INET6.

PORT

Output parameter. A halfword binary field set to the port number bound to this socket. If the socket is not bound, a 0 is returned.

FLOW-INFO

Output parameter. A fullword binary field specifying the traffic class and flow label. This value of this field is undefined.

IPv6-ADDRESS

Output parameter. A 16-byte binary field that is set to the 128-bit IPv6 Internet address, in network byte order, of the local host machine.

SCOPE-ID

A fullword binary field which identifies a set of interfaces as appropriate for the scope of the address carried in the IPv6-ADDRESS field. For a link scope IPv6-ADDRESS, SCOPE-ID contains the link index for the IPv6-ADDRESS. For all other address scopes, SCOPE-ID is undefined.

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, this field contains an error number. See Appendix B, “Socket call error return codes,” on page 743 for information about ERRNO return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value	
Description	

0	Successful call.
----------	------------------

-1	An error occurred.
-----------	--------------------

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte ECB posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the ECB has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

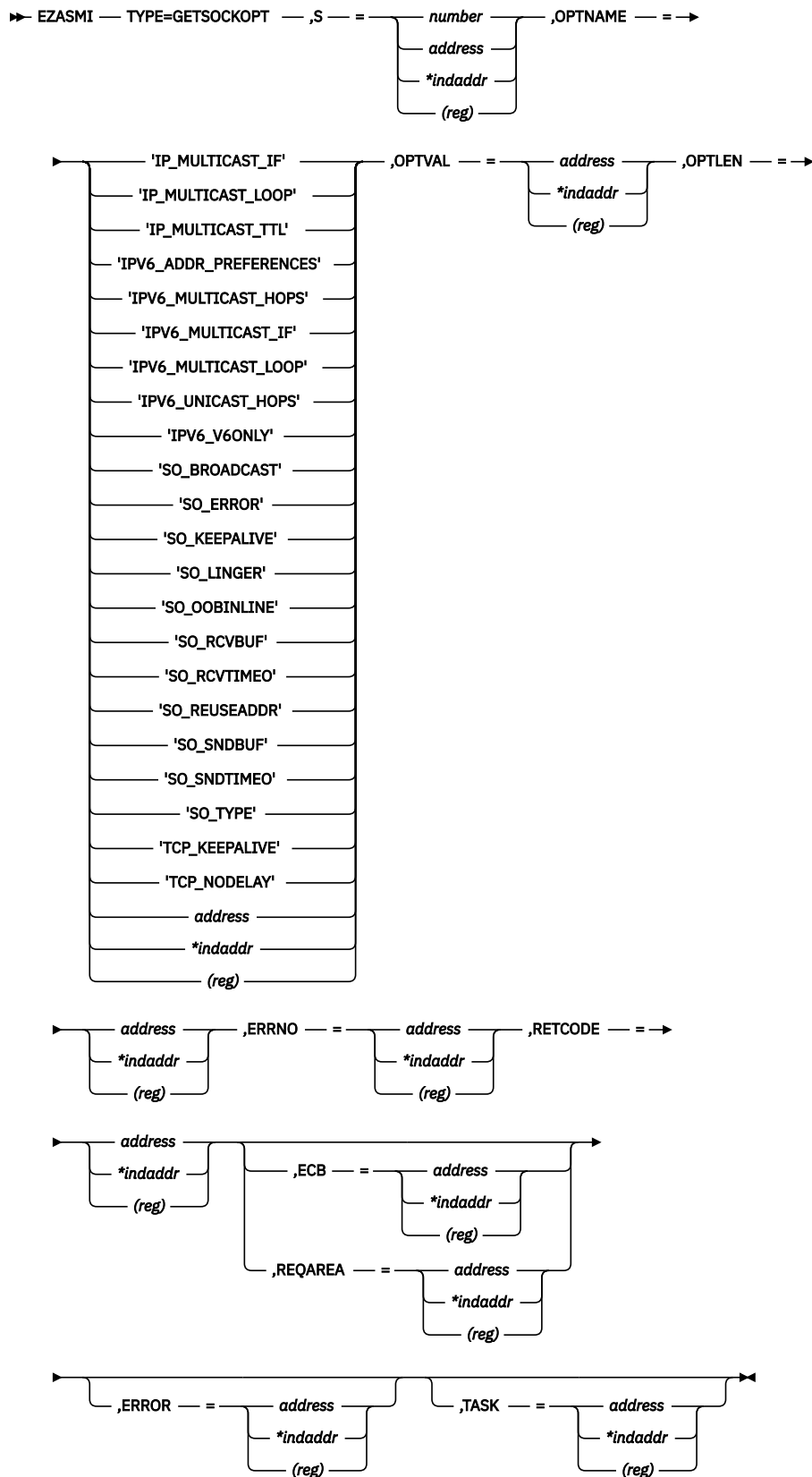
GETSOCKOPT

The GETSOCKOPT macro gets the options associated with a socket that were set using the SETSOCKOPT macro.

The options for each socket are described by the following parameters. You must specify the option that you want when you issue the GETSOCKOPT macro.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

S

Input parameter. A value or the address of a halfword binary number specifying the socket descriptor of the socket requiring options.

OPTNAME

Input parameter. See the table below for a list of the options and their unique requirements. See [Appendix D, “GETSOCKOPT/SETSOCKOPT command values,” on page 767](#) for the numeric values of OPTNAME.

OPTLEN

Input parameter. A fullword binary field containing the length of the data returned in OPTVAL. See the table below for determining on what to base the value of OPTLEN.

OPTVAL

Output parameter. See the table below for a list of the options and their unique requirements.

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, this field contains an error number. See [Appendix B, “Socket call error return codes,” on page 743](#) for information about ERRNO return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check ERRNO for an error code.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte ECB posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the ECB has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

Table 15. OPTNAME options for GETSOCKOPT and SETSOCKOPT

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IP_ADD_MEMBERSHIP</p> <p>Use this option to enable an application to join a multicast group on a specific interface. An interface has to be specified with this option. Only applications that want to receive multicast datagrams need to join multicast groups.</p> <p>This is an IPv4-only socket option.</p>	<p>Contains the IP_MREQ structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 interface address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IP-MREQ.</p>	<p>N/A</p>
<p>IP_ADD_SOURCE_MEMBERSHIP</p> <p>Use this option to enable an application to join a source multicast group on a specific interface and a specific source address. You must specify an interface and a source address with this option. Applications that want to receive multicast datagrams need to join source multicast groups.</p> <p>This is an IPv4-only socket option.</p>	<p>Contains the IP_MREQ_SOURCE structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ_SOURCE structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 source address and a 4-byte IPv4 interface address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ_SOURCE.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IP-MREQ-SOURCE.</p>	<p>N/A</p>
<p>IP_BLOCK_SOURCE</p> <p>Use this option to enable an application to block multicast packets that have a source address that matches the given IPv4 source address. You must specify an interface and a source address with this option. The specified multicast group must have been joined previously.</p> <p>This is an IPv4-only socket option.</p>	<p>Contains the IP_MREQ_SOURCE structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ_SOURCE structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 source address and a 4-byte IPv4 interface address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ_SOURCE.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IP-MREQ-SOURCE.</p>	<p>N/A</p>

Table 15. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IP_DROP_MEMBERSHIP</p> <p>Use this option to enable an application to exit a multicast group or to exit all sources for a multicast group.</p> <p>This is an IPv4-only socket option.</p>	<p>Contains the IP_MREQ structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 interface address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IP_MREQ.</p>	<p>N/A</p>
<p>IP_DROP_SOURCE_MEMBERSHIP</p> <p>Use this option to enable an application to exit a source multicast group.</p> <p>This is an IPv4-only socket option.</p>	<p>Contains the IP_MREQ_SOURCE structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ_SOURCE structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 source address and a 4-byte IPv4 interface address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ_SOURCE.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IP_MREQ-SOURCE.</p>	<p>N/A</p>
<p>IP_MULTICAST_IF</p> <p>Use this option to set or obtain the IPv4 interface address used for sending outbound multicast datagrams from the socket application.</p> <p>This is an IPv4-only socket option.</p> <p>Note: Multicast datagrams can be transmitted only on one interface at a time.</p>	<p>A 4-byte binary field containing an IPv4 interface address.</p>	<p>A 4-byte binary field containing an IPv4 interface address.</p>
<p>IP_MULTICAST_LOOP</p> <p>Use this option to control or determine whether a copy of multicast datagrams is looped back for multicast datagrams sent to a group to which the sending host itself belongs. The default is to loop the datagrams back.</p> <p>This is an IPv4-only socket option.</p>	<p>A 1-byte binary field.</p> <p>To enable, set to 1.</p> <p>To disable, set to 0.</p>	<p>A 1-byte binary field.</p> <p>If enabled, will contain a 1.</p> <p>If disabled, will contain a 0.</p>

Table 15. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IP_MULTICAST_TTL</p> <p>Use this option to set or obtain the IP time-to-live of outgoing multicast datagrams. The default value is '01'x meaning that multicast is available only to the local subnet.</p> <p>This is an IPv4-only socket option.</p>	<p>A 1-byte binary field containing the value of '00'x to 'FF'x.</p>	<p>A 1-byte binary field containing the value of '00'x to 'FF'x.</p>
<p>IP_UNBLOCK_SOURCE</p> <p>Use this option to enable an application to unblock a previously blocked source for a given IPv4 multicast group. You must specify an interface and a source address with this option.</p> <p>This is an IPv4-only socket option.</p>	<p>Contains the IP_MREQ_SOURCE structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ_SOURCE structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 source address and a 4-byte IPv4 interface address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ_SOURCE.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IP-MREQ-SOURCE.</p>	

Table 15. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IPV6_ADDR_PREFERENCES</p> <p>Use this option to query or set IPv6 address preferences of a socket. The default source address selection algorithm considers these preferences when it selects an IP address that is appropriate to communicate with a given destination address.</p> <p>This is an AF_INET6-only socket option.</p> <p>Result: These flags are only preferences. The stack could assign a source IP address that does not conform to the IPV6_ADDR_PREFERENCES flags that you specify.</p> <p>Guideline: Use the INET6_IS_SRCADDR function to test whether the source IP address matches one or more IPV6_ADDR_PREFERENCES flags.</p>	<p>Contains the 4-byte flags field IPV6_ADDR_PREFERENCES_FLAGS that is defined in SYS1.MACLIB(BPXYSOCK) with the following flags:</p> <p>IPV6_PREFER_SRC_HOME (X'00000001') Prefer home address</p> <p>IPV6_PREFER_SRC_COA (X'00000002') Prefer care-of address</p> <p>IPV6_PREFER_SRC_TMP (X'00000004') Prefer temporary address</p> <p>IPV6_PREFER_SRC_PUBLIC (X'00000008') Prefer public address</p> <p>IPV6_PREFER_SRC_CGA (X'00000010') Prefer cryptographically generated address</p> <p>IPV6_PREFER_SRC_NONCGA (X'00000020') Prefer non-cryptographically generated address</p> <p>Some of these flags are contradictory. Combining contradictory flags, such as IPV6_PREFER_SRC_CGA and IPV6_PREFER_SRC_NONCGA, results in error code EINVAL.</p> <p>See IPV6_ADDR_PREFERENCES and Mapping of GAI_HINTS/GAI_ADDRINFO EFLAGS in SEZAINST(CBLOCK) for the PL/I example of the OPTNAME and flag definitions.</p> <p>See IPV6_ADDR_PREFERENCES and AI_EFLAGS mappings in SEZAINST(EZACOBOL) for the COBOL example of the OPTNAME and flag definitions.</p>	<p>Contains the 4-byte flags field IPV6_ADDR_PREFERENCES_FLAGS that is defined in SYS1.MACLIB(BPXYSOCK) with the following flags:</p> <p>IPV6_PREFER_SRC_HOME (X'00000001') Prefer home address</p> <p>IPV6_PREFER_SRC_COA (X'00000002') Prefer care-of address</p> <p>IPV6_PREFER_SRC_TMP (X'00000004') Prefer temporary address</p> <p>IPV6_PREFER_SRC_PUBLIC (X'00000008') Prefer public address</p> <p>IPV6_PREFER_SRC_CGA (X'00000010') Prefer cryptographically generated address</p> <p>IPV6_PREFER_SRC_NONCGA (X'00000020') Prefer non-cryptographically generated address</p> <p>See IPV6_ADDR_PREFERENCES and Mapping of GAI_HINTS/GAI_ADDRINFO EFLAGS in SEZAINST(CBLOCK) for the PL/I example of the OPTNAME and flag definitions.</p> <p>See IPV6_ADDR_PREFERENCES and AI_EFLAGS mappings in SEZAINST(EZACOBOL) for the COBOL example of the OPTNAME and flag definitions.</p>

Table 15. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IPV6_JOIN_GROUP</p> <p>Use this option to control the reception of multicast packets and specify that the socket join a multicast group.</p> <p>This is an IPv6-only socket option.</p>	<p>Contains the IPV6_MREQ structure as defined in SYS1.MACLIB(BPXYSOCK). The IPV6_MREQ structure contains a 16-byte IPv6 multicast address followed by a 4-byte IPv6 interface index number.</p> <p>If the interface index number is 0, then the stack chooses the local interface.</p> <p>See the SEZAINST(CBLOCK) for the PL/I example of IPV6_MREQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IPV6-MREQ.</p>	<p>N/A</p>
<p>IPV6_LEAVE_GROUP</p> <p>Use this option to control the reception of multicast packets and specify that the socket leave a multicast group.</p> <p>This is an IPv6-only socket option.</p>	<p>Contains the IPV6_MREQ structure as defined in SYS1.MACLIB(BPXYSOCK). The IPV6_MREQ structure contains a 16-byte IPv6 multicast address followed by a 4-byte IPv6 interface index number.</p> <p>If the interface index number is 0, then the stack chooses the local interface.</p> <p>See the SEZAINST(CBLOCK) for the PL/I example of IPV6_MREQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IPV6-MREQ.</p>	<p>N/A</p>
<p>IPV6_MULTICAST_HOPS</p> <p>Use to set or obtain the hop limit used for outgoing multicast packets.</p> <p>This is an IPv6-only socket option.</p>	<p>Contains a 4-byte binary value specifying the multicast hops. If not specified, then the default is 1 hop.</p> <p>-1 indicates use stack default.</p> <p>0 – 255 is the valid hop limit range.</p> <p>Note: An application must be APF authorized to enable it to set the hop limit value above the system defined hop limit value. CICS applications cannot execute as APF authorized.</p>	<p>Contains a 4-byte binary value in the range 0 – 255 indicating the number of multicast hops.</p>

Table 15. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IPv6_MULTICAST_IF</p> <p>Use this option to set or obtain the index of the IPv6 interface used for sending outbound multicast datagrams from the socket application.</p> <p>This is an IPv6-only socket option.</p>	<p>Contains a 4-byte binary field containing an IPv6 interface index number.</p>	<p>Contains a 4-byte binary field containing an IPv6 interface index number.</p>
<p>IPv6_MULTICAST_LOOP</p> <p>Use this option to control or determine whether a multicast datagram is looped back on the outgoing interface by the IP layer for local delivery when datagrams are sent to a group to which the sending host itself belongs. The default is to loop multicast datagrams back.</p> <p>This is an IPv6-only socket option.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to 1.</p> <p>To disable, set to 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>
<p>IPv6_UNICAST_HOPS</p> <p>Use this option to set or obtain the hop limit used for outgoing unicast IPv6 packets.</p> <p>This is an IPv6-only socket option.</p>	<p>Contains a 4-byte binary value specifying the unicast hops. If not specified, then the default is 1 hop.</p> <p>-1 indicates use stack default.</p> <p>0 – 255 is the valid hop limit range.</p> <p>Note: APF authorized applications are permitted to set a hop limit that exceeds the system configured default. CICS applications cannot execute as APF authorized.</p>	<p>Contains a 4-byte binary value in the range 0 – 255 indicating the number of unicast hops.</p>
<p>IPv6_V6ONLY</p> <p>Use this option to set or determine whether the socket is restricted to send and receive only IPv6 packets. The default is to not restrict the sending and receiving of only IPv6 packets.</p> <p>This is an IPv6-only socket option.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to 1.</p> <p>To disable, set to 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>

Table 15. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>MCAST_BLOCK_SOURCE</p> <p>Use this option to enable an application to block multicast packets that have a source address that matches the given source address. You must specify an interface index and a source address with this option. The specified multicast group must have been joined previously.</p>	<p>Contains the GROUP_SOURCE_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_SOURCE_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address and a socket address structure of the source address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_SOURCE_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-SOURCE-REQ.</p>	N/A
<p>MCAST_JOIN_GROUP</p> <p>Use this option to enable an application to join a multicast group on a specific interface. You must specify an interface index. Applications that want to receive multicast datagrams must join multicast groups.</p>	<p>Contains the GROUP_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-REQ.</p>	N/A
<p>MCAST_JOIN_SOURCE_GROUP</p> <p>Use this option to enable an application to join a source multicast group on a specific interface and a source address. You must specify an interface index and the source address. Applications that want to receive multicast datagrams only from specific source addresses need to join source multicast groups.</p>	<p>Contains the GROUP_SOURCE_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_SOURCE_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address and a socket address structure of the source address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_SOURCE_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-SOURCE-REQ.</p>	N/A

Table 15. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>MCAST_LEAVE_GROUP</p> <p>Use this option to enable an application to exit a multicast group or exit all sources for a given multicast groups.</p>	<p>Contains the GROUP_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-REQ.</p>	<p>N/A</p>
<p>MCAST_LEAVE_SOURCE_GROUP</p> <p>Use this option to enable an application to exit a source multicast group.</p>	<p>Contains the GROUP_SOURCE_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_SOURCE_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address and a socket address structure of the source address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_SOURCE_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-SOURCE-REQ.</p>	<p>N/A</p>
<p>MCAST_UNBLOCK_SOURCE</p> <p>Use this option to enable an application to unblock a previously blocked source for a given multicast group. You must specify an interface index and a source address with this option.</p>	<p>Contains the GROUP_SOURCE_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_SOURCE_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address and a socket address structure of the source address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_SOURCE_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-SOURCE-REQ.</p>	<p>N/A</p>

Table 15. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_ASCII</p> <p>Use this option to set or determine the translation to ASCII data option. When SO_ASCII is set, data is translated to ASCII. When SO_ASCII is not set, data is not translated to or from ASCII.</p> <p>Note: This is a REXX-only socket option.</p>	<p>To enable, set to ON.</p> <p>To disable, set to OFF.</p> <p>Note: The <i>optvalue</i> is returned and is optionally followed by the name of the translation table that is used if translation is applied to the data.</p>	<p>If enabled, contains ON.</p> <p>If disabled, contains OFF.</p> <p>Note: The <i>optvalue</i> is returned and is optionally followed by the name of the translation table that is used if translation is applied to the data.</p>
<p>SO_BROADCAST</p> <p>Use this option to set or determine whether a program can send broadcast messages over the socket to destinations that can receive datagram messages. The default is disabled.</p> <p>Note: This option has no meaning for stream sockets.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to 1 or a positive value.</p> <p>To disable, set to 0.</p>	<p>A 4-byte field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>
<p>SO_DEBUG</p> <p>Use SO_DEBUG to set or determine the status of the debug option. The default is <i>disabled</i>. The debug option controls the recording of debug information.</p> <p>Notes:</p> <ol style="list-style-type: none"> 1. This is a REXX-only socket option. 2. This option has meaning only for stream sockets. 	<p>To enable, set to ON.</p> <p>To disable, set to OFF.</p>	<p>If enabled, contains ON.</p> <p>If disabled, contains OFF.</p>
<p>SO_EBCDIC</p> <p>Use this option to set or determine the translation to EBCDIC data option. When SO_EBCDIC is set, data is translated to EBCDIC. When SO_EBCDIC is not set, data is not translated to or from EBCDIC. This option is ignored by EBCDIC hosts.</p> <p>Note: This is a REXX-only socket option.</p>	<p>To enable, set to ON.</p> <p>To disable, set to OFF.</p> <p>Note: The <i>optvalue</i> is returned and is optionally followed by the name of the translation table that is used if translation is applied to the data.</p>	<p>If enabled, contains ON.</p> <p>If disabled, contains OFF.</p> <p>Note: The <i>optvalue</i> is returned and is optionally followed by the name of the translation table that is used if translation is applied to the data.</p>
<p>SO_ERROR</p> <p>Use this option to request pending errors on the socket or to check for asynchronous errors on connected datagram sockets or for other errors that are not explicitly returned by one of the socket calls. The error status is clear afterwards.</p>	<p>N/A</p>	<p>A 4-byte binary field containing the most recent ERRNO for the socket.</p>

Table 15. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_KEEPAIVE</p> <p>Use this option to set or determine whether the keep alive mechanism periodically sends a packet on an otherwise idle connection for a stream socket.</p> <p>The default is disabled.</p> <p>When activated, the keep alive mechanism periodically sends a packet on an otherwise idle connection. If the remote TCP does not respond to the packet or to retransmissions of the packet, the connection is terminated with the error ETIMEDOUT.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to 1 or a positive value.</p> <p>To disable, set to 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>
<p>SO_LINGER</p> <p>Use this option to control or determine how TCP/IP processes data that has not been transmitted when a CLOSE is issued for the socket. The default is disabled.</p> <p>Notes:</p> <ol style="list-style-type: none"> 1. This option has meaning only for stream sockets. 2. If you set a zero linger time, the connection cannot close in an orderly manner, but stops, resulting in a RESET segment being sent to the connection partner. Also, if the aborting socket is in nonblocking mode, the close call is treated as though no linger option had been set. <p>When SO_LINGER is set and CLOSE is called, the calling program is blocked until the data is successfully transmitted or the connection has timed out.</p> <p>When SO_LINGER is not set, the CLOSE returns without blocking the caller, and TCP/IP continues to attempt to send data for a specified time. This usually allows sufficient time to complete the data transfer.</p> <p>Use of the SO_LINGER option does not guarantee successful completion because TCP/IP waits only the amount of time specified in OPTVAL for SO_LINGER.</p>	<p>Contains an 8-byte field containing two 4-byte binary fields.</p> <p>Assembler coding:</p> <pre>ONOFF DS F LINGER DS F</pre> <p>COBOL coding:</p> <pre>ONOFF PIC 9(8) BINARY. LINGER PIC 9(8) BINARY.</pre> <p>Set ONOFF to a nonzero value to enable and set to 0 to disable this option. Set LINGER to the number of seconds that TCP/IP lingers after the CLOSE is issued.</p>	<p>Contains an 8-byte field containing two 4-byte binary fields.</p> <p>Assembler coding:</p> <pre>ONOFF DS F LINGER DS F</pre> <p>COBOL coding:</p> <pre>ONOFF PIC 9(8) BINARY. LINGER PIC 9(8) BINARY.</pre> <p>A nonzero value returned in ONOFF indicates enabled, a 0 indicates disabled. LINGER indicates the number of seconds that TCP/IP will try to send data after the CLOSE is issued.</p>

Table 15. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_OOBLINE</p> <p>Use this option to control or determine whether out-of-band data is received.</p> <p>Note: This option has meaning only for stream sockets.</p> <p>When this option is set, out-of-band data is placed in the normal data input queue as it is received and is available to a RECV or a RECVFROM even if the OOB flag is not set in the RECV or the RECVFROM.</p> <p>When this option is disabled, out-of-band data is placed in the priority data input queue as it is received and is available to a RECV or a RECVFROM only when the OOB flag is set in the RECV or the RECVFROM.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to 1 or a positive value.</p> <p>To disable, set to 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>
<p>SO_RCVBUF</p> <p>Use this option to control or determine the size of the data portion of the TCP/IP receive buffer.</p> <p>The size of the data portion of the receive buffer is protocol-specific, based on the following values prior to any SETSOCKOPT call:</p> <ul style="list-style-type: none"> • TCPRCVBufsize keyword on the TCPCONFIG statement in the PROFILE.TCPIP data set for a TCP Socket • UDPRCVBufsize keyword on the UDPCONFIG statement in the PROFILE.TCPIP data set for a UDP Socket • The default of 65535 for a raw socket 	<p>A 4-byte binary field.</p> <p>To enable, set to a positive value specifying the size of the data portion of the TCP/IP receive buffer.</p> <p>To disable, set to a 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a positive value indicating the size of the data portion of the TCP/IP receive buffer.</p> <p>If disabled, contains a 0.</p>

Table 15. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_RCVTIMEO</p> <p>Use this option to control or determine the maximum length of time that a receive-type function can wait before it completes.</p> <p>If a receive-type function has blocked for the maximum length of time that was specified without receiving data, control is returned with an errno set to EWOULDBLOCK. The default value for this option is 0, which indicates that a receive-type function does not time out.</p> <p>When the MSG_WAITALL flag (stream sockets only) is specified, the timeout takes precedence. The receive-type function can return the partial count. See the explanation of that operation's MSG_WAITALL flag parameter.</p> <p>The following receive-type functions are supported:</p> <ul style="list-style-type: none"> • READ • READV • RECV • RECVFROM • RECVMMSG 	<p>This option requires a TIMEVAL structure, which is defined in SYS1.MACLIB(BPXYRLIM) macro. The TIMEVAL structure contains the number of seconds and microseconds specified as fullword binary numbers. The seconds can be a value in the range 0 - 2 678 400 (equal to 31 days), and the microseconds can be a value in the range 0 - 1 000 000 (equal to 1 second). Although TIMEVAL value can be specified using microsecond granularity, the internal TCP/IP timers that are used to implement this function have a granularity of approximately 100 milliseconds.</p>	<p>This option stores a TIMEVAL structure that is defined in the SYS1.MACLIB(BPXYRLIM) macro. The TIMEVAL structure contains the number of seconds and microseconds, which are specified as fullword binary numbers. The number of seconds value that is returned is in the range 0 - 2 678 400 (equal to 31 days). The number of microseconds value that is returned is in the range 0 - 1 000 000.</p>

Table 15. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_REUSEADDR</p> <p>Use this option to control or determine whether local addresses are reused. The default is disabled. This alters the normal algorithm used with BIND. The normal BIND algorithm allows each Internet address and port combination to be bound only once. If the address and port have been already bound, then a subsequent BIND will fail and result error will be EADDRINUSE.</p> <p>When this option is enabled, the following situations are supported:</p> <ul style="list-style-type: none"> • A server can BIND the same port multiple times as long as every invocation uses a different local IP address and the wildcard address INADDR_ANY is used only one time per port. • A server with active client connections can be restarted and can bind to its port without having to close all of the client connections. • For datagram sockets, multicasting is supported so multiple bind() calls can be made to the same class D address and port number. • If you require multiple servers to BIND to the same port and listen on INADDR_ANY, see the SHAREPORT option on the PORT statement in TCPIP.PROFILE. 	<p>A 4-byte binary field.</p> <p>To enable, set to 1 or a positive value.</p> <p>To disable, set to 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>
<p>SO_SNDBUF</p> <p>Use this option to control or determine the size of the data portion of the TCP/IP send buffer. The size of the TCP/IP send buffer is protocol specific and is based on the following conditions:</p> <ul style="list-style-type: none"> • The TCPSENDBufsize keyword on the TCPCONFIG statement in the PROFILE.TCPIP data set for a TCP socket • The UDPSENDBufsize keyword on the UDPCONFIG statement in the PROFILE.TCPIP data set for a UDP socket • The default of 65535 for a raw socket 	<p>A 4-byte binary field.</p> <p>To enable, set to a positive value specifying the size of the data portion of the TCP/IP send buffer.</p> <p>To disable, set to a 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a positive value indicating the size of the data portion of the TCP/IP send buffer.</p> <p>If disabled, contains a 0.</p>

Table 15. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_SNDTIMEO</p> <p>Use this option to control or determine the maximum length of time that a send-type function can remain blocked before it completes.</p> <p>If a send-type function has blocked for this length of time, it returns with a partial count or, if no data is sent, with an errno set to EWOULDBLOCK. The default value for this is 0, which indicates that a send-type function does not time out.</p> <p>For a SETSOCKOPT, the following send-type functions are supported:</p> <ul style="list-style-type: none"> • SEND • SENDMSG • SENDTO • WRITE • WRITEV 	<p>This option requires a TIMEVAL structure, which is defined in the SYS1.MACLIB(BPXYRLIM) macro. The TIMEVAL structure contains the number of seconds and microseconds specified as fullword binary numbers. The seconds value is in the range 0 - 2 678 400 (equal to 31 days), and the microseconds value is in the range 0 - 1 000 000 (equal to 1 second). Although the TIMEVAL value can be specified using microsecond granularity, the internal TCP/IP timers that are used to implement this function have a granularity of approximately 100 milliseconds.</p>	<p>This option stores a TIMEVAL structure that is defined in SYS1.MACLIB(BPXYRLIM). The TIMEVAL structure contains the number of seconds and microseconds, which are specified as fullword binary numbers. The number of seconds value that is returned is in the range 0 - 2 678 400 (equal to 31 days). The microseconds value that is returned is in the range 0 - 1 000 000.</p>
<p>SO_TYPE</p> <p>Use this option to return the socket type.</p>	<p>N/A</p>	<p>A 4-byte binary field indicating the socket type:</p> <p>X'1' indicates SOCK_STREAM.</p> <p>X'2' indicates SOCK_DGRAM.</p> <p>X'3' indicates SOCK_RAW.</p>
<p>TCP_KEEPAIVE</p> <p>Use this option to set or determine whether a socket-specific timeout value (in seconds) is to be used in place of a configuration-specific value whenever keep alive timing is active for that socket.</p> <p>When activated, the socket-specified timer value remains in effect until respecified by SETSOCKOPT or until the socket is closed. For more information about the socket option parameters, see TCP_KeepAlive socket option inz/OS Communications Server: IP Programmer's Guide and Reference.</p> <p>Tip: The site administrator can enable the global keep-alive mechanism by specifying the INTERVAL parameter on the TCPCONFIG statement in the TCP/IP stack profile data set, TCPIP.PROFILE.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to a value in the range of 1 – 2 147 460.</p> <p>To disable, set to a value of 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains the specific timer value (in seconds) that is in effect for the given socket.</p> <p>If disabled, contains a 0 indicating keep alive timing is not active.</p>

Table 15. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>TCP_NODELAY</p> <p>Use this option to set or determine whether data sent over the socket is subject to the Nagle algorithm (RFC 896).</p> <p>Under most circumstances, TCP sends data when it is presented. When this option is enabled, TCP will wait to send small amounts of data until the acknowledgment for the previous data sent is received. When this option is disabled, TCP will send small amounts of data even before the acknowledgment for the previous data sent is received.</p> <p>Note: Use the following format to set TCP_NODELAY OPTNAME value for COBOL programs:</p> <pre> 01 TCP-NODELAY-VAL PIC 9(10) COMP VALUE 2147483649. 01 TCP-NODELAY-REDEF REDEFINES TCP-NODELAY-VAL. 05 FILLER PIC 9(6) BINARY. 05 TCP-NODELAY PIC 9(8) BINARY. </pre>	<p>A 4-byte binary field.</p> <p>To enable, set to a 0.</p> <p>To disable, set to a 1 or nonzero.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 0.</p> <p>If disabled, contains a 1.</p>

GIVESOCKET

The GIVESOCKET macro makes the socket available for a TAKESOCKET macro issued by another program. The GIVESOCKET macro can specify any connected stream socket. Typically, the GIVESOCKET macro is issued by a concurrent server program that creates sockets to be passed to a subtask.

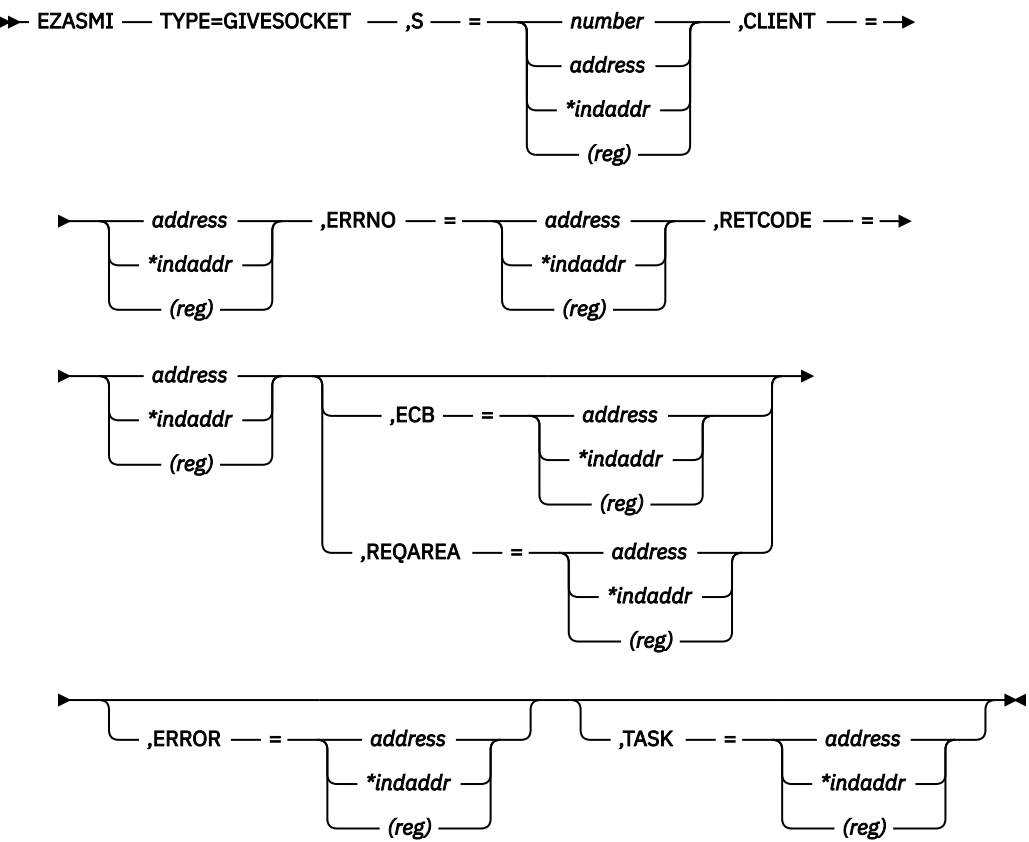
After a program has issued a GIVESOCKET macro for a socket, it can only issue a CLOSE macro for the same socket. Sockets which are given but not taken for a period of four days will be closed and will no longer be available for taking. If a select for the socket is outstanding, it is posted.

Note: Both concurrent servers and iterative servers use this interface. An iterative server handles one client at a time. A concurrent server receives connection requests from multiple clients and creates subtasks that process the client requests. When a subtask is created, the concurrent server gets a new socket, passes the new socket to the subtask, and dissociates itself from the connection. The TCP/IP Listener program is an example of a concurrent server.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit.
	<p>Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.</p>
ASC mode:	Primary address space control (ASC) mode.

Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

S

Input parameter. A value or the address of a halfword binary number specifying the descriptor of the socket to be given.

CLIENT

Input parameter. The client ID for this application.

Field Description

DOMAIN

Input parameter. A fullword binary number specifying the domain of the client. For TCP/IP the value is a decimal 2, indicating AF_INET, or a decimal 19, indicating AF_INET6.

Note: A socket given by GIVESOCKET can only be taken by a TAKESOCKET with the same DOMAIN, address family (AF_INET or AF_INET6).

NAME

An 8-character field, left-aligned, padded to the right with blanks. On completion of the call, this field contains the MVS address space name of the application that is going to take the socket. If the socket-taking application is in the same address space as the socket-giving application, NAME can be obtained using the GETCLIENTID call. If this field is set to blanks, any MVS address space requesting a socket can take this socket.

TASK

Specifies an 8-byte field that is set to the MVS subtask identifier of the socket-taking task (specified on the SUBTASK parameter on its INITAPI macro). If this field is set to blanks, any subtask in the address space specified in the NAME field can take the socket.

RESERVED

Input parameter. A 20-byte reserved field. This field is required, but not used.

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, ERRNO contains a valid error number. Otherwise, ignore ERRNO.

See [Appendix B, “Socket call error return codes,” on page 743](#) for information about ERRNO return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value**Description****0**

Successful call.

-1

Check ERRNO for an error code.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte ECB posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the Ecb has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

GLOBAL

The GLOBAL macro allocates a global storage area that is addressable by all socket users in an address space. If more than one module is using sockets, you must supply the address of the global storage area to each user. Each program using the sockets interface should define global storage using the instruction EZASMI TYPE=GLOBAL with STORAGE=DSECT.

If this macro is not named, the default name EZASMGWA is assumed.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.

Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

➤ EZASMI — TYPE=GLOBAL — ,STORAGE — = — DSECT —
CSECT —

Keyword Description

STORAGE

Input parameter. Defines one of the following storage definitions:

Keyword Description

DSECT

Generates a DSECT.

CSECT

Generates an inline storage definition that can be used within a CSECT or as a part of a larger DSECT.

INET6_IS_SRCADDR

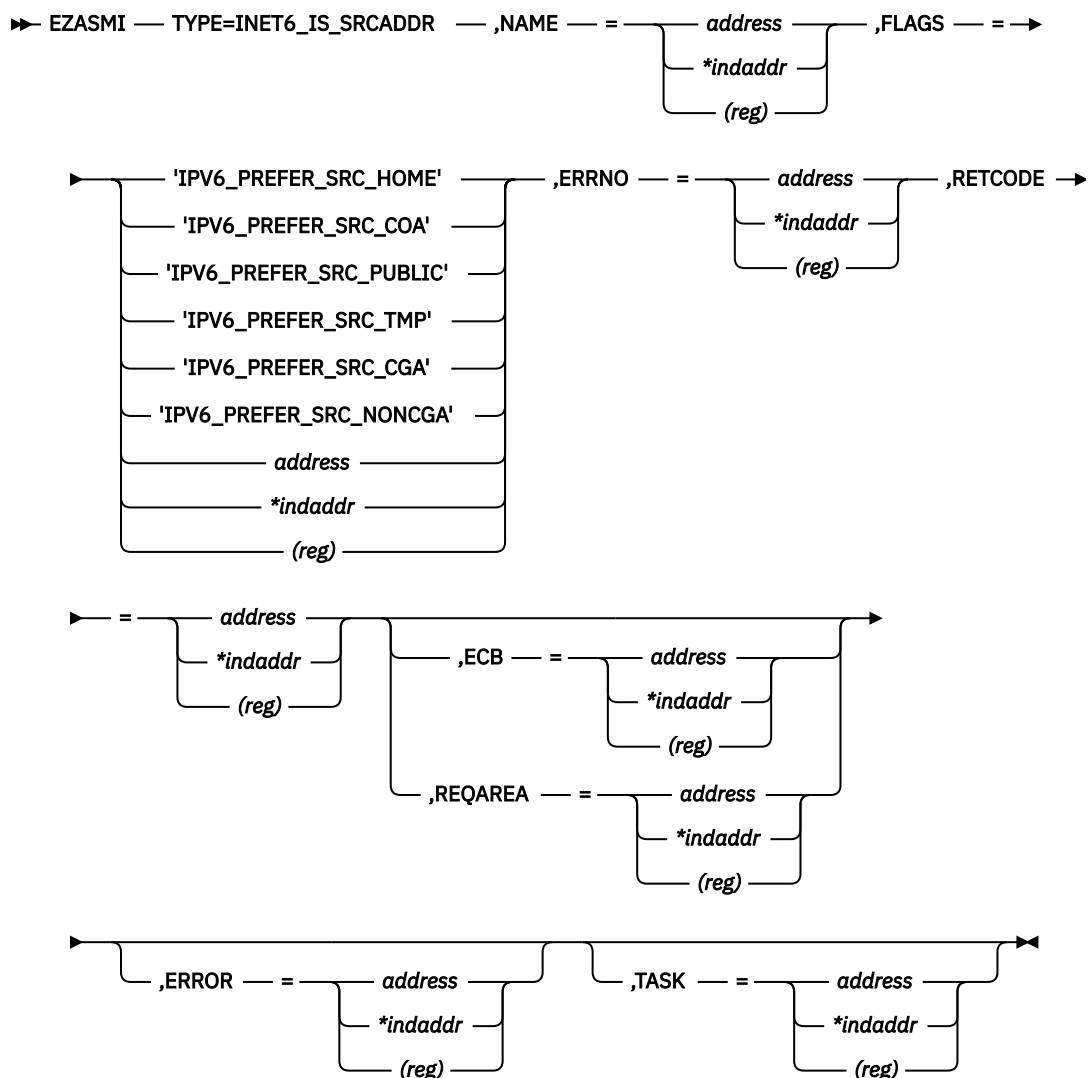
The INET6_IS_SRCADDR macro indicates whether an input IPV6 socket address matches an address that is defined to the stack, which conforms to one or more input IPV6_ADDR_PREFERENCES flags.

A client or server program can use the INET6_IS_SRCADDR macro when it strictly requires the IP addresses that it uses to have one or more specific attributes before allowing network activity. Such an application would perform the following tasks:

1. (Optional) Issue the SETSOCKOPT macro with the IPV6_ADDR_PREFERENCES option to set the selection preferences of the source IP address for the socket.
2. Issue the BIND2ADDRSEL macro to bind a socket to a local IP address. The BIND2ADDRSEL macro attempts to assign a local IP address that matches the IPV6_ADDR_PREFERENCES flags that are specified with the SETSOCKOPT macro, but does not guarantee that the local IP address that it selects satisfies all preferences.
3. Issue the GETSOCKNAME macro to obtain the local IP address bound to the socket.
4. Issue the INET6_IS_SRCADDR macro to verify whether the local IP address has the correct attributes.

The macro returns true (value 1) when the IPv6 address corresponds to a valid address in the node, and satisfies the given IPV6_ADDR_PREFERENCES flag or flags. If the IPv6 address input value does not correspond to any address in the node, or if the flags are not valid preference flags, the macro returns a failure (value -1). If the input address is a valid address on the node, but does not satisfy the preference flags indicated, the function returns false (value 0).

You can specify more than one IPV6_ADDR_PREFERENCES flag on this macro. All flags must be satisfied for the result to be true. Some IPV6_ADDR_PREFERENCES flags are contradictory. If you combine contradictory flags on a single macro invocation, the result will be false.



Keyword Descriptions

NAME

An input parameter. An IPv6 socket address structure describing the interface address to be tested. Include the `SYS1.MACLIB(BPXYSOCK)` macro in your program to get the assembler mappings for the socket address structure. The socket address structure mappings begin at the `SOCKADDR` label. The `AF_INET6` socket address structure fields start at the `SOCK_SIN6` label.

The IPv6 socket address structure specifies the following fields:

FAMILY

A halfword binary field that specifies the IPv6 addressing family. This value must be set to `AF_INET6`.

Tip: You can specify an IPv4-mapped IPv6 address.

PORT

This field is ignored by the `TYPE=INET6_IS_SRCADDR` macro.

FLOWINFO

This field is ignored by the `TYPE=INET6_IS_SRCADDR` macro.

IP-ADDRESS

A 16-byte binary field that specifies the 128-bit IPv6 Internet address, in network byte order.

SCOPE-ID

A fullword binary field that specifies the scope for an IPv6 address as an interface index. This field must be nonzero if the address specified in the IPv6-ADDRESS field is a link-local address and must be 0 for any other scope.

FLAGS

An input parameter. The FLAGS argument can be a literal value or a fullword binary field. The following flags can be specified individually or in combination.

Flag name	Binary value	Description
IPV6_PREFER_SRC_HOME	X'00000001'	Test whether the input IP address is a home address. ¹
IPV6_PREFER_SRC_COA	X'00000002'	Test whether the input IP address is a care-of address. ²
IPV6_PREFER_SRC_TMP	X'00000004'	Test whether the input IP address is a temporary address.
IPV6_PREFER_SRC_PUBLIC	X'00000008'	Test whether the input IP address is a public address.
IPV6_PREFER_SRC_CGA	X'00000010'	Test whether the input IP address is cryptographically generated. ²
IPV6_PREFER_SRC_NONCGA	X'00000020'	Test whether the input IP address is not cryptographically generated. ¹
Notes: 1. Any valid IP address that is known to the stack satisfies this flag. 2. z/OS Communications Server does not support this type of address. The macro always returns FALSE if this flag is specified with a valid IP address that is known to the stack.		

Requirement: You must pass a binary number to the macro to specify more than one preference flags. Include the BPXYSOCK macro in your program to get the binary values for the flags.

Tip: Some of these flags are contradictory, for example:

- The flag IPV6_PREFER_SRC_HOME contradicts the flag IPV6_PREFER_SRC_COA.
- The flag IPV6_PREFER_SRC_CGA contradicts the flag IPV6_PREFER_SRC_NONCGA.
- The flag IPV6_PREFER_SRC_TMP contradicts the flags IPV6_PREFER_SRC_PUBLIC.

Result: If you specify contradictory flags on the macro, the result is FALSE.

Example:

```
MYFLAGS DC A(IPV6_PREFER_SRC_HOME+IPV6_PREFER_SRC_NONCGA) one way to combine flags
EZASMI  TYPE=INET6_IS_SRCADDR,
        NAME=name,
        FLAGS=MYFLAGS
```

ERRNO

Output parameter. A fullword binary field. If the RETCODE value is negative, the ERRNO field contains a valid error number; otherwise, ignore ERRNO.

See Appendix B, “Socket call error return codes,” on page 743 for information about ERRNO return codes.

ECB or REQAREA

Input parameter. This parameter is required if you use APITYPE=3. This parameter points to a 104-byte field that contains one of the following values:

For ECB

A 4-byte ECB that is posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For both ECB and REQAREA

The last 100 bytes is a storage field that is used by the interface to save the state information.

Requirement: This storage must not be modified until the macro function has completed and the ECB has been posted, or the asynchronous exit has been driven.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value**Description****0**

False. The IPv6 address corresponds to a valid address in the node, but does not satisfy one or more of the input IPV6_ADDR_PREFERENCES flags.

1

True. The IPv6 address corresponds to a valid address in the node and satisfies the given IPV6_ADDR_PREFERENCES flag or flags.

Tip: Any valid IPv6 address on a z/OS V1R12 node satisfies the following preference flags:

- IPV6_PREFER_SRC_HOME
- IPV6_PREFER_SRC_NONCGA

-1

The input IPV6 address does not correspond to an address on the node or the input flags are not valid IPV6_PREFER_SRC_ADDR flags. Check the ERRNO value for an error code.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

INITAPI

The INITAPI macro connects an application to the TCP/IP interface. Almost all sockets programs that are written in COBOL, PL/I, or assembler language must issue the INITAPI macro before they issue other sockets macros.

Note: Because the default INITAPI still requires the TERMAPI to be issued, you always code the INITAPI command.

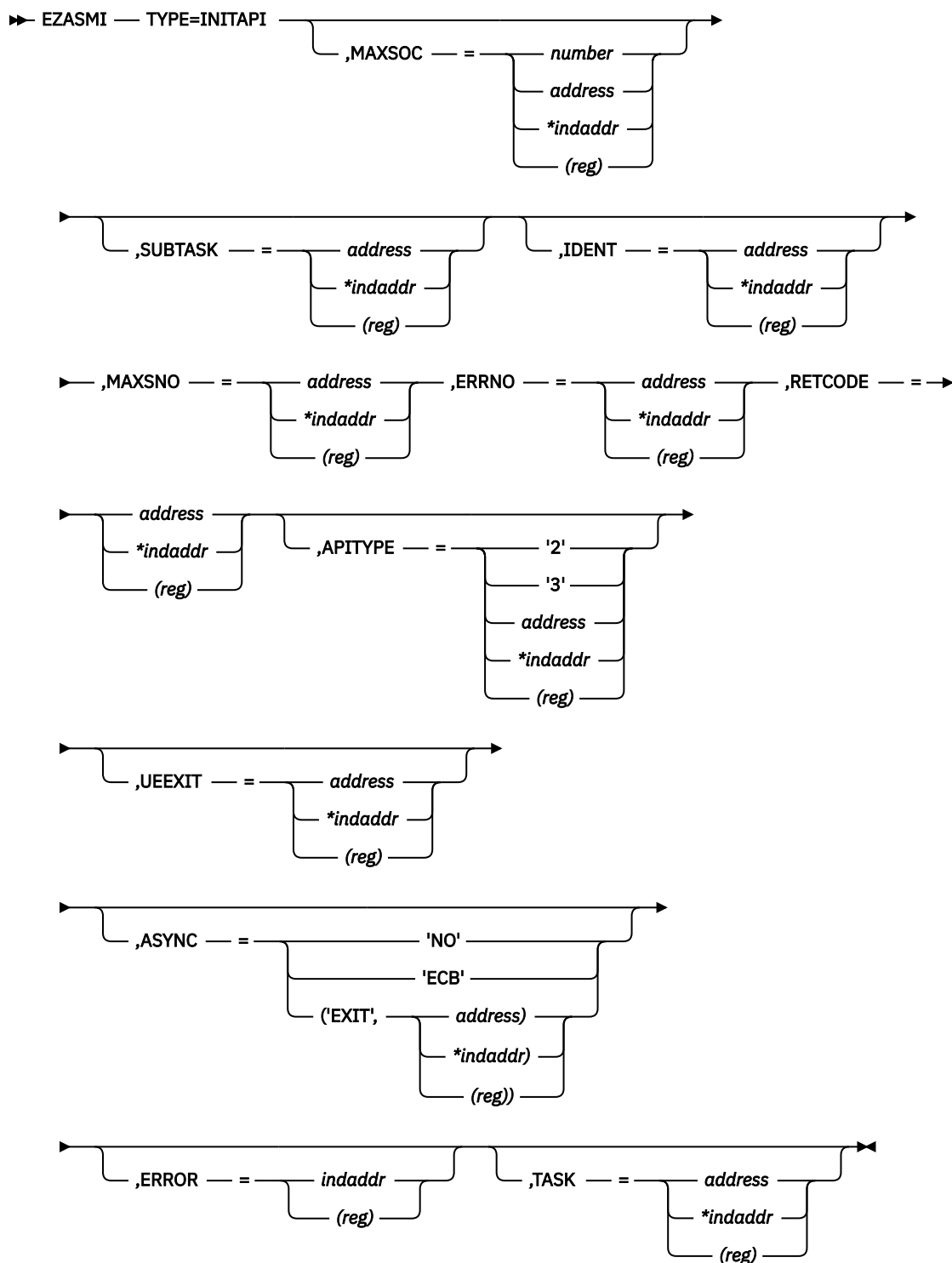
The exceptions to this rule are the following calls, which, when issued first, will generate a default INITAPI call:

- GETCLIENTID
- GETHOSTID
- GETHOSTNAME
- GETIBMOPT
- SELECT
- SELECTEX
- SOCKET
- TAKESOCKET

Note: Only the first INITAPI triggers a read of the TCPIP.DATA and all other INITAPIs under that address space will use the values read by the first INITAPI.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Descriptions

MAXSOC

Optional input parameter. A halfword binary field specifying the maximum number of sockets supported by this application. The maximum number is 65535 and the minimum number is 50. The default value for MAXSOC is 50. If less than 50 are requested, MAXSOC defaults to 50.

SUBTASK

Optional input parameter. An 8-byte field that is used to identify a subtask in an address space that can contain multiple subtasks. It is suggested that you use your own job name as part of your subtask

name. This will ensure that, if you issue more than one INITAPI command from the same address space, each SUBTASK parameter will be unique.

IDENT

Optional input parameter. A structure containing the identities of the TCP/IP address space and your address space. Specify IDENT on the INITAPI macro from an address space. The structure is as follows:

Field

Description

TCPNAME

Input parameter. An 8-byte character field set to the name of the TCP/IP address space that you want to connect to. If this is not specified, the system derives a value from the configuration file, as described in the [z/OS Communications Server: IP Configuration Reference](#).

ADSNAME

Input parameter. An 8-byte character field set to the name of the calling program's address space. If this is not specified, the system will derive a value from the MVS control block structure.

MAXSNO

Output parameter. A fullword binary field containing the greatest descriptor number assigned to this application. The lowest socket number is 0. If you have 50 sockets, they are numbered in the range 0 – 49. If MAXNO is not specified, the value for MAXNO is 49.

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, ERRNO field contains a valid error number. Otherwise, ignore ERRNO.

See [Appendix B, “Socket call error return codes,” on page 743](#) for information about ERRNO return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check ERRNO for an error code.

APITYPE

Optional input parameter. A halfword binary field specifying the APITYPE. For details on usage, see [“Task management and asynchronous function processing” on page 198](#).

Value

Meaning

2

APITYPE 2. This is the default.

3

APITYPE 3

For an APITYPE value of 3, the ASYNC parameter must be either 'ECB' or 'EXIT'.

UEEXIT

Optional input parameter. A doubleword value as follows:

- A fullword specifying the entry point address of the user unsolicited event exit.
- A fullword specifying the token that will be presented to the unsolicited event exit at invocation.

ASYNC

Optional input parameter. One of the following:

- The literal 'NO' indicating no asynchronous support.

- The literal 'ECB' indicating the asynchronous support using ECBs is to be used.
- The combination of the literal 'EXIT' and the address of a doubleword value as follows:
 - A fullword specifying the entry point address of the user's asynchronous event exit.
 - A fullword specifying the token which will be presented to the asynchronous event exit at invocation.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

IOCTL

The IOCTL macro is used to control certain operating characteristics for a socket.

Before you issue an IOCTL macro, you must load a value representing the characteristic that you want to control in COMMAND.

Note: IOCTL can be used only with programming languages that support address pointers.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

S

Input parameter. A value or the address of a halfword binary number specifying the socket to be controlled.

COMMAND

Input parameter. To control an operating characteristic, set this field to one of the following symbolic names. A value in a bit mask is associated with each symbolic name. By specifying one of these names, you are turning on a bit in a mask that communicates the requested operating characteristic to TCP/IP.

Value

Description

'FIONBIO'

Sets or clears blocking status.

'FIONREAD'

Returns the number of immediately readable bytes for the socket.

'SIOCATMARK'

Determines whether the current location in the data input is pointing to out-of-band data.

'SIOCGHOMEIF6'

Requests all IPv6 home interfaces.

Note: To request OSM interfaces, the application must have READ authorization to the EZB.OSM.*sysname.tcpname* resource.

When the SIOCGHOMEIF6 IOCTL is issued, the REQARG must contain a Network Configuration Header. The NETCONFHDR is defined in the SYS1.MACLIB(BPXIOCC6). The following fields are input fields and must be filled out:

NchEyeCatcher

Contains Eye Catcher '6NCH'.

NchIoctl

Contains the command code.

NchBufferLength

Buffer length of storage pointed to by NchBufferPTR. This buffer needs to be large enough to contain all the IPv6 interface records. Each interface record is length of HOMEIFADDRESS. If the buffer is not large enough, then errno will be set to ERANGE and the NchNumEntryRet will be set to number of interfaces. Based on NchNumEntryRet and size of HOMEIFADDRESS, calculate the necessary storage to contain the entire list.

NchBufferPtr

This is the pointer to an array of HOMEIF structures returned on a successful call. The size depends on the number of qualifying interfaces returned.

NchNumEntryRet

If return code is 0, this will be set to number of HOMEIFADDRESS returned. If errno is ERANGE, this will be set to number of qualifying interfaces. No interfaces are returned. Recalculate the NchBufferLength based on this value times the size of HOMEIFADDRESS.

'SIOCGIFADDR'

Requests the IPv4 network interface address for an interface name. For the address format, see the IOCN_SADDRIF field in the SYS1.MACLIB(BPXIOCC) macro.

'SIOCGIFBRDADDR'

Requests the IPv4 network interface broadcast address for an interface name. For the address format, see the IOCN_SADDRIFBROADCAST field in the SYS1.MACLIB(BPXIOCC) macro.

'SIOCGIFCONF'

Requests the IPv4 network interface configuration. The configuration consists of a variable number of 32-byte arrays. For the structure format, see the IOCN_IFREQ field in the SYS1.MACLIB(BPXIOCC) macro.

- When IOCTL is issued, the first word in REQARG must contain the length (in bytes) of the array to be returned, and the second word in REQARG should be set to the number of interfaces requested times 32 (one address structure for each network interface). The maximum number of array elements that TCP/IP Services will return is 100.
- When IOCTL is issued, RETARG must be set to the beginning of the area in your program's storage, which is reserved for the array that is to be returned by IOCTL.
- The COMMAND 'SIOGIFCONF' returns a variable number of network interface configurations.

'SIOCGIFDSTADDR'

Requests the IPv4 network interface destination address. For the address format, see the IOCN_SADDRIFDEST field in the SYS1.MACLIB(BPX1IOCC) macro.

'SIOCGIFMTU'

Requests the IPv4 network interface MTU (maximum transmission unit). For the MTU format, see the IOCN_MTU_SIZE field in the SYS1.MACLIB(BPX1IOCC) macro.

'SIOCGIFNAMEINDEX'

Requests all interface names and indexes including local loopback but excluding VIPAs. Information is returned for both IPv4 and IPv6 interfaces whether they are active or inactive. For IPv6 interfaces, information is returned to an interface only if it has at least one available IP address. See [z/OS Communications Server: IPv6 Network and Appl Design Guide](#) for more information.

Note: To request OSM interfaces, the application must have READ authorization to the EZB.OSM.*sysname.tcpname* resource.

The configuration consists of the IF_NAMEINDEX structure which is defined in SYS1.MACLIB(BPX1IOCC).

- When the SIOCGIFNAMEINDEX IOCTL is issued, REQARG must contain the length of application storage (in bytes) being used to contain the returned IF_NAMEINDEX structure. The formula to compute this length is as follows:
 1. Determine the number of interfaces expected to be returned upon successful completion of this command.
 2. Multiply the number of interfaces by the array element (size of IF_NIINDEX, IF_NINAME, and IF_NIEXT) to determine the size of the array element.
 3. To the size of the array add the size of IF_NITOTALIF and IF_NIENTRIES to determine the total number of bytes needed to accommodate the name and index information returned.

Upon successful completion of this call, the stack returns the number of entries required by the way of the IF_NITOTALIF field in the storage referenced by RETARG.

- When IOCTL is issued, RETARG must be set to the address of the beginning of the area in your program's storage which is reserved for the IF_NAMEINDEX structure that is to be returned by IOCTL.
- The command 'SIOCGIFNAMEINDEX' returns a variable number of all the qualifying network interfaces.

'SIOCGIPMSFILTER'

Requests a list of the IPv4 source addresses that comprise the source filter, with the current mode on a given interface and a multicast group for a socket. The source filter can include or exclude the set of source address, depending on the filter mode (MCAST_INCLUDE or MCAST_EXCLUDE). When the SIOCGIPMSFILTER IOCTL is issued, the REQARG parameters must contain an IP_MSFILTER structure, which is defined in SYS1.MACLIB(BPX1IOCC). The IP_MSFILTER option must include an interface address (input), a multicast address (input), filter mode (output), the number of source addresses in the following array (input and output), and an array of source addresses (output). On input, the number of source addresses is the number of source addresses that fit in the input array. On output, the number of source addresses contains the total number of source filters in the source filter list for the multicast group. If the application does not know the size of the source list prior to processing, it makes a reasonable guess (for example, 0), and if

when the call completes the number of source addresses is a greater value, then the IOCTL can be repeated with a larger buffer. On output, the number of source addresses is always updated to be the total number of sources in the filter, but the array holds as many source addresses as will fit, up to the minimum of the array size that is passed in as the input number.

Calculate the size of IP_MSFILTER value as follows:

1. Determine the expected number of source addresses.
2. Multiply the number of source addresses by the array element (size of the IMSF_SrcEntry value) to determine the size of all array elements.
3. Add the size of all array elements to the size of the IMSF_Header value to determine the total number of bytes that are needed to accommodate the source addresses information that is returned.

'SIOCGMONDATA'

Returns TCP/IP stack IPv4 and IPv6 statistical counters. REQARG must point to a MonDataIn structure. The counters are returned in a MonDataOut structure. Both of these structures are defined in EZBZMONP in SEZANMAC.

Note: The ARP counter data provided differs depending on the type of device. See the [z/OS Communications Server: IP Configuration Guide](#) for information about devices that support ARP Offload and what is supported for each device.

'SIOCGMSFILTER'

Requests a list of the IPv4 or IPv6 source addresses that comprise the source filter, with the current mode on a given interface index and a multicast group for a socket. The source filter can include or exclude the set of source address, depending on the filter mode (MCAST_INCLUDE or MCAST_EXCLUDE). When the SIOCGMSFILTER IOCTL is issued, the REQARG parameter must contain a GROUP_FILTER structure, which is defined in SYS1.MACLIB(BPXYIOCC). The GROUP_FILTER option must include an interface index (input), a sockaddr_storage structure of the multicast address (input), filter mode (output), the number of source addresses in the following array (input and output), and an array of the sockaddr_storage structure of source addresses (output). On input, the number of source addresses is the number of source addresses that will fit in the input array. On output, the number of source addresses contains the total number of source filters in the source filter list for the multicast group. If the application does not know the size of the source list prior to processing, it can make a reasonable guess (for example, 0), and if when the call completes the number of source addresses is a greater value, the IOCTL can be repeated with a buffer that is large enough. That is, on output, the number of source addresses is always updated to be the total number of sources in the filter, but the array holds as many source addresses as will fit, up to the minimum of the array size passed in as the input number.

The application calculates the size of GROUP_FILTER value in the following way:

1. Determines the expected number of source addresses.
2. Multiplies the number of source addresses by the array element (size of the GF_SrcEntry value) to determine the size of all array elements.
3. Adds the size of all array elements to the size of the GF_Header value to determine the total number of bytes that are needed to accommodate the source-address information that is returned.

'SIOCGPARTNERINFO'

Provides an interface for an application to retrieve security credentials about its partner. When you issue the SIOCGPARTNERINFO IOCTL, the REQARG parameter must contain a PartnerInfo structure as defined by the EZBPINFA macro in SEZANMAC. For more information about using the SIOCGPARTNERINFO IOCTL, see [z/OS Communications Server: IP Programmer's Guide and Reference](#).

'SIOCGSPLXFQDN'

Requests the fully qualified domain name for a given server and group name in a sysplex. This is a special purpose command to support applications that have registered with WorkLoad Manager (WLM) for connection optimization services by way of the DNS. When IOCTL is issued,

REQARG and RETARG must use the address structure sysplexFqDn, which contains the pointer for sysplexFqDnData structure. The fully qualified domain name is returned in the domainName field of sysplexFqDnData. The group name and the server name can be passed using the groupName and serverName fields of sysplexFqDnData structure. Their structures are defined in the EZBZSDNP MACRO file.

'SIOCSAPPLDATA'

The SIOCSAPPLDATA IOCTL enables an application to set 40 bytes of user-specified application data against a socket endpoint. You can use this application data to identify socket endpoints in interfaces such as Netstat, SMF, or network management applications. When you issue the SIOCSAPPLDATA IOCTL, the REQARG parameter must contain a SetApplData structure as defined by the EZBYAPPL macro. See [z/OS Communications Server: IP Programmer's Guide and Reference](#) for more information about programming the SIOCSAPPLDATA IOCTL.

SetAD_buffer: The user-defined application data is 40 bytes of data that identifies the endpoint with the application. You can obtain this application data from the following sources:

- Netstat reports. The information is displayed in the ALL/-A report. If you use the APPLDATA modifier, then the information also is displayed on the ALLConn/-a and COnn/-c reports.
- The SMF 119 TCP connection termination record. See [TCP connection termination record \(subtype 2\)](#) in [z/OS Communications Server: IP Programmer's Guide and Reference](#) for more information.
- Network management interfaces. See [IPSec NMI](#) in [z/OS Communications Server: IP Programmer's Guide and Reference](#) for more information.

Consider the following guidelines:

- The application must document the content, format and meaning of the ApplData strings that it associates with the sockets that it owns.
- The application should uniquely identify itself with printable EBCDIC characters at the beginning of the string. Strings beginning with 3-character IBM product identifiers, such as TCP/IP's EZA or EZB, are reserved for IBM use. IBM product identifiers begin with a letter in the range A-I.
- Use printable EBCDIC characters for the entire string to enable searching with Netstat filters.

Tip: Separate application data elements with a blank for easier reading.

'SIOCSIPMSFILTER'

Sets a list of the IPv4 source addresses that comprise the source filter, with the current mode on a given interface and a multicast group for a socket. The source filter can include or exclude the set of source address, depending on the filter mode (MCAST_INCLUDE or MCAST_EXCLUDE). When the SIOCSIPMSFILTER IOCTL is issued, the REQARG value must contain an IP_MSFILTER structure, which is defined in SYS1.MACLIB(BPXYIOCC). The IP_MSFILTER option must include an interface address, a multicast address, filter mode, the number of source addresses in the following array, and an array of source addresses.

Calculate the size of the IP_MSFILTER structure as follows:

1. Determine the expected number of source addresses.
2. Multiply the number of source addresses by the array element (size of IMSF_SrcEntry) to get the size of all array elements.
3. Add the size of all array elements to the size of the IMSF_Header value to get the total number of bytes needed to accommodate the source addresses information that is returned.

'SIOCSMSFILTER'

Sets a list of the IPv4 or IPv6 source addresses that comprise the source filter, with the current mode on a given interface index and a multicast group for a socket. The source filter can include or exclude the set of source address, depending on the filter mode (MCAST_INCLUDE or MCAST_EXCLUDE). When the SIOCSMSFILTER IOCTL is issued, the REQARG parameter must contain a GROUP_FILTER option structure, which is defined in SYS1.MACLIB(BPXYIOCC). The GROUP_FILTER option must include an interface index, a sockaddr_storage structure of the

multicast address, filter mode, the number of source addresses in the following array, and an array of the sockaddr_storage structure of source addresses.

Calculate the size of the GROUP_FILTER value as follows:

1. Determine the expected number of source addresses.
2. Multiply the number of source addresses by the array element (size of the GF_SrcEntry value) to determine the size of all array elements.
3. Add the size of all array elements to the size of the GF_Header value to determine the total number of bytes needed to accommodate the source addresses information that is returned.

'SIOCSPARTNERINFO'

The SIOCSPARTNERINFO IOCTL sets an indicator to retrieve the partner security credentials during connection setup and saves the information. In this way, an application can issue a SIOCGPARTNERINFO IOCTL without suspending the application or can at least minimize the time it takes to retrieve the information. When you issue the SIOCSPARTNERINFO IOCTL, the REQARG parameter must contain a constant value, PI_REQTYPE_SET_PARTNERDATA, as defined by the EZBPINFA macro in SEZANMAC. For more information about using the SIOCSPARTNERINFO IOCTL, see [z/OS Communications Server: IP Programmer's Guide and Reference](#).

SIOCTLSTLSCTL

Controls Application Transparent Transport Layer Security (AT-TLS) for the connection. REQARG and RETARG must contain a TTLS_IOCTL structure. If a partner certificate is requested, the TTLS_IOCTL must include a pointer to additional buffer space and the length of that buffer. Information is returned in the TTLS_IOCTL structure. If a partner certificate is requested and one is available, it is returned in the additional buffer space. The TTLS_IOCTL structure for assembler programs is defined in EZBZTLSP in SEZANMAC. For details about usage, see the [Application Transparent Transport Layer Security \(AT-TLS\) information in z/OS Communications Server: IP Programmer's Guide and Reference](#).

Restriction: Use of this ioctl for functions other than query requires that the AT-TLS policy mapped to the connection be defined with the ApplicationControlled parameter set to On.

REQARG and RETARG

Point to arguments that are passed between the calling program and IOCTL. The length of the argument is determined by the COMMAND request. REQARG is an input parameter or an output parameter and is used to pass and receive arguments to and from IOCTL. RETARG is an output parameter and is used for arguments returned by IOCTL.

For the lengths and meanings of REQARG and RETARG for each COMMAND type, see [Table 16 on page 284](#).

Table 16. IOCTL macro arguments

COMMAND/CODE	SIZE	REQARG	SIZE	RETARG
FIONBIO X'8004A77E'	4	Set socket mode to: X'00'=blocking; X'01'=nonblocking.	0	Not used.
FIONREAD X'4004A77F'	0	Not used.	4	Number of characters available for read.
SIOCATMARK X'4004A707'	0	Not used.	4	X'00'= not at OOB data X'01'= at OOB data .
SIOCGHOMEIF6 X' C014F608'	20	NetConfHdr		See NETCONFHDR in macro BPXYIOC6.

Table 16. IOCTL macro arguments (continued)

COMMAND/CODE	SIZE	REQARG	SIZE	RETARG
SIOCGIFADDR X'C020A70D'	32	First 16-bytes - interface name. Last 16-bytes - not used.	32	Network interface address, see the IOCN_SADDRIF field in the SYS1.MACLIB (BPXYIOCC) macro for format.
SIOCGIFBRDADDR X'C020A712'	32	First 16-bytes - interface name. Last 16-bytes - not used.	32	Network interface address, see the IOCN_SADDRIF BROADCAST field in the SYS1.MACLIB (BPXYIOCC) macro for format.
SIOCGIFCONF X'C008A714'	8	First 4 bytes- size of return buffer. Last 4 bytes - address of return buffer.	See note ¹ .	
SIOCGIFDSTADDR X'C020A70F'	32	First 16-bytes - interface name. Last 16-bytes - not used.	32	Destination interface address, see the IOCN_SADDRIFDEST field in the SYS1.MACLIB (BPXYIOCC) macro for format.
SIOCGIFMTU X'C020A726'	32	First 16-bytes - interface name. Last 16-bytes - not used.	32	IPv4 interface MTU (maximum transmission unit), see the IOCN_MTUSIZE field in the SYS1.MACLIB (BPXYIOCC) macro for format.
SIOCGIFNAMEINDEX X'4000F603'	4	First 4 bytes size of return buffer.		See IF_NAMEINDEX in macro BPXYIOCC.
SIOCGIPMSFILTER X'C000A724'	—	See IP_MSFILTER structure in macro BPXYIOCC. See note 3.	0	Not used
SIOCGMONDATA X'C018D902'	—	See MONDATAIN structure in macro EZBZMONP.	—	See MONDATAOUT structure in macro EZBZMONP.
SIOCGMSFILTER X'C000F610'	—	See GROUP_FILTER structure in macro BPXYIOCC. See note 4.	0	Not used
SIOCGPARTNERINFO X'C000F612'	—	For the PartnerInfo structure layout, see SEZANMAC(EZBPINFA). See note 5.	0	Not used
SIOCGSPLXFQDN X'C018D905'	408 ²	See sysplexFqDn and sysplexFqDnData in macro EZBZSDNP.	408 ²	See sysplexFqDn and sysplexFqDnData in macro EZBZSDNP.
SIOCSAPPLDATA X'8018D90C'	—	See SETAPPLDATA structure in macro EZBYAPPL.	0	Not used
SIOCSIPMSFILTER X'8000A725'	—	See IP_MSFILTER structure in macro BPXYIOCC. See note 3.	0	Not used

Table 16. IOCTL macro arguments (continued)

COMMAND/CODE	SIZE	REQARG	SIZE	RETARG
SIOCSMSFILTER X'8000F611'	—	See GROUP_FILTER structure in macro BPXYIOCC. See note 4.	0	Not used
SIOCSPARTNERINFO X'8004F613'	4	See PI_REQTYPE_SET_PARTNERDATA in SEZANMAC(EZBPINFA).	0	Not used
SIOCTLTLCTL X'C038D90B'	56	For IOCTL structure layout, see SEZANMAC (EZBZTLSP).	56	For IOCTL structure layout, see SEZANMAC (EZBZTLSP).

Notes:

1. The second 4-bytes in the RETARG is the address of the user buffer containing an array of 32-byte socket name structures (see IOCN_IFREQ in the SYS1.MACLIB(BPXYIOCC) macro for format). Each interface is assigned a 32-byte array element and the REQARG value should be set to the number of interfaces times 32. TCP/IP services can return up to 100 array elements.
2. REQARG and RETARG must contain both sysplexFqDn and sysplexFqDnData.
3. The size of the IP_MSFILTER structure must be equal to or greater than the size of the IMSF_Header.
4. The size of the GROUP_FILTER structure must be equal to or greater than the size of the GF_Header.
5. The size of the PartnerInfo structure must be equal to or greater than the PI_FIXED_SIZE value.

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, ERRNO contains a valid error number. Otherwise, ignore ERRNO.

See Appendix B, “Socket call error return codes,” on page 743 for information about ERRNO return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check ERRNO for an error code.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte ECB posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the ECB has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

LISTEN

Only servers use the LISTEN macro. The LISTEN macro:

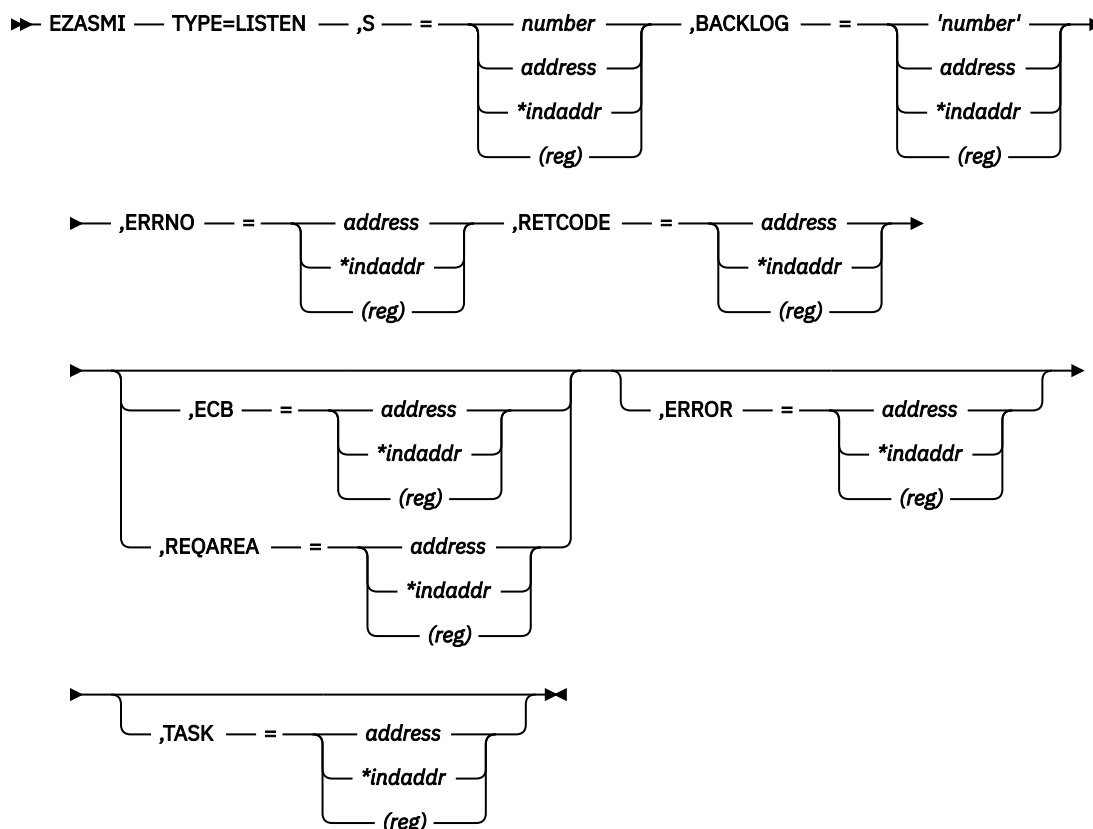
- Completes the bind, if BIND has not already been called for the socket. If the BIND has already been called for in the socket, the LISTEN macro uses what was specified in the BIND call.
- Creates a connection-request queue of a specified number of entries for incoming connection requests.

The LISTEN macro 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 macro. The original socket continues to listen for additional connection requests.

Note: Concurrent servers and iterative servers use this macro. An iterative server handles one client at a time. A concurrent server receives connection requests from multiple clients and creates subtasks that process the client requests. When a subtask is created, the concurrent server gets a new socket, passes the new socket to the subtask, and dissociates itself from the connection. The TCP/IP Listener program is an example of a concurrent server.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space



Keyword Description

S

Input parameter. A value or the address of a halfword binary number specifying the socket descriptor.

BACKLOG

Input parameter. A value (enclosed in single quotation marks) or the address of a fullword binary number specifying the number of messages that can be backlogged.

Rule: The BACKLOG value specified on the LISTEN macro is limited to the value configured by the SOMAXCONN statement in the stack's TCPIP PROFILE (default=10); no error is returned if a larger backlog is requested. You might need to update SOMAXCONN if a larger backlog is desired. See the [SOMAXCONN](#) information in the [z/OS Communications Server: IP Configuration Reference](#) for details.

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, ERRNO contains a valid error number. Otherwise, ignore ERRNO.

See [Appendix B, "Socket call error return codes,"](#) on page 743 for information about ERRNO return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check ERRNO for an error code.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte ECB posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the ECB has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

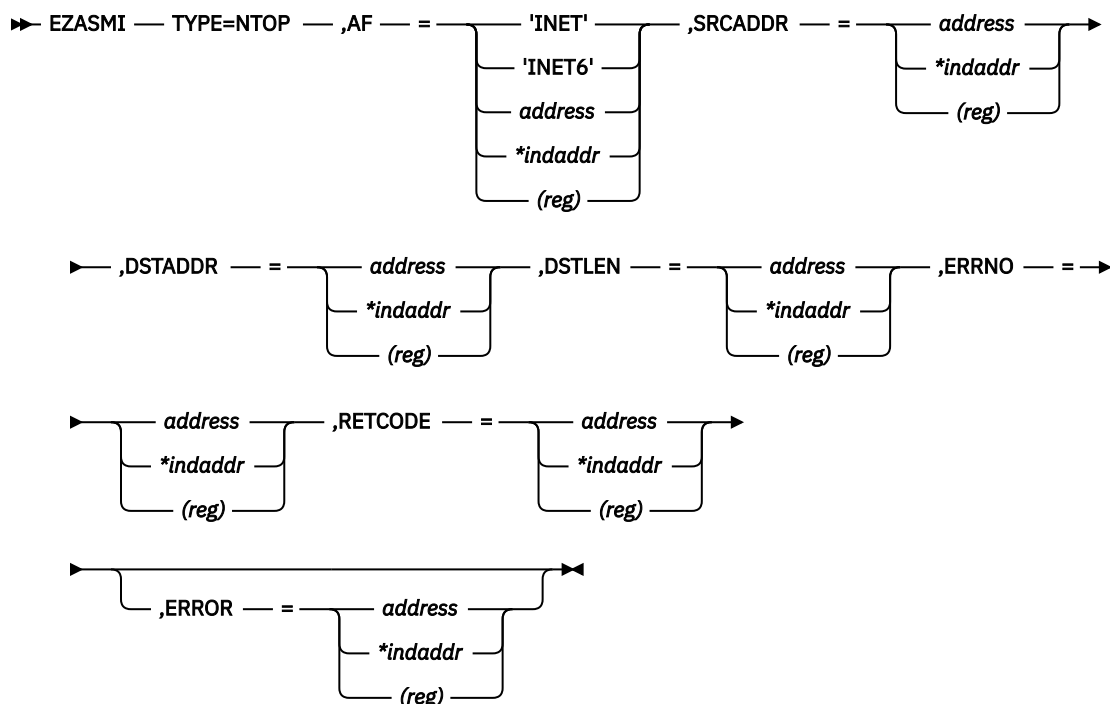
Input parameter. The location of the task storage area in your program.

NTOP

The NTOP macro converts an IP address from its numeric binary form into a standard text presentation form. On successful completion, NTOP returns the converted IP address in the buffer provided.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

AF

Input parameter. Specify one of the following:

Value Description

'INET' or a decimal '2'

Indicates the address being converted is an IPv4 address.

'INET6' or a decimal '19'

Indicates the address being converted is an IPv6 address.

AF can also indicate a fullword binary number specifying the address family.

SRCADDR

Input parameter. A field containing the numeric binary form of the IPv4 or IPv6 address being converted. For an IPv4 address, this field must be a fullword. For an IPv6 address, this field must be 16 bytes. The address must be in network byte order.

DSTADDR

Input parameter. A field used to receive the standard text presentation form of the IPv4 or IPv6 address being converted. For IPv4 the address will be in dotted-decimal format and for IPv6 the address will be in colon-hex format. The size of the converted IPv4 address will be a maximum of 15 bytes and the size of the converted IPv6 address will be a maximum of 45 bytes. Consult the value returned in DSTLEN for the actual length of the value in DSTADDR.

DSTLEN

Initially, an input parameter. The address of a binary halfword field that is used to specify the length of the DSTADDR field on input and upon a successful return will contain the length of the converted IP address.

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, ERRNO contains a valid error number. Otherwise, ignore the ERRNO field.

See Appendix B, “Socket call error return codes,” on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check ERRNO for an error code.

ERROR

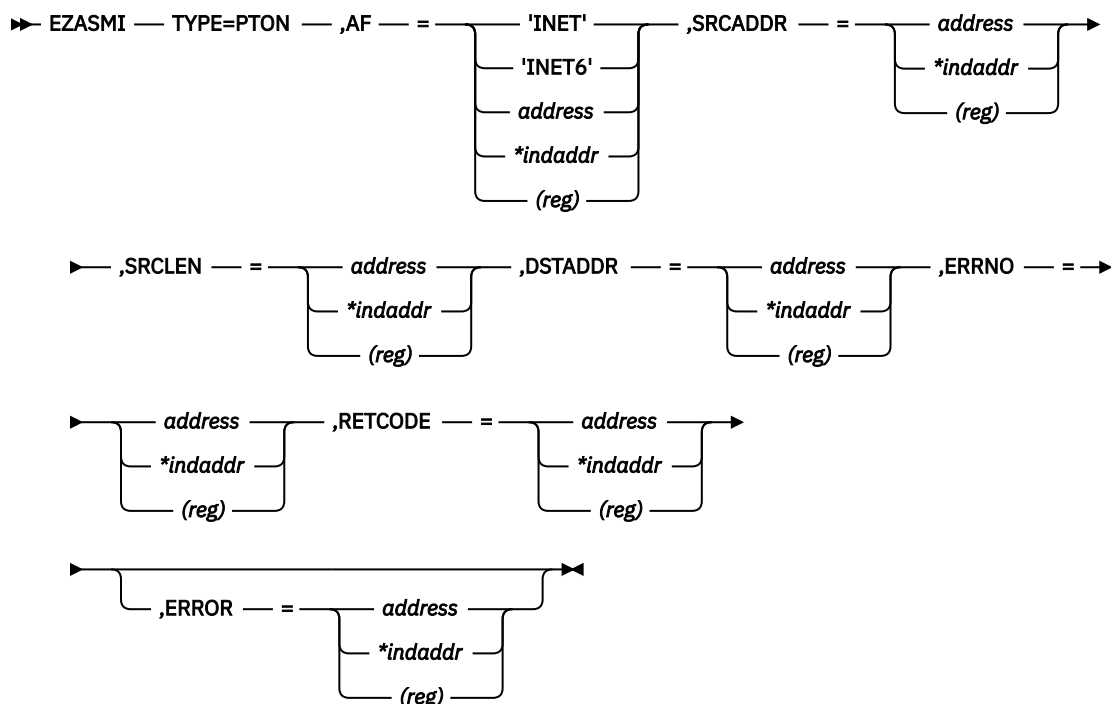
Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

PTON

The PTON macro converts an IP address in its standard text presentation form to its numeric binary form. On successful completion, PTON returns the converted IP address in the buffer provided.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

AF

Input parameter. Specify one of the following:

Value

Description

'INET' or a decimal '2'

Indicates the address being converted is an IPv4 address.

'INET6' or a decimal '19'

Indicates the address being converted is an IPv6 address.

AF can also indicate a fullword binary number specifying the address family.

SRCADDR

Input parameter. A field containing the standard text presentation form of the IPv4 or IPv6 address being converted. For IPv4 the address must be in dotted-decimal format and for IPv6 the address must be in colon-hex format. The size of the field for an IPv4 address must be 15 bytes and the size for an IPv6 address must be 45 bytes.

SRCLen

Input parameter. A binary halfword field that must contain the length of the IP address to be converted.

DSTADDR

A field used to receive the numeric binary form of the IPv4 or IPv6 address being converted in network byte order. For an IPv4 address, this field must be a fullword. For an IPv6 address, this field must be 16 bytes.

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, ERRNO contains a valid error number. Otherwise, ignore the ERRNO field.

See Appendix B, “Socket call error return codes,” on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check ERRNO for an error code.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

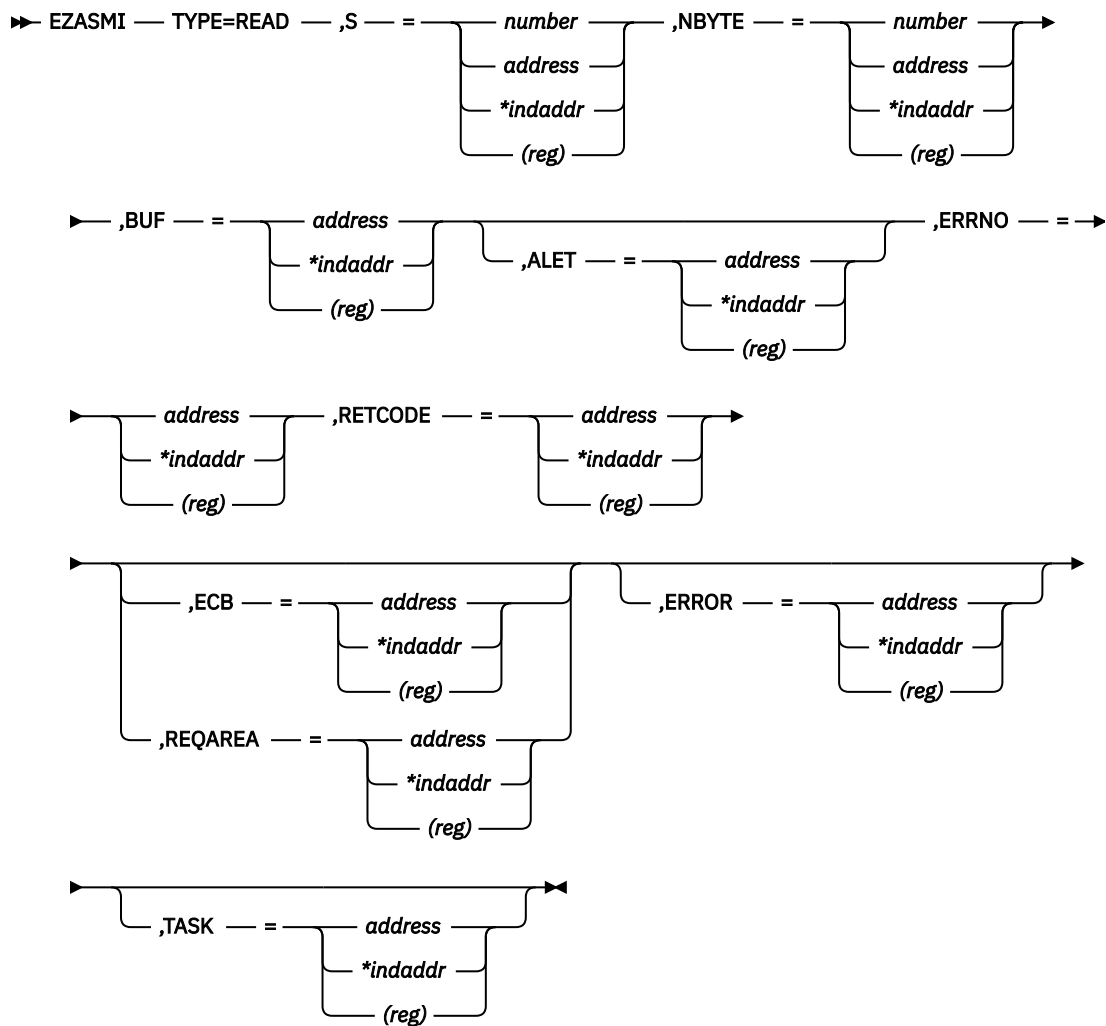
READ

The READ macro reads data on a socket and stores it in a buffer. The READ macro applies only to connected sockets.

For datagram sockets, the READ call returns the entire datagram that was sent. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard extra bytes.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

S

Input parameter. A value or the address of a halfword binary number specifying the socket that is going to read the data.

NBYTE

Input parameter. A fullword binary number set to the size of **BUF**. READ does not return more than the number of bytes of data in **NBYTE** even if more data is available.

BUF

On input, a buffer to be filled by completion of the call. The length of **BUF** must be at least as long as the value of **NBYTE**.

ALET

Optional input parameter. A fullword binary field containing the **ALET** or **BUF**. The default is 0 (primary address space).

If a nonzero ALET is specified, the ALET must represent a valid entry in the dispatchable unit access list (DU-AL) for the task issuing this call. Note that ALETs can be specified only for synchronous socket calls (for example, ECB/REQAREA cannot be specified). An exception to this is an ALET representing a SCOPE=COMMON data space.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, **ERRNO** contains a valid error number. Otherwise, ignore the **ERRNO** field.

See Appendix B, “Socket call error return codes,” on page 743 for information about **ERRNO** return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

0

A 0 return code indicates that the connection is closed and no data is available.

>0

A positive value indicates the number of bytes copied into the buffer.

-1

Check **ERRNO** for an error code.

ECB or REQAREA

Input parameter. This parameter is required if you are using **APITYPE=3**. It points to a 104-byte field containing the following information:

For ECB

A 4-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

READ returns up to the number of bytes specified by **NBYTE**. If less than the number of bytes requested is available, the READ macro returns the number currently available.

If data is not available for the socket and the socket is in blocking mode, the READ macro blocks the caller until data arrives. If data is not available and the socket is in nonblocking mode, READ returns a -1 and sets **ERRNO** to 35 (EWOULDBLOCK). See “[IOCTL](#)” on page 278 or “[FCNTL](#)” on page 218 for a description of how to set the nonblocking mode.

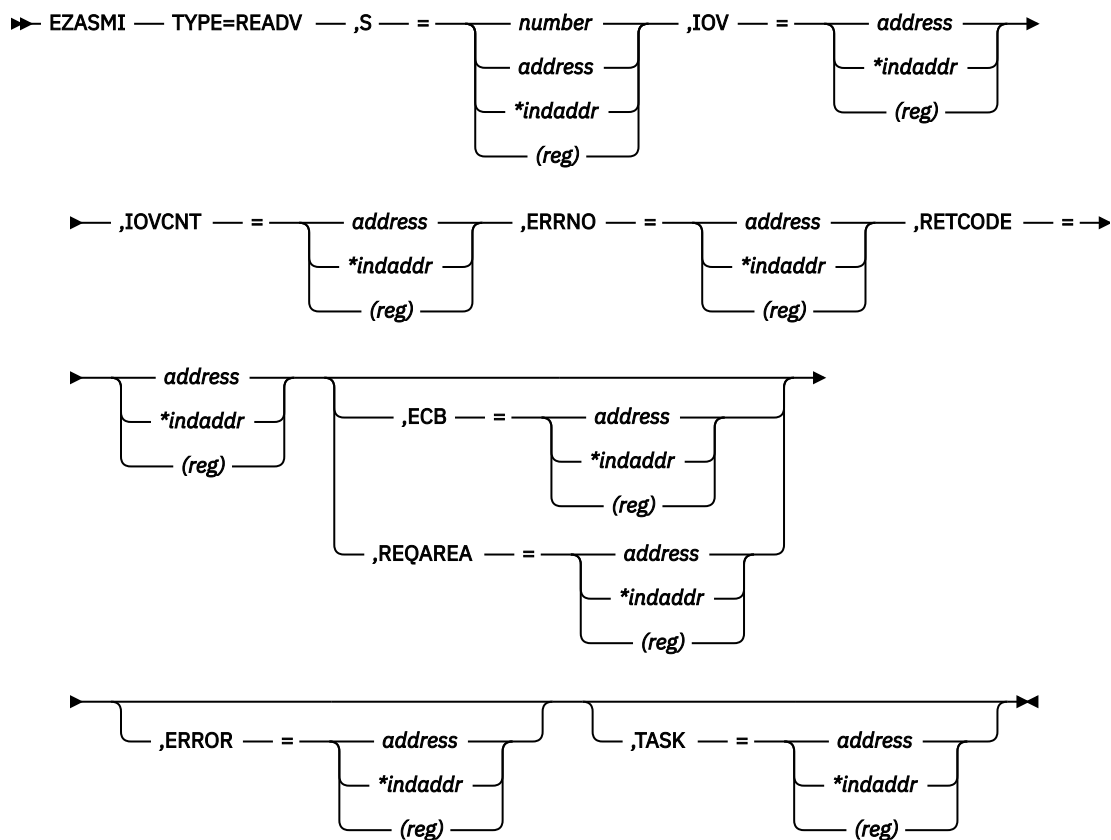
READV

The READV macro reads data on a socket and stores it in a set of buffers. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard extra bytes.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit.
Note: See "Addressability mode (Amode) considerations" under “ CALL instruction API environmental restrictions and programming requirements ” on page 389.	

ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

S

Input parameter. A value or the address of a halfword binary number specifying the descriptor of the socket into which the data is to be read.

IOV

An array of three fullword structures with the number of structures equal to the value in **IOVCNT** and the format of the structures as follows:

Fullword 1

Input parameter. A buffer to be filled by the completion of the call.

Fullword 2

Input parameter. The **ALET** for this buffer. If the buffer is in the primary address space, this should be zeros.

If a nonzero **ALET** is specified, the **ALET** must represent a valid entry in the dispatchable unit access list (DU-AL) for the task issuing this call. Note that **ALETs** can be specified only for synchronous socket calls (for example, ECB/REQAREA cannot be specified). An exception to this is an **ALET** representing a SCOPE=COMMON data space.

Fullword 3

Input parameter. The length of the data buffer referred to in Fullword 1.

IOVCNT

Input parameter. A fullword binary field specifying the number of data buffers provided for this call. The limit is 120.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, this contains an error number.

RETCODE

A fullword binary field that returns one of the following values:

Value**Description****0**

A 0 return code indicates that the connection is closed and no data is available.

>0

A positive value indicates the number of bytes copied into the buffer.

-1

Check **ERRNO** for an error code.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

RECV

The RECV macro receives data on a socket and stores it in a buffer. The RECV macro applies only to connected sockets. RECV can read the next message, but leaves the data in a buffer, and can read out-of-band data. RECV gives you the option of setting flags with the FLAGS parameter.

Note: Out-of-band data (called urgent data in TCP) appears to the application like a separate stream of data from the main data stream.

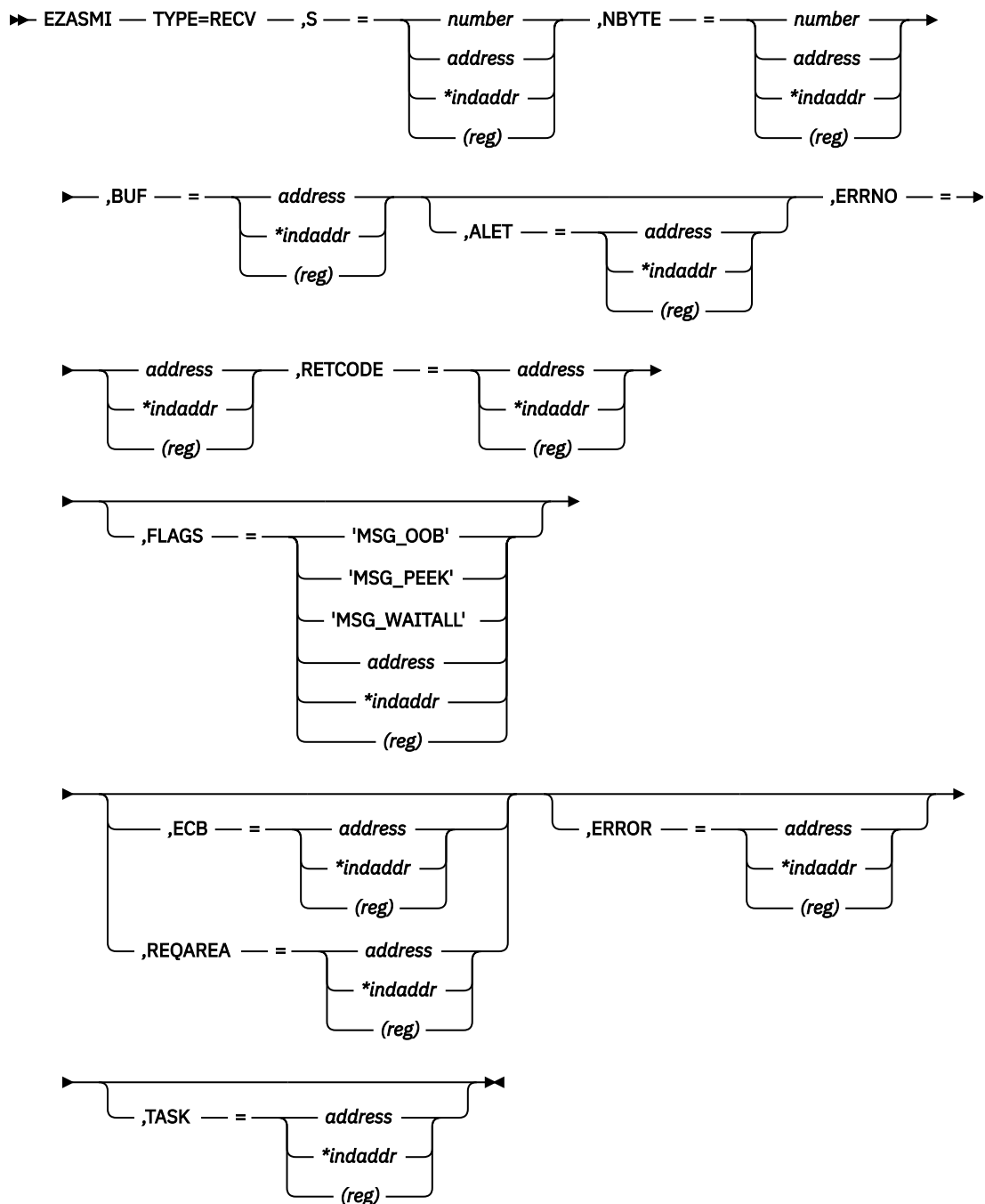
RECV returns the length of the incoming message or data. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard extra bytes.

For stream sockets, the data is processed like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and Application A sends 1000 bytes, each call to RECV can return 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place RECV in a loop that repeats the call until all data has been received.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.

Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

S

Input parameter. A value or the address of a halfword binary number specifying the socket descriptor.

NBYTE

Input parameter. A fullword binary number set to the size of BUF. RECV does not receive more than the number of bytes of data in NBYTE even if more data is available.

BUF

On input, a buffer to be filled by completion of the call. The length of BUF must be at least as long as the value of NBYTE.

ALET

Optional input parameter. A fullword binary field containing the ALET of BUF. The default is 0 (primary address space).

If a nonzero *ALET* is specified, the *ALET* must represent a valid entry in the dispatchable unit access list (DU-AL) for the task issuing this call. Note that *ALETs* can be specified only for synchronous socket calls (for example, ECB/REQAREA cannot be specified). An exception to this is an *ALET* representing a SCOPE=COMMON data space.

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, this field contains an error number. See Appendix B, “Socket call error return codes,” on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value**Description****0**

A 0 return code indicates that the connection is closed and no data is available.

>0

A positive value indicates the number of bytes copied into the buffer.

-1

Check ERRNO for an error code.

FLAGS

Input parameter. FLAGS can be a literal value or a fullword binary field.

Literal Value	Binary Value	Description
'MSG_OOB'	X'00000001'	Receive out-of-band data (stream sockets only). Out-of-band data can be read if the SO_OOBINLINE option is set for the socket regardless of whether the MSG_OOB flag is set.
'MSG_PEEK'	X'00000002'	Peek at the data, but do not destroy the data. If the peek flag is set, the next receive operation reads the same data.
'MSG_WAITALL'	X'00000040'	Requests that the function block until the full amount of requested data can be returned (stream sockets only). The function might return a smaller amount of data if the connection is terminated, if an error is pending, or if the SO_RCVTIMEO field is set and the timer expired for the socket.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte ECB posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the ECB has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

If data is not available for the socket and the socket is in blocking mode, the RECV macro blocks the caller until data arrives. If data is not available and the socket is in nonblocking mode, RECV returns a -1 and sets ERRNO to 35 (EWOULDBLOCK). See [“FCNTL” on page 218](#) or [“IOCTL” on page 278](#) for a description of how to set nonblocking mode.

RECVFROM

The RECVFROM macro receives data for a socket and stores it in a buffer. RECVFROM returns the length of the incoming message or data stream.

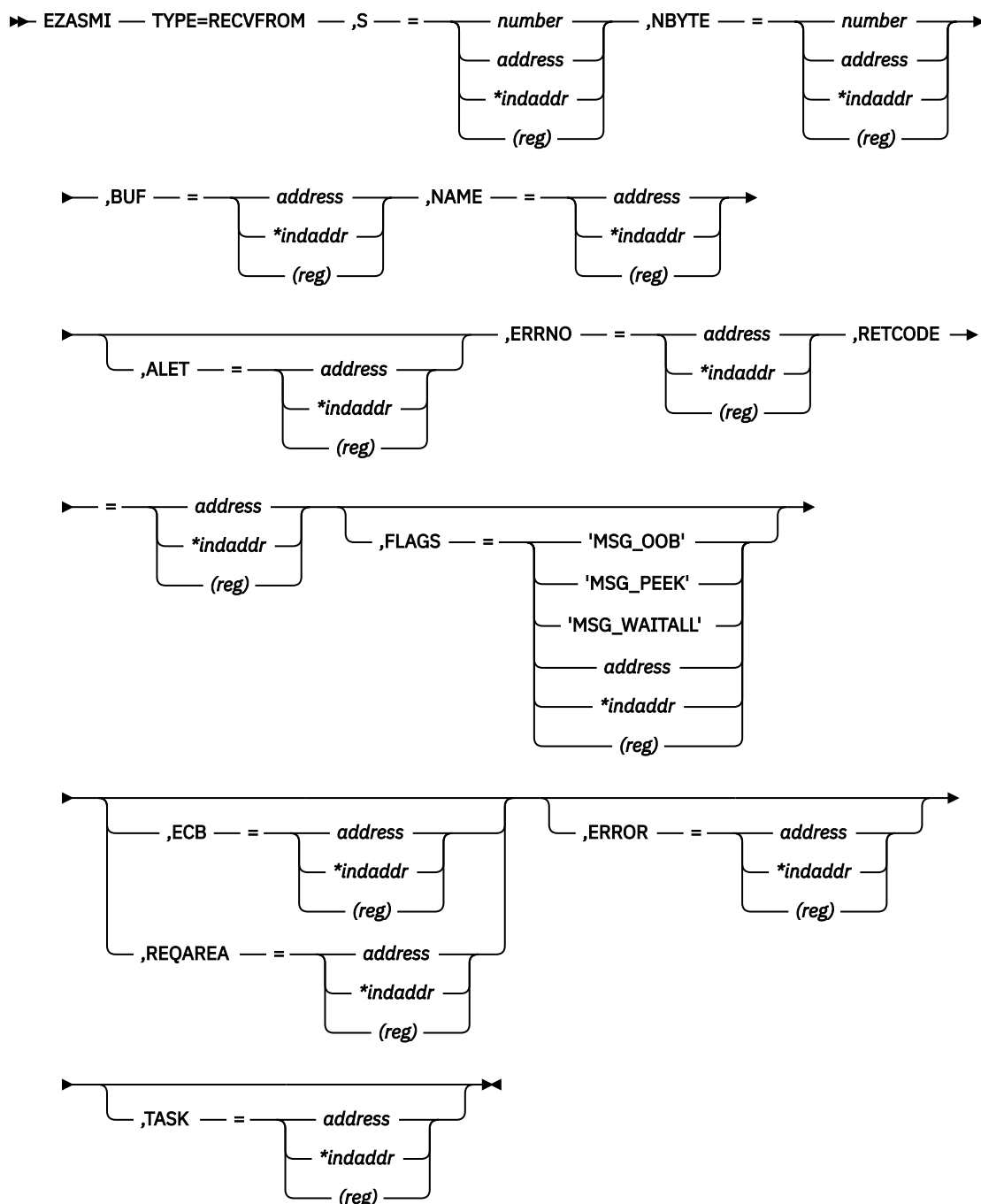
If data is not available for the socket designated by descriptor S, and socket S is in blocking mode, the RECVFROM call blocks the caller until data arrives.

If data is not available and socket S is in nonblocking mode, RECVFROM returns a -1 and sets ERRNO to 35 (EWOULDBLOCK). Because RECVFROM returns the socket address in the NAME structure, it applies to any datagram socket, whether connected or unconnected. See [“FCNTL” on page 218](#) or [“IOCTL” on page 278](#) for a description of how to set nonblocking mode. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard extra bytes.

For stream sockets, the data is processed as streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and Application A sends 1000 bytes, each call to this function can return 1 byte, or 10 bytes, or the entire 1000 bytes. Applications using stream sockets should place RECVFROM in a loop that repeats until all of the data has been received.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389 .
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

S

Input parameter. A value or the address of a halfword binary number specifying the socket to receive the data.

NBYTE

Input parameter. A value or the address of a fullword binary number specifying the length of the input buffer. NBYTE must first be initialized to the size of the buffer associated with NAME. On return the NBYTE contains the number of bytes of data received.

BUF

On input, a buffer to be filled by completion of the call. The length of BUF must be at least as long as the value of NBYTE.

NAME

Initially, the IPv4 or IPv6 application provides a pointer to a structure that will contain the peer socket name on completion of the call. If the NAME parameter value is nonzero, the IPv4 or IPv6 source address of the message is specified with the address of who sent the datagram. Include the SYS1.MACLIB(BPXYSOCK) macro to get the assembler mappings for the socket address structure. The socket address structure mappings begin at the SOCKADDR label. The AF_INET socket address structure fields start at the SOCK_SIN label. The AF_INET6 socket address structure fields start at the SOCK_SIN6 label.

The IPv4 socket address structure contains the following fields:

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

FAMILY

Output parameter. A halfword binary number specifying the IPv4 addressing family. The value for the IPv4 socket descriptor (S parameter) is a decimal 2, indicating AF_INET.

PORT

Output parameter. A halfword binary number specifying the port number of the sending socket.

IPv4-ADDRESS

Output parameter. A fullword binary number specifying the 32-bit IPv4 Internet address of the sending socket.

RESERVED

Output parameter. An 8-byte reserved field. This field is required, but is not used.

The IPv6 socket structure contains the following fields:

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

NAMELEN

Output parameter. A 1-byte binary field specifying the length of this IPv6 socket address structure. Any value specified by the use of this field is ignored when processed as input and the field is set to 0 when processed as output.

FAMILY

Output parameter. A 1-byte binary field specifying the IPv6 addressing family. The value for IPv6 socket descriptor (S parameter) is a decimal 19, indicating AF_INET6.

PORT

Output parameter. A halfword binary number specifying the port number of the sending socket.

FLOW-INFO

Output parameter. A fullword binary field specifying the traffic class and flow label. This value of this field is undefined.

IPv6-ADDRESS

Output parameter. A 16-byte binary field that is set to the 128-bit IPv6 Internet address, in network byte order, of the sending socket.

SCOPE-ID

A fullword binary field which identifies a set of interfaces as appropriate for the scope of the address carried in the IPv6-ADDRESS field. For a link scope IPv6-ADDRESS, SCOPE-ID contains the link index for the IPv6-ADDRESS. For all other address scopes, SCOPE-ID is undefined.

ALET

Optional input parameter. A fullword binary field containing the ALET of BUF. The default is 0 (primary address space).

If a nonzero *ALET* is specified, the *ALET* must represent a valid entry in the dispatchable unit access list (DU-AL) for the task issuing this call. Note that *ALET*s can only be specified for synchronous socket calls (for example, ECB/REQAREA cannot be specified). An exception to this is an *ALET* representing a SCOPE=COMMON data space.

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, this field contains an error number. See Appendix B, “Socket call error return codes,” on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

0

A 0 return code indicates that the connection is closed and no data is available.

>0

A positive value indicates the number of bytes transferred by the RECVFROM call.

-1

Check ERRNO for an error code.

FLAGS

Input parameter. FLAGS can be a literal value or a fullword binary field.

Literal Value	Binary Value	Description
'MSG_OOB'	X'00000001'	Receive out-of-band data (stream sockets only). Out-of-band data can be read if the SO_OOBINLINE option is set for the socket regardless of whether the MSG_OOB flag is set.
'MSG_PEEK'	X'00000002'	Peek at the data, but do not destroy the data. If the peek flag is set, the next receive operation reads the same data.
'MSG_WAITALL'	X'00000040'	Requests that the function block until the full amount of requested data can be returned (stream sockets only). The function might return a smaller amount of data if the connection is terminated, if an error is pending, or if the SO_RCVTIMEO field is set and the timer expired for the socket.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte ECB posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the ECB has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

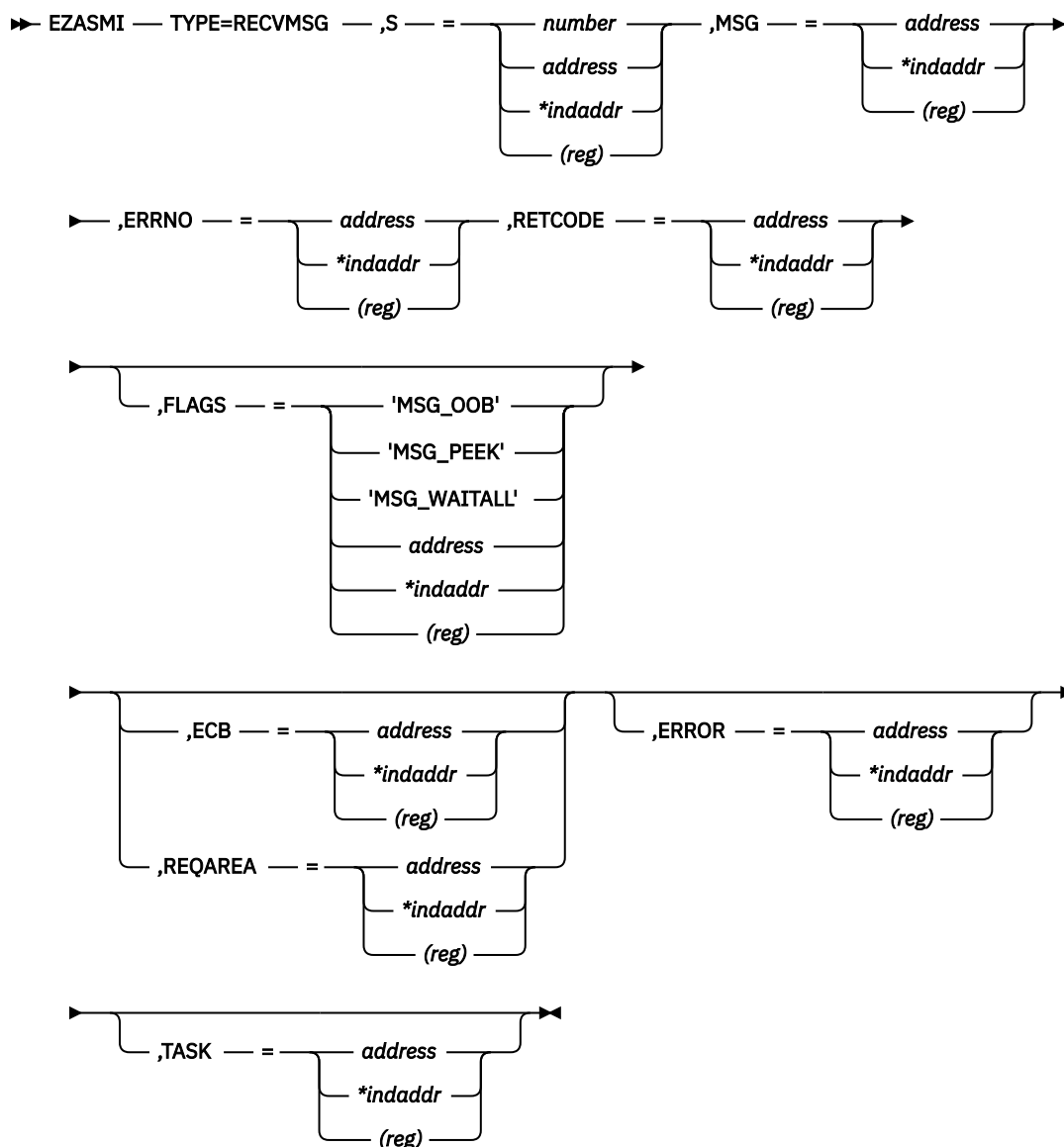
Input parameter. The location of the task storage area in your program.

RECVMSG

The RECVMSG macro receives messages on a socket with descriptor *s* and stores them in an array of message headers. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard extra bytes.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

S

Input parameter. A value or the address of a halfword binary number specifying the socket descriptor.

MSG

On input, this is a pointer to a message header into which the message is received on completion of the call.

NAME

On input, a pointer to a buffer where the sender's IPv4 or IPv6 address will be stored on completion of the call. The storage being pointed to should be for an IPv4 or IPv6 socket address. Include the SYS1.MACLIB(BPXYSOCK) macro to get the assembler mappings for the socket address structure. The socket address structure mappings begin at the SOCKADDR label. The AF_INET socket address structure fields start at the SOCK_SIN label. The AF_INET6 socket address structure fields start at the SOCK_SIN6 label.

The IPv4 socket address structure contains the following fields:

Field	
Description	

FAMILY

Output parameter. A halfword binary number specifying the IPv4 addressing family. The value for the IPv4 socket descriptor (S parameter) is a decimal 2, indicating AF_INET.

PORT

Output parameter. A halfword binary number specifying the port number of the sending socket.

IPv4-ADDRESS

Output parameter. A fullword binary number specifying the 32-bit IPv4 Internet address of the sending socket.

RESERVED

Output parameter. An 8-byte reserved field. This field is required, but is not used.

The IPv6 socket address structure contains the following fields:

Field	
Description	

NAMELEN

Output parameter. A 1-byte binary field specifying the length of this IPv6 socket address structure. Any value specified by the use of this field is ignored when processed as input and the field is set to 0 when processed as output.

FAMILY

Output parameter. A 1-byte binary field specifying the IPv6 addressing family. The value for the IPv6 socket descriptor (S parameter) is a decimal 19, indicating AF_INET6.

PORT

Output parameter. A halfword binary number specifying the port number of the sending socket.

FLOW-INFO

Output parameter. A fullword binary field specifying the traffic class and flow label. This value of this field is undefined.

IPv6-ADDRESS

Output parameter. A 16-byte binary field specifying the 128-bit IPv6 Internet address, in network byte order, of the sending socket.

SCOPE-ID

A fullword binary field which identifies a set of interfaces as appropriate for the scope of the address carried in the IPv6-ADDRESS field. For a link scope IPv6-ADDRESS, SCOPE-ID contains the link index for the IPv6-ADDRESS. For all other address scopes, SCOPE-ID is undefined.

IOV

On input, a pointer to an array of three fullword structures with the number of structures equal to the value in IOVCNT and the format of the structures as follows:

Fullword 1

Input parameter. The address of a data buffer.

Fullword 2

Input parameter. The **ALET** for this buffer. If the buffer is in the primary address space, this should be zeros.

If a nonzero *ALET* is specified, the *ALET* must represent a valid entry in the dispatchable unit access list (DU-AL) for the task issuing this call. Note that *ALETs* can only be specified for synchronous socket calls (for example, ECB/REQAREA cannot be specified). An exception to this is an *ALET* representing a SCOPE=COMMON data space.

Fullword 3

Input parameter. The length of the data buffer referenced in Fullword 1.

IOVCNT

On input, a pointer to a fullword binary field specifying the number of data buffers provided for this call.

ACCRIGHTS

On input, a pointer to the access rights received. This field is ignored.

ACCRLEN

On input, a pointer to the length of the access rights received. This field is ignored.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, this contains an error number.

RETCODE

Output parameter. A fullword binary field with the following values:

Value**Description****-1**

Call returned error. See **ERRNO** field.

0

Connection partner has closed connection.

>0

Number of bytes read.

FLAGS

Input parameter. **FLAGS** can be a literal value or a fullword binary field.

Literal Value	Binary Value	Description
'MSG_OOB'	X'00000001'	Receive out-of-band data (stream sockets only). Out-of-band data can be read if the SO_OOBLINE option is set for the socket regardless of whether the MSG_OOB flag is set.
'MSG_PEEK'	X'00000002'	Peek at the data, but do not destroy the data. If the peek flag is set, the next receive operation reads the same data.
'MSG_WAITALL'	X'00000040'	Requests that the function block until the full amount of requested data can be returned (stream sockets only). The function might return a smaller amount of data if the connection is terminated, if an error is pending, or if the SO_RCVTIMEO field is set and the timer expired for the socket.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

SELECT

In a process where multiple I/O operations can occur it is necessary for the program to be able to wait on one or several of the operations to complete. For example, consider a program that issues a READ to multiple sockets whose blocking mode is set. Because the socket would block on a READ macro, only one socket could be read at a time. Setting the sockets to nonblocking would solve this problem, but would require polling each socket repeatedly until data becomes available. The SELECT macro allows you to test several sockets and to process a later I/O macro only when one of the tested sockets is ready. This ensures that the I/O macro does not block.

Tip: You do not need to use the SELECT macro when you are using APITYPE=3 (asynchronous) calls. Simply issue the intended call and let the specified notification mechanism (ECB or exit routine) take effect.

To use the SELECT macro as a timer in your program, take either of the following actions:

- Set the read, write, and exception arrays to zeros.
- Do not specify MAXSOC.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Testing sockets

Read, write, and exception operations can be tested. The SELECT macro monitors activity on selected sockets to determine whether:

- A buffer for the specified sockets contains input data. If input data is available for a given socket, a read operation on that socket does not block.
- TCP/IP can accommodate additional output data. If TCP/IP can accept additional output for a socket, a write operation on the socket does not block.
- An exceptional condition occurs on a socket.
- A timeout occurs on the SELECT macro itself. A TIMEOUT period can be specified when the SELECT macro is issued.

Each socket descriptor is represented by a bit in a bit string. The length of this bit-mask array is dependent on the value in MAXSOC and must be a multiple of 4 bytes. See [“Selecting requests in a concurrent server program” on page 37](#) for more information.

Read operations

The ACCEPT, READ, READV, RECV, RECVFROM, and RECVMSG macros are read operations. A socket is ready for reading when data is received on it, or when an exception condition occurs.

To determine if a socket is ready for the read operation, set the appropriate bit in RSNDSK to 1 before issuing the SELECT macro. When the SELECT macro returns, the corresponding bits in the RRETMSK indicate sockets ready for reading.

Write operations

A socket is selected for writing, ready to be written, when:

- TCP/IP can accept additional outgoing data.
- A connection request is received in response to an ACCEPT macro.
- A CONNECT call for a nonblocking socket, which has previously returned ERRNO 36 (EINPROGRESS), completes the connection.

The WRITE, WRITEV, SEND, SENDMSG, or SENDTO macros block when the data to be sent exceeds the amount that TCP/IP can accept. To avoid this, you can precede the write operation with a SELECT macro to ensure that the socket is ready for writing. After a socket is selected for WRITE, your program can determine the amount of TCP/IP buffer space available by issuing the GETSOCKOPT macro with the SO_SNDBUF option.

To determine if a socket is ready for the write operation, set the appropriate bit in WSNDSK to 1.

Exception operations

For each socket to be tested, the SELECT macro can check for an exception condition. The exception conditions are:

- The calling program (concurrent server) has issued a GIVESOCKET command and the target subtask has successfully issued the TAKESOCKET call. When this condition is selected, the calling program (concurrent server) should issue CLOSE to dissociate itself from the socket.
- A socket has received out-of-band data. For this condition, a READ macro returns the out-of-band data before the program data.

To determine if a socket has an exception condition, use the ESNDSK character string and set the appropriate bits to 1.

Returning the results

For each event tested by a xSNDSK, a bit string records the results of the check. The bit strings are RRETMSK, WRETMSK, and ERETMSK for read, write, and exceptional events. On return from the SELECT macro, each bit set to 1 in the xRETMSK is a read, write, or exceptional event for the associated socket.

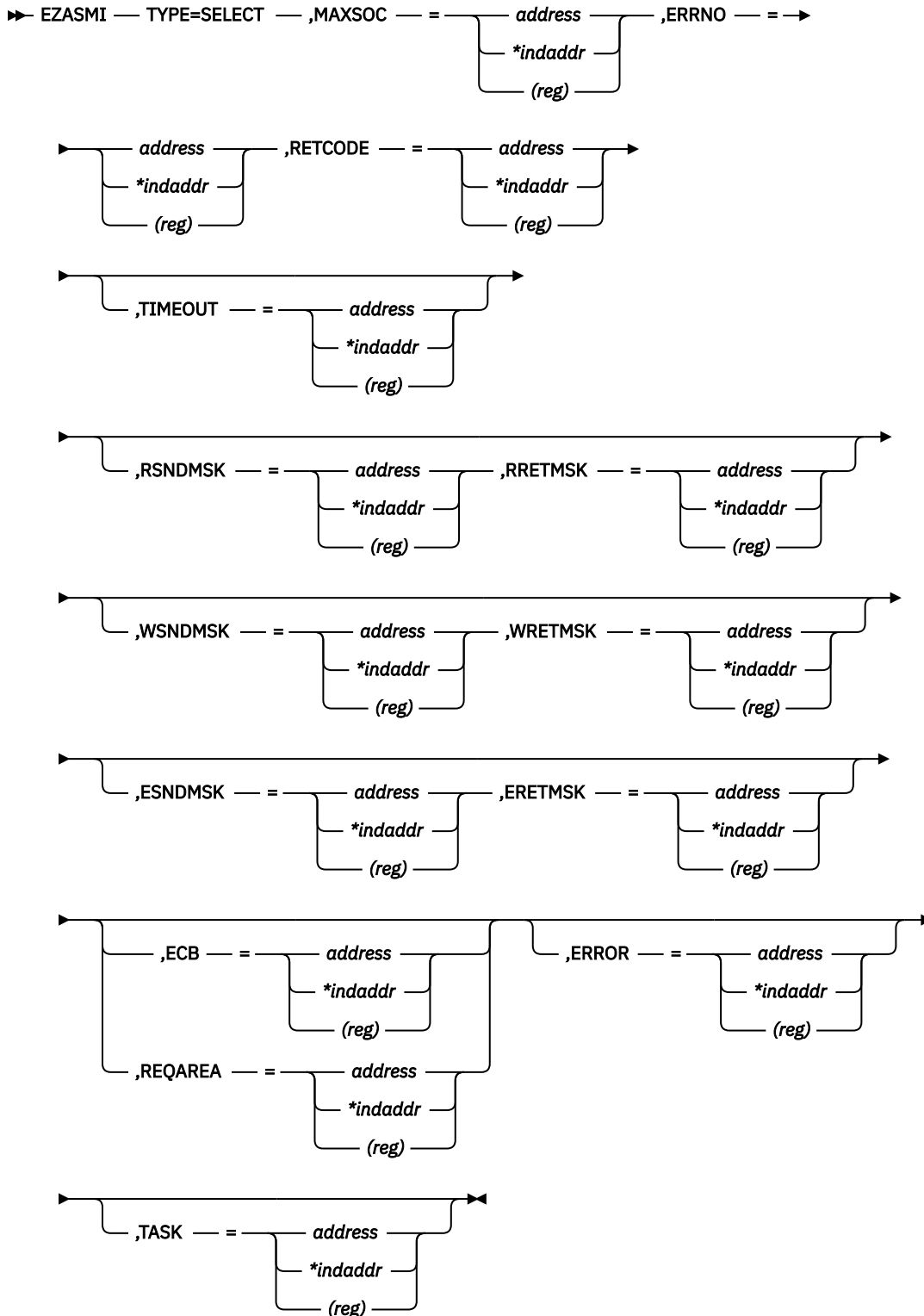
MAXSOC parameter

The SELECT call must test each bit in each string before returning results. For efficiency, the MAXSOC parameter can be used to specify the largest socket descriptor number that needs to be tested for any event type. The SELECT call tests only bits in the range 0 through the MAXSOC value minus one.

Example: If MAXSOC value is set to 50, the range would be 0 – 49.

TIMEOUT parameter

If the time in the TIMEOUT parameter elapses before an event is detected, the SELECT call returns and RETCODE is set to 0.



**Keyword
Description**

MAXSOC

Input parameter. A fullword binary field that specifies the largest socket descriptor value being checked. The SELECT call tests only bits that are in the range 0 through the MAXSOC value minus 1. For example, if you set the MAXSOC value to 50, the range is 0 – 49.

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, ERRNO contains a valid error number. Otherwise, ignore ERRNO.

See [Appendix B, “Socket call error return codes,” on page 743](#) for information about ERRNO return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value

Description

>0

Indicates the number of ready sockets in the three return masks.

Note: If the number of ready sockets is greater than 65535, only 65535 is reported.

=0

Indicates that the SELECT time limit has expired.

-1

Check ERRNO for an error code.

TIMEOUT

Input parameter.

If TIMEOUT is not specified, the SELECT call blocks until a socket becomes ready.

If TIMEOUT is specified, TIMEOUT is the maximum interval for the SELECT call to wait until completion of the call. If you want SELECT to poll the sockets and return immediately, TIMEOUT should be specified to point to a 0-valued TIMEVAL structure.

TIMEOUT is specified in the two-word TIMEOUTs as follows:

- TIMEOUT-SECONDS, word one of TIMEOUT, is the seconds component of the timeout value.
- TIMEOUT-MICROSEC, word two of TIMEOUT, is the microseconds component of the timeout value (0–999999).

For example, if you want SELECT to timeout after 3.5 seconds, set TIMEOUT-SECONDS to 3 and TIMEOUT-MICROSEC to 500000.

For APITYPE=3 with an ECB specified, the SELECT call will return immediately because it is asynchronous; the ECB will be POSTed when the timer pops.

RSNDMSK

Input parameter. The bit-mask array to control checking for read interrupts. If this parameter is not specified or the specified bit-mask is zeros, the SELECT does not check for read interrupts. The length of this bit-mask array is dependent on the value in MAXSOC and must be a multiple of 4 bytes. See [“Selecting requests in a concurrent server program” on page 37](#) for more information.

RRETMSK

Output parameter. The bit-mask array returned by the SELECT if RSNDMSK is specified. The length of this bit-mask array is dependent on the value in MAXSOC and must be a multiple of 4 bytes. See [“Selecting requests in a concurrent server program” on page 37](#) for more information.

WSNDMSK

Input parameter. The bit-mask array to control checking for write interrupts. If this parameter is not specified or the specified bit-mask is zeros, the SELECT does not check for write interrupts. The length of this bit-mask array is dependent on the value in MAXSOC and must be a multiple of 4 bytes. See [“Selecting requests in a concurrent server program” on page 37](#) for more information.

WREMSK

Output parameter. The bit-mask array returned by the SELECT if WSNDSK is specified. The length of this bit-mask array is dependent on the value in MAXSOC and must be a multiple of 4 bytes. See [“Selecting requests in a concurrent server program” on page 37](#) for more information.

ESNDMSK

Input parameter. The bit-mask array to control checking for exception interrupts. If this parameter is not specified or the specified bit-mask is zeros, the SELECT does not check for exception interrupts. The length of this bit-mask array is dependent on the value in MAXSOC and must be a multiple of 4 bytes. See [“Selecting requests in a concurrent server program” on page 37](#) for more information.

ERETMSK

Output parameter. The bit-mask array returned by the SELECT if ESNDMSK is specified. The length of this bit-mask array is dependent on the value in MAXSOC and must be a multiple of 4 bytes. See [“Selecting requests in a concurrent server program” on page 37](#) for more information.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte ECB posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the ECB has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

SELECTEX

The SELECTEX macro monitors a set of sockets, a time value, and an ECB or list of ECBs. It completes when either one of the sockets has activity, the time value expires, or the ECBs are posted.

To use the SELECTEX call as a timer in your program, take either of the following actions:

- Set the read, write, and exception arrays to zeros.
- Do not specify MAXSOC.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389 .
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.

Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Testing sockets

Read, write, and exception operations can be tested. The SELECTEX macro monitors activity on selected sockets to determine whether:

- A buffer for the specified sockets contains input data. If input data is available for a given socket, a read operation on that socket does not block.
- TCP/IP can accommodate additional output data. If TCP/IP can accept additional output for a socket, a write operation on the socket does not block.
- An exceptional condition occurs on a socket.
- A timeout occurs on the SELECTEX macro itself. A TIMEOUT period can be specified when the SELECTEX macro is issued.

Each socket descriptor is represented by a bit in a bit string.

Read operations

The ACCEPT, READ, READV, RECV, RECVFROM, and RECVMSG macros are read operations. A socket is ready for reading when data is received on it, or when an exception condition occurs.

To determine if a socket is ready for the read operation, set the appropriate bit in RSNDSK to 1 before issuing the SELECTEX macro. When the SELECTEX macro returns, the corresponding bits in the RRETMSK indicate sockets ready for reading.

Write operations

A socket is selected for writing, ready to be written, when:

- TCP/IP can accept additional outgoing data.
- A connection request is received in response to an ACCEPT macro.
- A CONNECT call for a nonblocking socket, which has previously returned ERRNO 36 (EINPROGRESS), completes the connection.

The WRITE, WRITEV, SEND, SENDMSG, or SENDTO macros block when the data to be sent exceeds the amount that TCP/IP can accept. To avoid this, you can precede the write operation with a SELECTEX macro to ensure that the socket is ready for writing. After a socket is selected for WRITE, your program can determine the amount of TCP/IP buffer space available by issuing the GETSOCKOPT macro with the SO_SNDBUF option.

To determine if a socket is ready for the write operation, set the appropriate bit in WSNDSK to 1.

Exception operations

For each socket to be tested, the SELECTEX macro can check for an exception condition. The exception conditions are:

- The calling program (concurrent server) has issued a GIVESOCKET command and the target subtask has successfully issued the TAKESOCKET call. When this condition is selected, the calling program (concurrent server) should issue CLOSE to dissociate itself from the socket.
- A socket has received out-of-band data. For this condition, a READ macro returns the out-of-band data before the program data.

To determine whether a socket has an exception condition, use the ESNDSK character string and set the appropriate bits to 1.

Returning the results

For each event tested by a `xSNDMSK`, a bit string records the results of the check. The bit strings are `RRETMSK`, `WRETMSK`, and `ERETMSK` for read, write, and exceptional events. On return from the `SELECTEX` macro, each bit set to 1 in the `xRETMSK` is a read, write, or exceptional event for the associated socket.

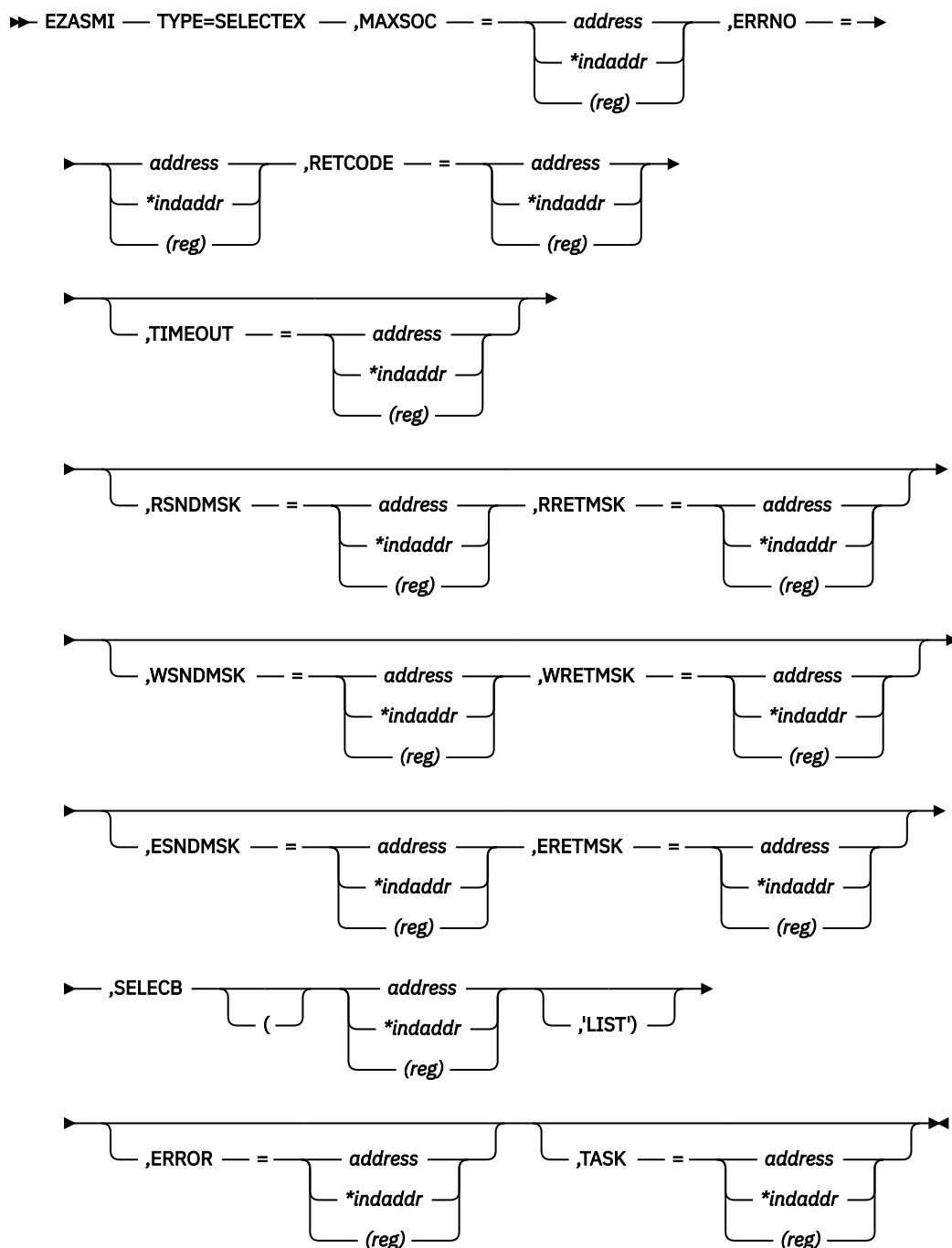
MAXSOC parameter

The `SELECTEX` call must test each bit in each string before returning results. For efficiency, the `MAXSOC` parameter can be used to specify the largest socket descriptor number that needs to be tested for any event type. The `SELECTEX` call tests only bits that are in the range 0 through the `MAXSOC` value minus 1.

Example: If `MAXSOC` value is set to 50, the range would be 0 – 49.

TIMEOUT parameter

If the time in the `TIMEOUT` parameter elapses before an event is detected, the `SELECTEX` macro returns and `RETCODE` is set to 0.



Keyword Description

MAXSOC

Input parameter. A fullword binary field that specifies the largest socket descriptor value being checked. The `SELECTEX` call tests only bits that are in the range 0 through the `MAXSOC` value minus 1. For example, if you set the `MAXSOC` value to 50, the range is 0 – 49.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, this contains an error number.

RETCODE

Output parameter. A fullword binary field.

Value

Meaning

>0

The number of ready sockets.

Note: If the number of ready sockets is greater than 65535, only 65535 is reported.

0

Either the SELECTEX time limit has expired (ECB value is 0) or one of the caller's ECBs has been posted (ECB value is nonzero and the caller's descriptor sets is set to 0). The caller must initialize the ECB values to zero before issuing the SELECTEX socket command.

-1

Check **ERRNO** for an error code.

TIMEOUT

Input parameter.

If **TIMEOUT** is not specified, the SELECTEX call blocks until a socket becomes ready or until a user **ECB** is posted.

If a **TIMEOUT** value is specified, **TIMEOUT** is the maximum interval for the SELECTEX call to wait until completion of the call. If you want SELECTEX to poll the sockets and return immediately, **TIMEOUT** should be specified to point to a zero-valued TIMEVAL structure.

TIMEOUT is specified in the two-word TIMEOUT as follows:

- TIMEOUT-SECONDS, word one of **TIMEOUT**, is the seconds component of the timeout value.
- TIMEOUT-MICROSEC, word two of **TIMEOUT**, is the microseconds component of the timeout value (0–999999).

For example, if you want SELECT to timeout after 3.5 seconds, set TIMEOUT-SECONDS to 3 and TIMEOUT-MICROSEC to 500000. TIMEOUT, SELECTEX returns to the calling program.

RSNDMSK

Input parameter. The bit-mask array to control checking for read interrupts. If this parameter is not specified or the specified bit-mask is zeros, the SELECT does not check for read interrupts. The length of this bit-mask array is dependent on the value in MAXSOC and must be a multiple of 4 bytes. See [“Selecting requests in a concurrent server program” on page 37](#) for more information.

RRETMSK

Output parameter. The bit-mask array returned by the SELECT if **RSNDMSK** is specified. The length of this bit-mask array is dependent on the value in MAXSOC and must be a multiple of 4 bytes. See [“Selecting requests in a concurrent server program” on page 37](#) for more information.

WSNDMSK

Input parameter. The bit-mask array to control checking for write interrupts. If this parameter is not specified or the specified bit-mask is zeros, the SELECT does not check for write interrupts. The length of this bit-mask array is dependent on the value in MAXSOC and must be a multiple of 4 bytes. See [“Selecting requests in a concurrent server program” on page 37](#) for more information.

WRETMSK

Output parameter. The bit-mask array returned by the SELECT if **WSNDMSK** is specified. The length of this bit-mask array is dependent on the value in MAXSOC and must be a multiple of 4 bytes. See [“Selecting requests in a concurrent server program” on page 37](#) for more information.

ESNDMSK

Input parameter. The bit-mask array to control checking for exception interrupts. If this parameter is not specified or the specified bit-mask is zeros, the SELECT does not check for exception interrupts. The length of this bit-mask array is dependent on the value in MAXSOC and must be a multiple of 4 bytes. See [“Selecting requests in a concurrent server program” on page 37](#) for more information.

ERETMSK

Output parameter. The bit-mask array returned by the SELECT if **ESNDMSK** is specified. The length of this bit-mask array is dependent on the value in MAXSOC and must be a multiple of 4 bytes. See [“Selecting requests in a concurrent server program” on page 37](#) for more information.

SELECB

Input parameter. An **ECB** or list of **ECB** addresses which, if posted, causes completion of the SELECTEX.

If the address of an **ECB** list is specified, you must set the high-order bit of the last entry in the **ECB** list to 1 and you must also add the LIST keyword. The **ECBs** must reside in the caller's home address space.

Note: The maximum number of **ECBs** that can be specified in a list is 1013.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

SEND

The SEND macro sends datagrams on a specified connected socket.

FLAGS allows you to:

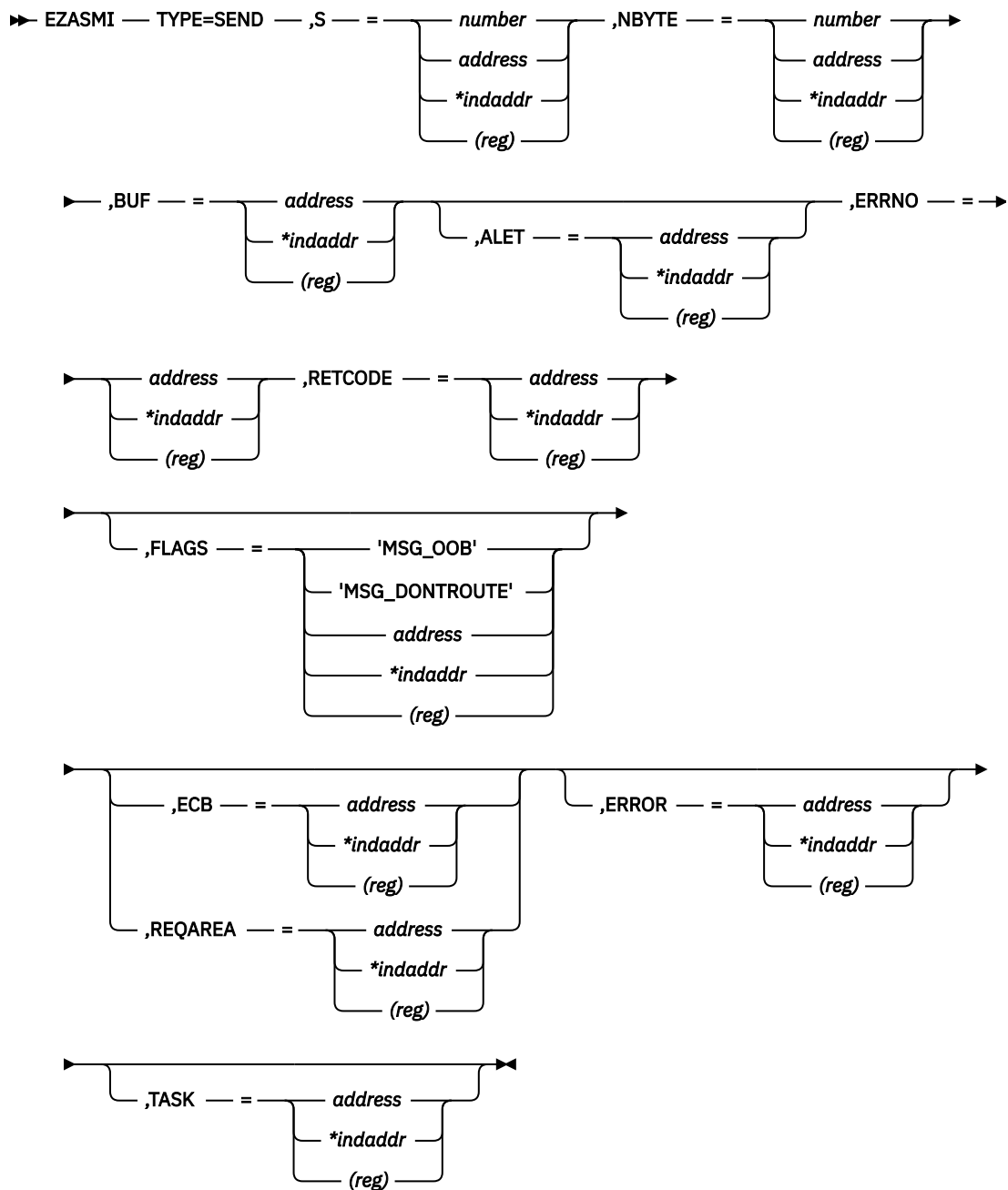
- Send out-of-band data, for example, interrupts, aborts, and data marked urgent. Only stream sockets created in the AF_INET address family support out-of-band data.
- Suppress use of local routing tables. This implies that the caller takes control of routing and writing network software.

For datagram sockets, SEND transmits the entire datagram if it fits into the receiving buffer. Extra data is discarded.

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 call to this function can send any number of bytes, up to the entire 1000 bytes, with the number of bytes sent returned in RETCODE. Therefore, programs using stream sockets should place this call in a loop and reissue the call until all data has been sent.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

S

Input parameter. A value or the address of a halfword binary number specifying the socket descriptor of the socket that is sending data.

NBYTE

Input parameter. A value or the address of a fullword binary number specifying the number of bytes to transmit.

BUF

The address of the data being transmitted. The length of BUF must be at least as long as the value of NBYTE.

ALET

Optional input parameter. A fullword binary field containing the ALET of BUF. The default is 0 (primary address space).

If a nonzero *ALET* is specified, the *ALET* must represent a valid entry in the dispatchable unit access list (DU-AL) for the task issuing this call. Note that *ALET*s can be specified only for synchronous socket calls (for example, ECB/REQAREA cannot be specified). An exception to this is an *ALET* representing a SCOPE=COMMON data space.

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, this field contains an error number. See Appendix B, “Socket call error return codes,” on page 743 for information about ERRNO return codes.

RETCODE

Output parameter. A fullword binary field.

Value

Description

0 or >0

A successful call. The value is set to the number of bytes transmitted.

-1

Check ERRNO for an error code.

FLAGS

Input parameter. FLAGS can be a literal value or a fullword binary field:

Literal Value	Binary Value	Description
'MSG_OOB'	X'00000001'	Send out-of-band data. (Stream sockets only.)
'MSG_DONTROUTE'	X'00000004'	Do not route. Routing is handled by the calling program.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte ECB posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the ECB has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

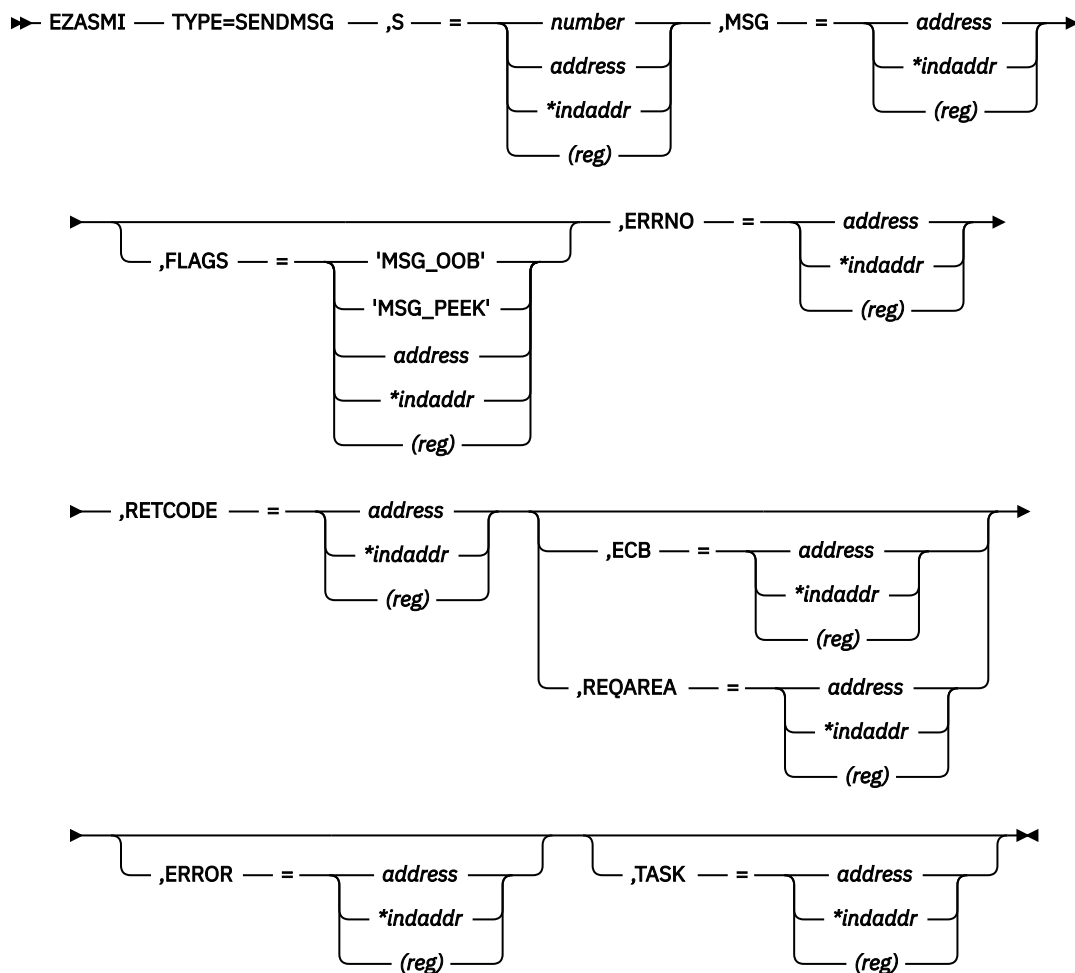
SENDMSG

The SENDMSG macro sends messages on a socket with descriptor *s* passed in an array of messages.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.

Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

S

Input parameter. A value or the address of a halfword binary number specifying the socket descriptor.

MSG

On input, this is a pointer to a message header into which the message is received on completion of the call.

NAME

On input, a pointer to a buffer where the sender's IPv4 or IPv6 address will be stored on completion of the call. The storage being pointed to should be for an IPv4 or IPv6 socket address. Include the SYS1.MACLIB(BPXYSOCK) macro to get the assembler mappings for the socket address structure. The socket address structure mappings begin at the SOCKADDR label. The AF_INET socket address structure fields start at the SOCK_SIN label. The AF_INET6 socket address structure fields start at the SOCK_SIN6 label.

The IPv4 socket address structure contains the following fields:

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

FAMILY

A halfword binary number specifying the IPv4 addressing family. The value for the IPv4 socket descriptor (S parameter) is a decimal 2, indicating AF_INET.

PORT

A halfword binary number specifying the port number of the sending socket.

IPv4-ADDRESS

A fullword binary number specifying the 32-bit IPv4 Internet address of the sending socket.

RESERVED

An 8-byte reserved field. This field is required, but is not used.

The IPv6 socket address structure contains the following fields:

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

NAMELEN

A 1-byte binary field specifying the length of this IPv6 socket address structure. Any value specified by the use of this field is ignored when processed as input and the field is set to 0 when processed as output.

FAMILY

A 1-byte binary field specifying the IPv6 addressing family. The value for the IPv6 socket descriptor (S parameter) is a decimal 19, indicating AF_INET6.

PORT

A halfword binary number specifying the port number of the sending socket.

FLOW-INFO

A fullword binary field specifying the traffic class and flow label. This field must be set to 0.

IPv6-ADDRESS

16-byte binary field specifying the 128-bit IPv6 Internet address, in network byte order, of the sending socket.

SCOPE-ID

A fullword binary field which identifies a set of interfaces as appropriate for the scope of the address carried in the IPv6-ADDRESS field. A value of 0 indicates the SCOPE-ID field does not identify the set of interfaces to be used, and can be specified for any address types and scopes. For a link scope IPv6-ADDRESS, SCOPE-ID can specify a link index which identifies a set of interfaces. For all other address scopes, SCOPE-ID can be set to 0.

IOV

A pointer to an array of three fullword structures with the number of structures equal to the value in IOVCNT and the format of the structures as follows:

Fullword 1

Input parameter. The address of a data buffer.

Fullword 2

Input parameter. The **ALET** for this buffer. If the buffer is in the primary address space, this should be zeros.

If a nonzero *ALET* is specified, the *ALET* must represent a valid entry in the dispatchable unit access list (DU-AL) for the task issuing this call. Note that *ALET*s can be specified only for synchronous socket calls (for example, ECB/REQAREA cannot be specified). An exception to this is an *ALET* representing a SCOPE=COMMON data space.

Fullword 3

Input parameter. The length of the data buffer referenced in Fullword 1.

IOVCNT

A pointer to a fullword binary field specifying the number of data buffers provided for this call.

ACCRIGHTS

A pointer to the access rights sent. This field is ignored.

ACCRLEN

A pointer to the length of the access rights sent. This field is ignored.

FLAGS

Input parameter. **FLAGS** can be a literal value or a fullword binary field:

Literal Value	Binary Value	Description
'MSG_OOB'	X'00000001'	Send out-of-band data. (Stream sockets only.)
'MSG_DONTROUTE'	X'00000004'	Do not route. Routing is handled by the calling program.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, this contains an error number.

RETCODE

Output parameter. A fullword binary field.

Value

Description

0 or >0

A successful call. The value is set to the number of bytes transmitted.

-1

Check **ERRNO** for an error code.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

SENDTO

SENDTO is similar to SEND, except that it includes the destination address parameter. You can use the destination address on the SENDTO macro to send datagrams on a UDP socket that is connected or not connected.

Use the **FLAGS** parameter to:

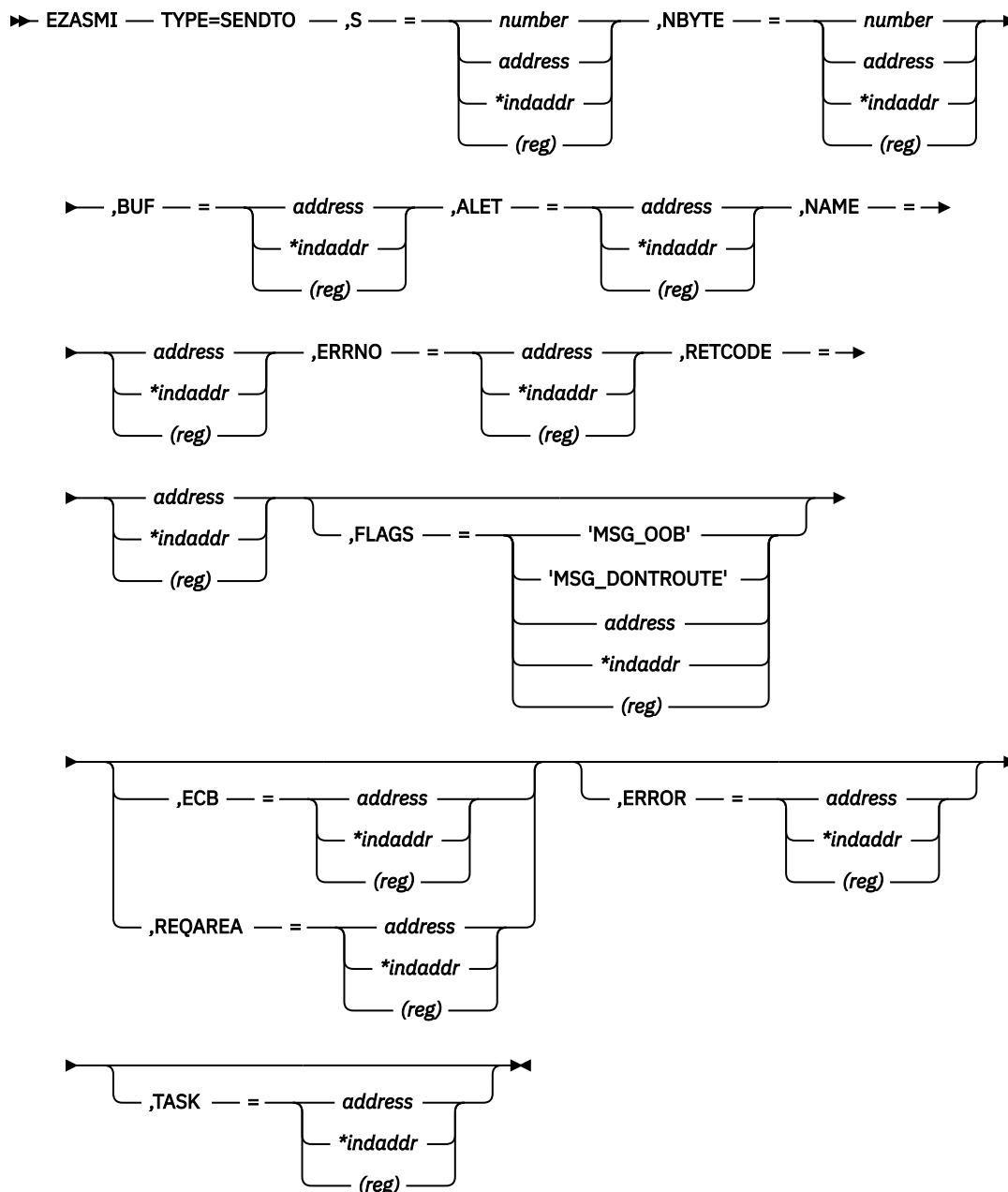
- Send out-of-band data, such as interrupts, aborts, and data marked as urgent.
- Suppress the local routing tables. This implies that the caller takes control of routing, which requires writing network software.

For datagram sockets, the SENDTO macro 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 macro call can send any number of bytes, up to the entire 1000 bytes, with the number of bytes sent returned in RETCODE. Therefore, programs using stream sockets should place SENDTO in a loop that repeats the macro until all data has been sent.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

S

Output parameter. A value or the address of a halfword binary number specifying the socket descriptor of the socket sending the data.

NBYTE

Input parameter. A value or the address of a fullword binary number specifying the number of bytes to transmit.

BUF

Input parameter. The address of the data being transmitted. The length of **BUF** must be at least as long as the value of **NBYTE**.

ALET

Optional input parameter. A fullword binary field containing the **ALET** of **BUF**. The default is 0 (primary address space).

If a nonzero *ALET* is specified, the *ALET* must represent a valid entry in the dispatchable unit access list (DU-AL) for the task issuing this call. Note that *ALET*s can be specified only for synchronous socket calls (for example, ECB/REQAREA cannot be specified). An exception to this is an *ALET* representing a SCOPE=COMMON data space.

NAME

Input parameter. The address of the IPv4 or IPv6 target. Include the SYS1.MACLIB(BPXYSOCK) macro to get the assembler mappings for the socket address structure. The socket address structure mappings begin at the SOCKADDR label. The AF_INET socket address structure fields start at the SOCK_SIN label. The AF_INET6 socket address structure fields start at the SOCK_SIN6 label.

The IPv4 socket address structure must specify the following fields:

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

FAMILY

A halfword binary field containing the IPv4 addressing family. The value for the IPv4 socket descriptor (S parameter) is a decimal 2, indicating AF_INET.

PORT

A halfword binary field containing the port number bound to the socket.

IP-ADDRESS

A fullword binary field containing the 32-bit IPv4 Internet Protocol address of the socket.

RESERVED

Specifies an 8-byte reserved field. This field is required, but is not used.

The IPv6 socket structure must specify the following fields:

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

NAMELEN

A 1-byte binary field specifying the length of this IPv6 socket address structure. Any value specified by the use of this field is ignored when processed as input and the field is set to 0 when processed as output.

FAMILY

A 1-byte binary field specifying the IPv6 addressing family. The value for IPv6 socket descriptor (S parameter) is a decimal 19, indicating AF_INET6.

PORT

A halfword binary field containing the port number bound to the socket.

FLOW-INFO

A fullword binary field specifying the traffic class and flow label. This field must be set to 0.

IPv6-ADDRESS

A 16-byte binary field that is set to the 128-bit IPv6 Internet Protocol address, in network byte order, of the client host machine. If 0 is specified, the application accepts connections from any network address.

SCOPE-ID

A fullword binary field which identifies a set of interfaces as appropriate for the scope of the address carried in the IPv6-ADDRESS field. A value of 0 indicates the SCOPE-ID field does not identify the set of interfaces to be used, and can be specified for any address types and scopes. For a link scope IPv6-ADDRESS, SCOPE-ID can specify a link index which identifies a set of interfaces. For all other address scopes, SCOPE-ID must be set to 0.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, **ERRNO** contains a valid error number. Otherwise, ignore **ERRNO**.

See [Appendix B, "Socket call error return codes,"](#) on page 743 for information about **ERRNO** return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value

Description

0 or >0

A successful call. The value is set to the number of bytes transmitted.

-1

Check **ERRNO** for an error code.

FLAGS

Input parameter. **FLAGS** can be a literal value or a fullword binary field:

Literal Value	Binary Value	Description
'MSG_OOB'	X'00000001'	Send out-of-band data. (Stream sockets only.)
'MSG_DONTROUTE'	X'00000004'	Do not route. Routing is handled by the calling program.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

SETSOCKOPT

The SETSOCKOPT macro sets the options associated with a socket.

The OPTVAL and OPTLEN parameters are used to pass data used by the particular set command. The OPTVAL parameter points to a buffer containing the data needed by the set command. The OPTLEN parameter must be set to the size of the data pointed to by OPTVAL.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.

Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

S

Input parameter. A value or the address of a halfword binary number specifying the socket descriptor of the socket requiring options.

OPTNAME

Input parameter. See the table below for a list of the options and their unique requirements. See [Appendix D, “GETSOCKOPT/SETSOCKOPT command values,” on page 767](#) for the numeric values of OPTNAME.

OPTVAL

Input parameter. Contains data about the option specified in OPTNAME. See the table below for a list of the options and their unique requirements

OPTLEN

Input parameter. A fullword binary field containing the length of the data returned in OPTVAL. See the table below for determining on what to base the value of OPTLEN.

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, this field contains an error number. See [Appendix B, “Socket call error return codes,” on page 743](#) for information about ERRNO return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check ERRNO for an error code.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte ECB posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the ECB has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

Table 17. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT*

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IP_ADD_MEMBERSHIP</p> <p>Use this option to enable an application to join a multicast group on a specific interface. An interface has to be specified with this option. Only applications that want to receive multicast datagrams need to join multicast groups.</p> <p>This is an IPv4-only socket option.</p>	<p>Contains the IP_MREQ structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 interface address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IP-MREQ.</p>	N/A
<p>IP_ADD_SOURCE_MEMBERSHIP</p> <p>Use this option to enable an application to join a source multicast group on a specific interface and a specific source address. You must specify an interface and a source address with this option. Applications that want to receive multicast datagrams need to join source multicast groups.</p> <p>This is an IPv4-only socket option.</p>	<p>Contains the IP_MREQ_SOURCE structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ_SOURCE structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 source address and a 4-byte IPv4 interface address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ_SOURCE.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IP-MREQ-SOURCE.</p>	N/A
<p>IP_BLOCK_SOURCE</p> <p>Use this option to enable an application to block multicast packets that have a source address that matches the given IPv4 source address. You must specify an interface and a source address with this option. The specified multicast group must have been joined previously.</p> <p>This is an IPv4-only socket option.</p>	<p>Contains the IP_MREQ_SOURCE structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ_SOURCE structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 source address and a 4-byte IPv4 interface address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ_SOURCE.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IP-MREQ-SOURCE.</p>	N/A

Table 17. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IP_DROP_MEMBERSHIP</p> <p>Use this option to enable an application to exit a multicast group or to exit all sources for a multicast group.</p> <p>This is an IPv4-only socket option.</p>	<p>Contains the IP_MREQ structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 interface address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IP-MREQ.</p>	<p>N/A</p>
<p>IP_DROP_SOURCE_MEMBERSHIP</p> <p>Use this option to enable an application to exit a source multicast group.</p> <p>This is an IPv4-only socket option.</p>	<p>Contains the IP_MREQ_SOURCE structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ_SOURCE structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 source address and a 4-byte IPv4 interface address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ_SOURCE.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IP-MREQ-SOURCE.</p>	<p>N/A</p>
<p>IP_MULTICAST_IF</p> <p>Use this option to set or obtain the IPv4 interface address used for sending outbound multicast datagrams from the socket application.</p> <p>This is an IPv4-only socket option.</p> <p>Note: Multicast datagrams can be transmitted only on one interface at a time.</p>	<p>A 4-byte binary field containing an IPv4 interface address.</p>	<p>A 4-byte binary field containing an IPv4 interface address.</p>
<p>IP_MULTICAST_LOOP</p> <p>Use this option to control or determine whether a copy of multicast datagrams is looped back for multicast datagrams sent to a group to which the sending host itself belongs. The default is to loop the datagrams back.</p> <p>This is an IPv4-only socket option.</p>	<p>A 1-byte binary field.</p> <p>To enable, set to 1.</p> <p>To disable, set to 0.</p>	<p>A 1-byte binary field.</p> <p>If enabled, will contain a 1.</p> <p>If disabled, will contain a 0.</p>

Table 17. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IP_MULTICAST_TTL</p> <p>Use this option to set or obtain the IP time-to-live of outgoing multicast datagrams. The default value is '01'x meaning that multicast is available only to the local subnet.</p> <p>This is an IPv4-only socket option.</p>	<p>A 1-byte binary field containing the value of '00'x to 'FF'x.</p>	<p>A 1-byte binary field containing the value of '00'x to 'FF'x.</p>
<p>IP_UNBLOCK_SOURCE</p> <p>Use this option to enable an application to unblock a previously blocked source for a given IPv4 multicast group. You must specify an interface and a source address with this option.</p> <p>This is an IPv4-only socket option.</p>	<p>Contains the IP_MREQ_SOURCE structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ_SOURCE structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 source address and a 4-byte IPv4 interface address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ_SOURCE.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IP-MREQ-SOURCE.</p>	

Table 17. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IPV6_ADDR_PREFERENCES</p> <p>Use this option to query or set IPv6 address preferences of a socket. The default source address selection algorithm considers these preferences when it selects an IP address that is appropriate to communicate with a given destination address.</p> <p>This is an AF_INET6-only socket option.</p> <p>Result: These flags are only preferences. The stack could assign a source IP address that does not conform to the IPV6_ADDR_PREFERENCES flags that you specify.</p> <p>Guideline: Use the INET6_IS_SRCADDR function to test whether the source IP address matches one or more IPV6_ADDR_PREFERENCES flags.</p>	<p>Contains the 4-byte flags field IPV6_ADDR_PREFERENCES_FLAGS that is defined in SYS1.MACLIB(BPXYSOCK) with the following flags:</p> <p>IPV6_PREFER_SRC_HOME (X'00000001') Prefer home address</p> <p>IPV6_PREFER_SRC_COA (X'00000002') Prefer care-of address</p> <p>IPV6_PREFER_SRC_TMP (X'00000004') Prefer temporary address</p> <p>IPV6_PREFER_SRC_PUBLIC (X'00000008') Prefer public address</p> <p>IPV6_PREFER_SRC_CGA (X'00000010') Prefer cryptographically generated address</p> <p>IPV6_PREFER_SRC_NONCGA (X'00000020') Prefer non-cryptographically generated address</p> <p>Some of these flags are contradictory. Combining contradictory flags, such as IPV6_PREFER_SRC_CGA and IPV6_PREFER_SRC_NONCGA, results in error code EINVAL.</p> <p>See IPV6_ADDR_PREFERENCES and Mapping of GAI_HINTS/GAI_ADDRINFO EFLAGS in SEZAINST(CBLOCK) for the PL/I example of the OPTNAME and flag definitions.</p> <p>See IPV6_ADDR_PREFERENCES and AI_EFLAGS mappings in SEZAINST(EZACOBOL) for the COBOL example of the OPTNAME and flag definitions.</p>	<p>Contains the 4-byte flags field IPV6_ADDR_PREFERENCES_FLAGS that is defined in SYS1.MACLIB(BPXYSOCK) with the following flags:</p> <p>IPV6_PREFER_SRC_HOME (X'00000001') Prefer home address</p> <p>IPV6_PREFER_SRC_COA (X'00000002') Prefer care-of address</p> <p>IPV6_PREFER_SRC_TMP (X'00000004') Prefer temporary address</p> <p>IPV6_PREFER_SRC_PUBLIC (X'00000008') Prefer public address</p> <p>IPV6_PREFER_SRC_CGA (X'00000010') Prefer cryptographically generated address</p> <p>IPV6_PREFER_SRC_NONCGA (X'00000020') Prefer non-cryptographically generated address</p> <p>See IPV6_ADDR_PREFERENCES and Mapping of GAI_HINTS/GAI_ADDRINFO EFLAGS in SEZAINST(CBLOCK) for the PL/I example of the OPTNAME and flag definitions.</p> <p>See IPV6_ADDR_PREFERENCES and AI_EFLAGS mappings in SEZAINST(EZACOBOL) for the COBOL example of the OPTNAME and flag definitions.</p>

Table 17. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IPV6_JOIN_GROUP</p> <p>Use this option to control the reception of multicast packets and specify that the socket join a multicast group.</p> <p>This is an IPv6-only socket option.</p>	<p>Contains the IPV6_MREQ structure as defined in SYS1.MACLIB(BPXYSOCK). The IPV6_MREQ structure contains a 16-byte IPv6 multicast address followed by a 4-byte IPv6 interface index number.</p> <p>If the interface index number is 0, then the stack chooses the local interface.</p> <p>See the SEZAINST(CBLOCK) for the PL/I example of IPV6_MREQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IPV6-MREQ.</p>	<p>N/A</p>
<p>IPV6_LEAVE_GROUP</p> <p>Use this option to control the reception of multicast packets and specify that the socket leave a multicast group.</p> <p>This is an IPv6-only socket option.</p>	<p>Contains the IPV6_MREQ structure as defined in SYS1.MACLIB(BPXYSOCK). The IPV6_MREQ structure contains a 16-byte IPv6 multicast address followed by a 4-byte IPv6 interface index number.</p> <p>If the interface index number is 0, then the stack chooses the local interface.</p> <p>See the SEZAINST(CBLOCK) for the PL/I example of IPV6_MREQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IPV6-MREQ.</p>	<p>N/A</p>
<p>IPV6_MULTICAST_HOPS</p> <p>Use to set or obtain the hop limit used for outgoing multicast packets.</p> <p>This is an IPv6-only socket option.</p>	<p>Contains a 4-byte binary value specifying the multicast hops. If not specified, then the default is 1 hop.</p> <p>-1 indicates use stack default.</p> <p>0 – 255 is the valid hop limit range.</p> <p>Note: An application must be APF authorized to enable it to set the hop limit value above the system defined hop limit value. CICS applications cannot execute as APF authorized.</p>	<p>Contains a 4-byte binary value in the range 0 – 255 indicating the number of multicast hops.</p>

Table 17. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IPv6_MULTICAST_IF</p> <p>Use this option to set or obtain the index of the IPv6 interface used for sending outbound multicast datagrams from the socket application.</p> <p>This is an IPv6-only socket option.</p>	<p>Contains a 4-byte binary field containing an IPv6 interface index number.</p>	<p>Contains a 4-byte binary field containing an IPv6 interface index number.</p>
<p>IPv6_MULTICAST_LOOP</p> <p>Use this option to control or determine whether a multicast datagram is looped back on the outgoing interface by the IP layer for local delivery when datagrams are sent to a group to which the sending host itself belongs. The default is to loop multicast datagrams back.</p> <p>This is an IPv6-only socket option.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to 1.</p> <p>To disable, set to 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>
<p>IPv6_UNICAST_HOPS</p> <p>Use this option to set or obtain the hop limit used for outgoing unicast IPv6 packets.</p> <p>This is an IPv6-only socket option.</p>	<p>Contains a 4-byte binary value specifying the unicast hops. If not specified, then the default is 1 hop.</p> <p>-1 indicates use stack default.</p> <p>0 – 255 is the valid hop limit range.</p> <p>Note: APF authorized applications are permitted to set a hop limit that exceeds the system configured default. CICS applications cannot execute as APF authorized.</p>	<p>Contains a 4-byte binary value in the range 0 – 255 indicating the number of unicast hops.</p>
<p>IPv6_V6ONLY</p> <p>Use this option to set or determine whether the socket is restricted to send and receive only IPv6 packets. The default is to not restrict the sending and receiving of only IPv6 packets.</p> <p>This is an IPv6-only socket option.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to 1.</p> <p>To disable, set to 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>

Table 17. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>MCAST_BLOCK_SOURCE</p> <p>Use this option to enable an application to block multicast packets that have a source address that matches the given source address. You must specify an interface index and a source address with this option. The specified multicast group must have been joined previously.</p>	<p>Contains the GROUP_SOURCE_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_SOURCE_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address and a socket address structure of the source address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_SOURCE_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-SOURCE-REQ.</p>	N/A
<p>MCAST_JOIN_GROUP</p> <p>Use this option to enable an application to join a multicast group on a specific interface. You must specify an interface index. Applications that want to receive multicast datagrams must join multicast groups.</p>	<p>Contains the GROUP_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-REQ.</p>	N/A
<p>MCAST_JOIN_SOURCE_GROUP</p> <p>Use this option to enable an application to join a source multicast group on a specific interface and a source address. You must specify an interface index and the source address. Applications that want to receive multicast datagrams only from specific source addresses need to join source multicast groups.</p>	<p>Contains the GROUP_SOURCE_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_SOURCE_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address and a socket address structure of the source address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_SOURCE_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-SOURCE-REQ.</p>	N/A

Table 17. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>MCAST_LEAVE_GROUP</p> <p>Use this option to enable an application to exit a multicast group or exit all sources for a given multicast groups.</p>	<p>Contains the GROUP_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-REQ.</p>	<p>N/A</p>
<p>MCAST_LEAVE_SOURCE_GROUP</p> <p>Use this option to enable an application to exit a source multicast group.</p>	<p>Contains the GROUP_SOURCE_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_SOURCE_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address and a socket address structure of the source address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_SOURCE_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-SOURCE-REQ.</p>	<p>N/A</p>
<p>MCAST_UNBLOCK_SOURCE</p> <p>Use this option to enable an application to unblock a previously blocked source for a given multicast group. You must specify an interface index and a source address with this option.</p>	<p>Contains the GROUP_SOURCE_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_SOURCE_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address and a socket address structure of the source address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_SOURCE_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-SOURCE-REQ.</p>	<p>N/A</p>

Table 17. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_ASCII</p> <p>Use this option to set or determine the translation to ASCII data option. When SO_ASCII is set, data is translated to ASCII. When SO_ASCII is not set, data is not translated to or from ASCII.</p> <p>Note: This is a REXX-only socket option.</p>	<p>To enable, set to ON.</p> <p>To disable, set to OFF.</p> <p>Note: The <i>optvalue</i> is returned and is optionally followed by the name of the translation table that is used if translation is applied to the data.</p>	<p>If enabled, contains ON.</p> <p>If disabled, contains OFF.</p> <p>Note: The <i>optvalue</i> is returned and is optionally followed by the name of the translation table that is used if translation is applied to the data.</p>
<p>SO_BROADCAST</p> <p>Use this option to set or determine whether a program can send broadcast messages over the socket to destinations that can receive datagram messages. The default is disabled.</p> <p>Note: This option has no meaning for stream sockets.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to 1 or a positive value.</p> <p>To disable, set to 0.</p>	<p>A 4-byte field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>
<p>SO_DEBUG</p> <p>Use SO_DEBUG to set or determine the status of the debug option. The default is <i>disabled</i>. The debug option controls the recording of debug information.</p> <p>Notes:</p> <ol style="list-style-type: none"> 1. This is a REXX-only socket option. 2. This option has meaning only for stream sockets. 	<p>To enable, set to ON.</p> <p>To disable, set to OFF.</p>	<p>If enabled, contains ON.</p> <p>If disabled, contains OFF.</p>
<p>SO_EBCDIC</p> <p>Use this option to set or determine the translation to EBCDIC data option. When SO_EBCDIC is set, data is translated to EBCDIC. When SO_EBCDIC is not set, data is not translated to or from EBCDIC. This option is ignored by EBCDIC hosts.</p> <p>Note: This is a REXX-only socket option.</p>	<p>To enable, set to ON.</p> <p>To disable, set to OFF.</p> <p>Note: The <i>optvalue</i> is returned and is optionally followed by the name of the translation table that is used if translation is applied to the data.</p>	<p>If enabled, contains ON.</p> <p>If disabled, contains OFF.</p> <p>Note: The <i>optvalue</i> is returned and is optionally followed by the name of the translation table that is used if translation is applied to the data.</p>
<p>SO_ERROR</p> <p>Use this option to request pending errors on the socket or to check for asynchronous errors on connected datagram sockets or for other errors that are not explicitly returned by one of the socket calls. The error status is clear afterwards.</p>	<p>N/A</p>	<p>A 4-byte binary field containing the most recent ERRNO for the socket.</p>

Table 17. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_KEEPALIVE</p> <p>Use this option to set or determine whether the keep alive mechanism periodically sends a packet on an otherwise idle connection for a stream socket.</p> <p>The default is disabled.</p> <p>When activated, the keep alive mechanism periodically sends a packet on an otherwise idle connection. If the remote TCP does not respond to the packet or to retransmissions of the packet, the connection is terminated with the error ETIMEDOUT.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to 1 or a positive value.</p> <p>To disable, set to 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>
<p>SO_LINGER</p> <p>Use this option to control or determine how TCP/IP processes data that has not been transmitted when a CLOSE is issued for the socket. The default is disabled.</p> <p>Notes:</p> <ol style="list-style-type: none"> 1. This option has meaning only for stream sockets. 2. If you set a zero linger time, the connection cannot close in an orderly manner, but stops, resulting in a RESET segment being sent to the connection partner. Also, if the aborting socket is in nonblocking mode, the close call is treated as though no linger option had been set. <p>When SO_LINGER is set and CLOSE is called, the calling program is blocked until the data is successfully transmitted or the connection has timed out.</p> <p>When SO_LINGER is not set, the CLOSE returns without blocking the caller, and TCP/IP continues to attempt to send data for a specified time. This usually allows sufficient time to complete the data transfer.</p> <p>Use of the SO_LINGER option does not guarantee successful completion because TCP/IP only waits the amount of time specified in OPTVAL for SO_LINGER.</p>	<p>Contains an 8-byte field containing two 4-byte binary fields.</p> <p>Assembler coding:</p> <pre>ONOFF DS F LINGER DS F</pre> <p>COBOL coding:</p> <pre>ONOFF PIC 9(8) BINARY. LINGER PIC 9(8) BINARY.</pre> <p>Set ONOFF to a nonzero value to enable and set to 0 to disable this option. Set LINGER to the number of seconds that TCP/IP lingers after the CLOSE is issued.</p>	<p>Contains an 8-byte field containing two 4-byte binary fields.</p> <p>Assembler coding:</p> <pre>ONOFF DS F LINGER DS F</pre> <p>COBOL coding:</p> <pre>ONOFF PIC 9(8) BINARY. LINGER PIC 9(8) BINARY.</pre> <p>A nonzero value returned in ONOFF indicates enabled, a 0 indicates disabled. LINGER indicates the number of seconds that TCP/IP will try to send data after the CLOSE is issued.</p>

Table 17. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_OOBLINE</p> <p>Use this option to control or determine whether out-of-band data is received.</p> <p>Note: This option has meaning only for stream sockets.</p> <p>When this option is set, out-of-band data is placed in the normal data input queue as it is received and is available to a RECV or a RECVFROM even if the OOB flag is not set in the RECV or the RECVFROM.</p> <p>When this option is disabled, out-of-band data is placed in the priority data input queue as it is received and is available to a RECV or a RECVFROM only when the OOB flag is set in the RECV or the RECVFROM.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to 1 or a positive value.</p> <p>To disable, set to 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>
<p>SO_RCVBUF</p> <p>Use this option to control or determine the size of the data portion of the TCP/IP receive buffer.</p> <p>The size of the data portion of the receive buffer is protocol-specific, based on the following values prior to any SETSOCKOPT call:</p> <ul style="list-style-type: none"> • TCPRCVBufsize keyword on the TCPCONFIG statement in the PROFILE.TCPIP data set for a TCP Socket • UDPRCVBufsize keyword on the UDPCONFIG statement in the PROFILE.TCPIP data set for a UDP Socket • The default of 65535 for a raw socket 	<p>A 4-byte binary field.</p> <p>To enable, set to a positive value specifying the size of the data portion of the TCP/IP receive buffer.</p> <p>To disable, set to a 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a positive value indicating the size of the data portion of the TCP/IP receive buffer.</p> <p>If disabled, contains a 0.</p>

Table 17. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_RCVTIMEO</p> <p>Use this option to control or determine the maximum length of time that a receive-type function can wait before it completes.</p> <p>If a receive-type function has blocked for the maximum length of time that was specified without receiving data, control is returned with an errno set to EWOULDBLOCK. The default value for this option is 0, which indicates that a receive-type function does not time out.</p> <p>When the MSG_WAITALL flag (stream sockets only) is specified, the timeout takes precedence. The receive-type function can return the partial count. See the explanation of that operation's MSG_WAITALL flag parameter.</p> <p>The following receive-type functions are supported:</p> <ul style="list-style-type: none"> • READ • READV • RECV • RECVFROM • RECVMMSG 	<p>This option requires a TIMEVAL structure, which is defined in SYS1.MACLIB(BPXRLIM) macro. The TIMEVAL structure contains the number of seconds and microseconds specified as fullword binary numbers. The seconds can be a value in the range 0 - 2 678 400 (equal to 31 days), and the microseconds can be a value in the range 0 - 1 000 000 (equal to 1 second). Although TIMEVAL value can be specified using microsecond granularity, the internal TCP/IP timers that are used to implement this function have a granularity of approximately 100 milliseconds.</p>	<p>This option stores a TIMEVAL structure that is defined in the SYS1.MACLIB(BPXRLIM) macro. The TIMEVAL structure contains the number of seconds and microseconds, which are specified as fullword binary numbers. The number of seconds value that is returned is in the range 0 - 2 678 400 (equal to 31 days). The number of microseconds value that is returned is in the range 0 - 1 000 000.</p>

Table 17. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_REUSEADDR</p> <p>Use this option to control or determine whether local addresses are reused. The default is disabled. This alters the normal algorithm used with BIND. The normal BIND algorithm allows each Internet address and port combination to be bound only once. If the address and port have been already bound, then a subsequent BIND will fail and result error will be EADDRINUSE.</p> <p>When this option is enabled, the following situations are supported:</p> <ul style="list-style-type: none"> • A server can BIND the same port multiple times as long as every invocation uses a different local IP address and the wildcard address INADDR_ANY is used only one time per port. • A server with active client connections can be restarted and can bind to its port without having to close all of the client connections. • For datagram sockets, multicasting is supported so multiple bind() calls can be made to the same class D address and port number. • If you require multiple servers to BIND to the same port and listen on INADDR_ANY, see the SHAREPORT option on the PORT statement in TCPIP.PROFILE. 	<p>A 4-byte binary field.</p> <p>To enable, set to 1 or a positive value.</p> <p>To disable, set to 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>
<p>SO_SNDBUF</p> <p>Use this option to control or determine the size of the data portion of the TCP/IP send buffer. The size of the TCP/IP send buffer is protocol specific and is based on the following conditions:</p> <ul style="list-style-type: none"> • The TCPSENDBufsize keyword on the TCPCONFIG statement in the PROFILE.TCPIP data set for a TCP socket • The UDPSENDBufsize keyword on the UDPCONFIG statement in the PROFILE.TCPIP data set for a UDP socket • The default of 65535 for a raw socket 	<p>A 4-byte binary field.</p> <p>To enable, set to a positive value specifying the size of the data portion of the TCP/IP send buffer.</p> <p>To disable, set to a 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a positive value indicating the size of the data portion of the TCP/IP send buffer.</p> <p>If disabled, contains a 0.</p>

Table 17. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_SNDTIMEO</p> <p>Use this option to control or determine the maximum length of time that a send-type function can remain blocked before it completes.</p> <p>If a send-type function has blocked for this length of time, it returns with a partial count or, if no data is sent, with an errno set to EWOULDBLOCK. The default value for this is 0, which indicates that a send-type function does not time out.</p> <p>For a SETSOCKOPT, the following send-type functions are supported:</p> <ul style="list-style-type: none"> • SEND • SENDMSG • SENDTO • WRITE • WRITEV 	<p>This option requires a TIMEVAL structure, which is defined in the SYS1.MACLIB(BPXYRLIM) macro. The TIMEVAL structure contains the number of seconds and microseconds specified as fullword binary numbers. The seconds value is in the range 0 - 2 678 400 (equal to 31 days), and the microseconds value is in the range 0 - 1 000 000 (equal to 1 second). Although the TIMEVAL value can be specified using microsecond granularity, the internal TCP/IP timers that are used to implement this function have a granularity of approximately 100 milliseconds.</p>	<p>This option stores a TIMEVAL structure that is defined in SYS1.MACLIB(BPXYRLIM). The TIMEVAL structure contains the number of seconds and microseconds, which are specified as fullword binary numbers. The number of seconds value that is returned is in the range 0 - 2 678 400 (equal to 31 days). The microseconds value that is returned is in the range 0 - 1 000 000.</p>
<p>SO_TYPE</p> <p>Use this option to return the socket type.</p>	<p>N/A</p>	<p>A 4-byte binary field indicating the socket type:</p> <p>X'1' indicates SOCK_STREAM.</p> <p>X'2' indicates SOCK_DGRAM.</p> <p>X'3' indicates SOCK_RAW.</p>
<p>TCP_KEEPAIVE</p> <p>Use this option to set or determine whether a socket-specific timeout value (in seconds) is to be used in place of a configuration-specific value whenever keep alive timing is active for that socket.</p> <p>When activated, the socket-specified timer value remains in effect until respecified by SETSOCKOPT or until the socket is closed. For more information about the socket option parameters, see TCP_KeepAlive socket option inz/OS Communications Server: IP Programmer's Guide and Reference.</p> <p>Tip: The site administrator can enable the global keep-alive mechanism by specifying the INTERVAL parameter on the TCPCONFIG statement in the TCP/IP stack profile data set, TCPIP.PROFILE.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to a value in the range of 1 – 2 147 460.</p> <p>To disable, set to a value of 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains the specific timer value (in seconds) that is in effect for the given socket.</p> <p>If disabled, contains a 0 indicating keep alive timing is not active.</p>

Table 17. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>TCP_NODELAY</p> <p>Use this option to set or determine whether data sent over the socket is subject to the Nagle algorithm (RFC 896).</p> <p>Under most circumstances, TCP sends data when it is presented. When this option is enabled, TCP will wait to send small amounts of data until the acknowledgment for the previous data sent is received. When this option is disabled, TCP will send small amounts of data even before the acknowledgment for the previous data sent is received.</p> <p>Note: Use the following format to set TCP_NODELAY OPTNAME value for COBOL programs:</p> <pre> 01 TCP-NODELAY-VAL PIC 9(10) COMP VALUE 2147483649. 01 TCP-NODELAY-REDEF REDEFINES TCP-NODELAY-VAL. 05 FILLER PIC 9(6) BINARY. 05 TCP-NODELAY PIC 9(8) BINARY. </pre>	<p>A 4-byte binary field.</p> <p>To enable, set to a 0.</p> <p>To disable, set to a 1 or nonzero.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 0.</p> <p>If disabled, contains a 1.</p>

SHUTDOWN

One way to terminate a network connection is to issue a CLOSE macro that attempts to complete all outstanding data transmission requests prior to breaking the connection. The SHUTDOWN macro 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.

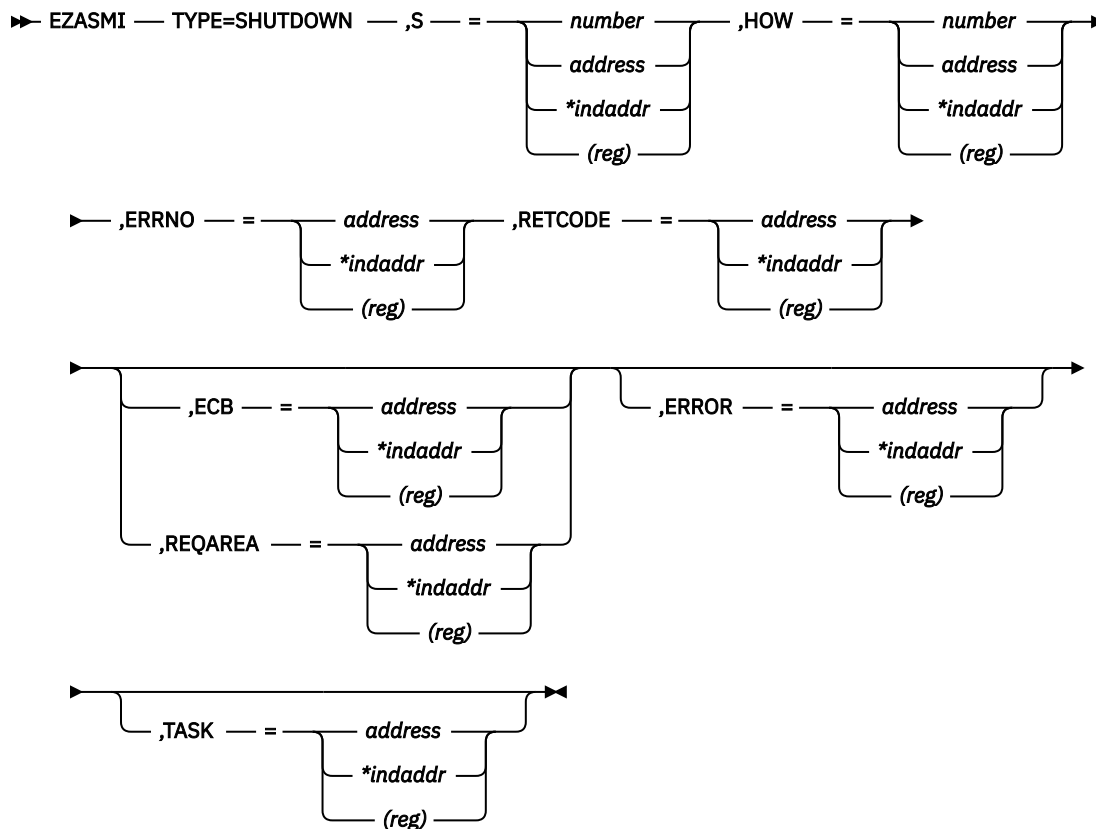
A client program can use the SHUTDOWN macro to reuse a given socket with a different connection.

If you issue SHUTDOWN for a socket that currently has outstanding socket calls pending, see [Table 3 on page 32](#) to determine the effects of this operation on the outstanding socket calls.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	<p>31-bit or 24-bit.</p> <p>Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.</p>
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.

Control parameters:	All parameters must be addressable by the caller and in the primary address space.
---------------------	--



Keyword Description

S

Input parameter. A value or the address of a halfword binary number specifying the socket to be shutdown.

HOW

Input parameter. A fullword binary field specifying the shutdown method.

Value

Description

0

Ends further receive operations.

1

Ends further send operations.

2

Ends further send and receive operations.

ERRNO

Output parameter. A fullword binary field. If `RETCODE` is negative, this field contains an error number. See Appendix B, “Socket call error return codes,” on page 743 for information about `ERRNO` return codes.

RETCODE

Output parameter. A fullword binary field that returns the following values:

Value

Description

0

Successful call.

-1

Check ERRNO for an error code.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte ECB posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the ECB has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

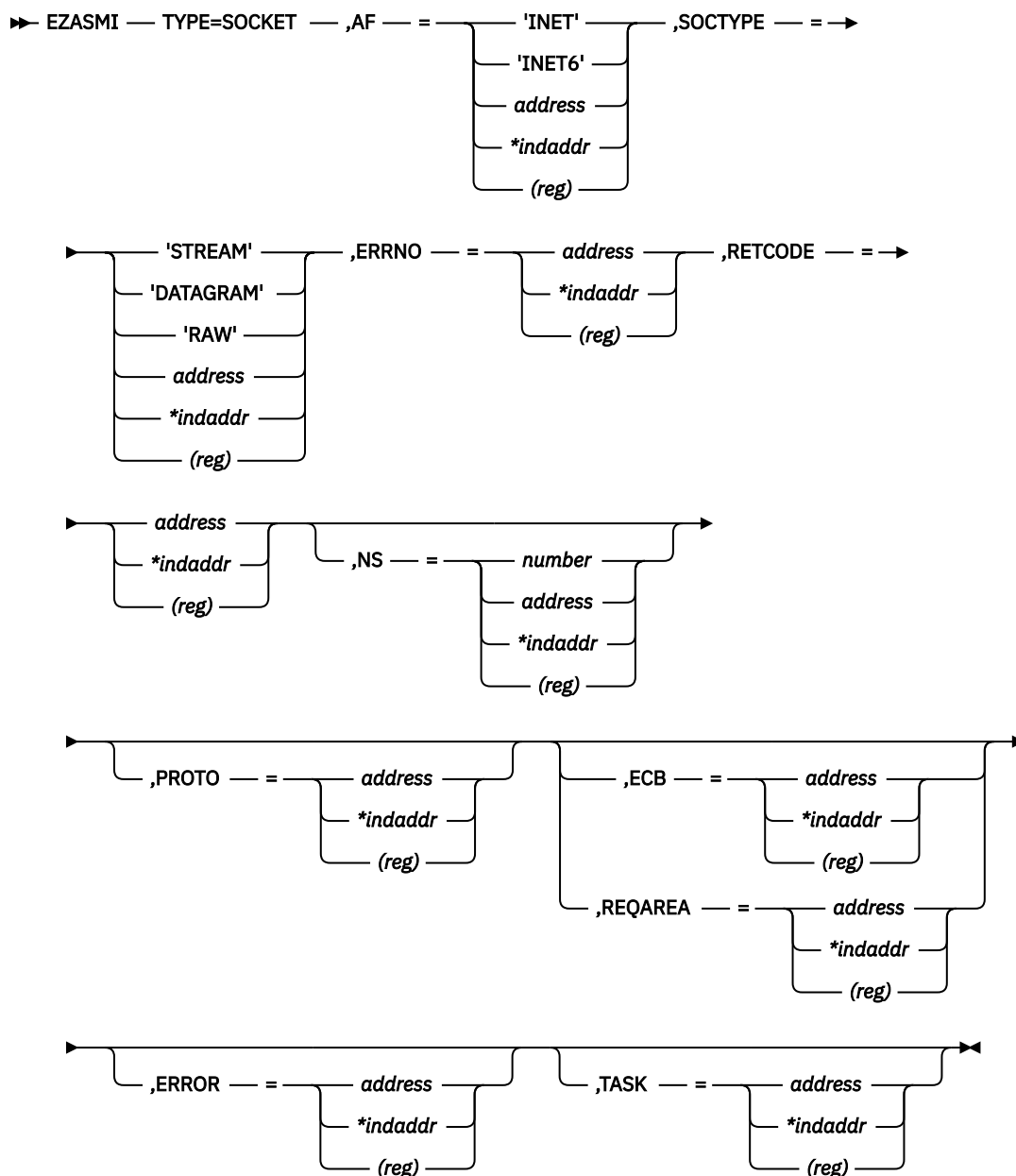
Input parameter. The location of the task storage area in your program.

SOCKET

The SOCKET macro creates an endpoint for communication and returns a socket descriptor representing the endpoint. Different types of sockets provide different communication services.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389 .
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

AF

Input parameter. Specify one of the following values:

Value

Description

'INET' or a decimal 2

Indicates the socket being created will use the IPv4 Internet protocol.

'INET6' or decimal 19

Indicates the socket being created will use the IPv6 Internet protocol.

Note: AF can also indicate a fullword binary number specifying the address family.

SOCTYPE

Input parameter. A fullword binary field set to the type of socket required. The types are:

Value	Description
-------	-------------

1 or 'STREAM'

Stream sockets provide sequenced, two-way byte streams that are reliable and connection-oriented. They support a mechanism for out-of-band data. This is the normal type for TCP/IP.

2 or 'DATAGRAM'

Datagram sockets provide datagrams, which are connectionless messages of a fixed maximum length whose reliability is not guaranteed. Datagrams can be corrupted, received out of order, lost, or delivered multiple times. This type is supported only in the AF_INET domain.

3 or 'RAW'

Raw sockets provide the interface to internal protocols (such as IP and ICMP).

Note: For SOCK_RAW sockets, the application must be APF-authorized.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, this field contains an error number. See Appendix B, “Socket call error return codes,” on page 743 for information about **ERRNO** return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value	Description
-------	-------------

> or = 0

Contains the new socket descriptor.

-1

Check **ERRNO** for an error code.

NS

Optional input. A value or the address of a halfword binary number specifying the socket number for the new socket. If a socket number is not specified, the interface assigns one.

PROTO

Input parameter. A fullword binary number specifying the protocol supported. PROTO only applies to new sockets and should be set to 0 for TCP/IP. PROTO for IPv6 raw sockets cannot be set to the following values:

Protocol name	Numeric value
---------------	---------------

IPROTO_HOPOPTS	0
-----------------------	---

IPPROTO_TCP	6
--------------------	---

IPPROTO_UDP	17
--------------------	----

IPPROTO_IPV6	41
---------------------	----

IPPROTO_ROUTING	43
------------------------	----

IPPROTO_FRAGMENT	44
-------------------------	----

IPPROTO_ESP	50
--------------------	----

IPPROTO_AH	51
-------------------	----

IPPROTO_NONE

59

IPPROTO_DSTOPTS

60

PROTO numbers are found in the *hlq.etc.proto* data set.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

TAKESOCKET

The TAKESOCKET macro acquires a socket from another program and creates a new socket. Typically, a subtask issues this macro using client ID and socket descriptor data that it obtained from the concurrent server.

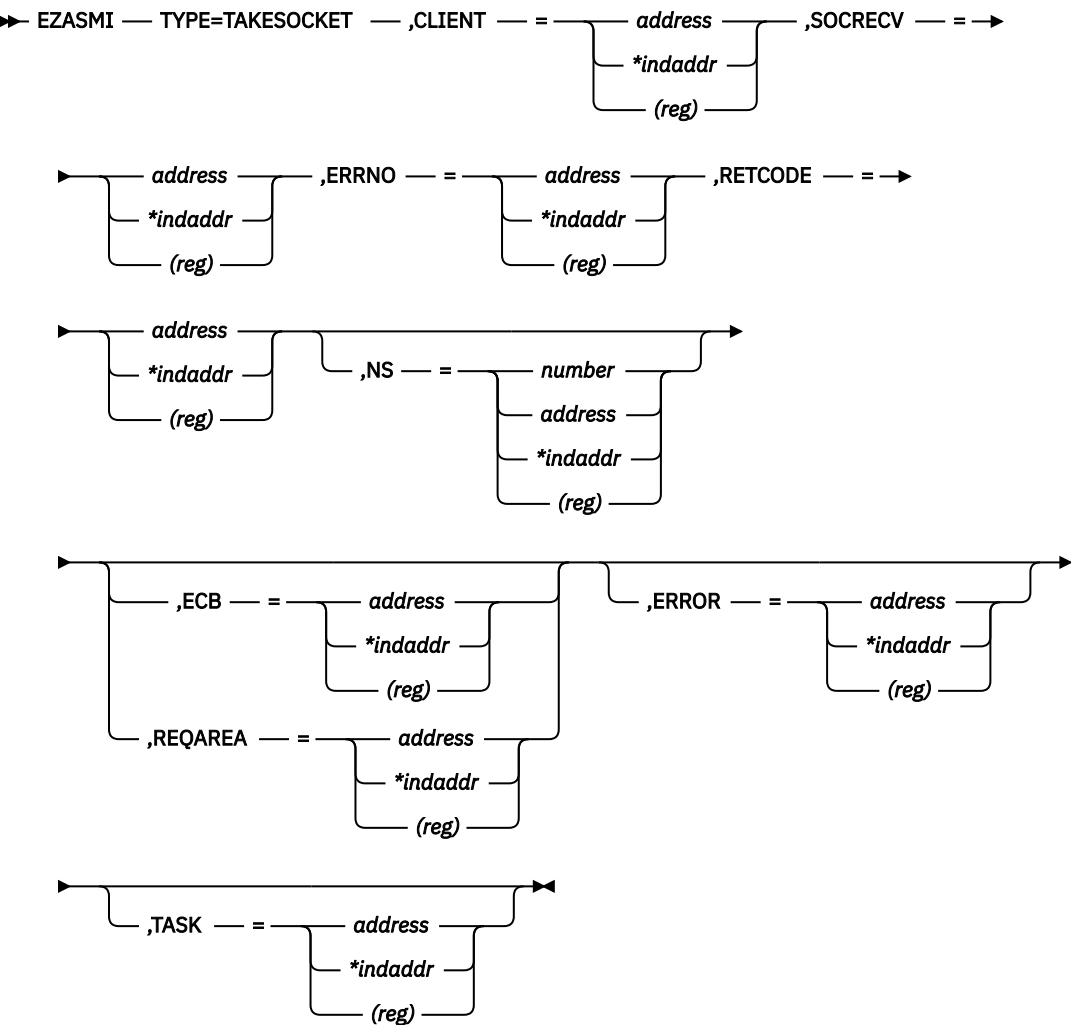
Notes:

1. When TAKESOCKET is issued, a new socket descriptor is returned in **RETCODE**. You should use this new socket descriptor in later macros such as GETSOCKOPT, which require the S (socket descriptor) parameter.
2. Both concurrent servers and iterative servers use this interface. An iterative server handles one client at a time. A concurrent server receives connection requests from multiple clients and creates subtasks that process the client requests. When a subtask is created, the concurrent server gets a new socket, passes the new socket to the subtask, and dissociates itself from the connection. The TCP/IP Listener program is an example of a concurrent server.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.

Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword

Description

CLIENT

Input parameter. The client data returned by the GETCLIENTID macro.

Field

Description

DOMAIN

Input parameter. A fullword binary number set to the domain of the program that is giving the socket. For TCP/IP the value is a decimal 2, indicating AF_INET, or a decimal 19, indicating AF_INET6.

Note: The TAKESOCKET can only acquire a socket of the same address family from a GIVESOCKET.

NAME

An 8-byte character field set to the MVS address space identifier of the program giving the socket.

TASK

Input parameter. Specifies an 8-byte field. This field must match the value of the SUBTASK parameter on the INITAPI for the MVS task that issued the GIVESOCKET request.

RESERVED

Input parameter. A 20-byte reserved field. This field is required, but not used.

SOCRECV

Input parameter. A halfword binary field containing the socket descriptor number assigned by the application that called GIVESOCKET.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, this field contains an error number. See [Appendix B, “Socket call error return codes,” on page 743](#) for information about **ERRNO** return codes.

RETCODE

Output parameter. A fullword binary field.

Value**Description****0 or >0**

Contains the new socket descriptor.

-1

Check **ERRNO** for an error code.

NS

Input parameter. A value or a halfword binary number specifying the socket descriptor number for the new socket. If a number is not specified, the interface assigns a socket number.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

TASK

The TASK macro allocates a task storage area addressable to all socket users communicating across a particular connection. Most commonly this is done by assigning one connection to each MVS subtask. If more than one module is using sockets within a connection or task, it is your responsibility to provide the task storage address to each user. Each program using sockets should define task storage using the instruction EZASMI TYPE=TASK with STORAGE=DSECT.

If this macro is not named, the default name EZASMTIE is assumed.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

►► EZASMI — TYPE=TASK — ,STORAGE — = — DSECT —►
CSECT

Keyword Description

STORAGE

Input parameter. Defines one of the following storage definitions:

Keyword Description

DSECT

Generates a DSECT.

CSECT

Generates an inline storage definition that can be used within a CSECT or as a part of a larger DSECT.

TERMAPI

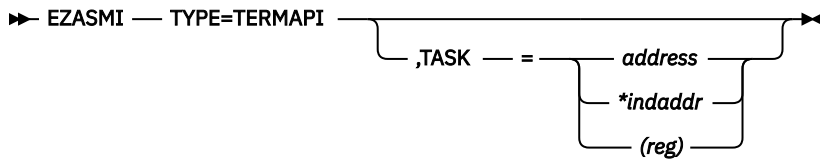
The TERMAPI macro ends the session created by the INITAPI macro.

Note: The INITAPI and TERMAPI macros must be issued under the same task.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.

Control parameters:	All parameters must be addressable by the caller and in the primary address space.
---------------------	--



Keyword Description

TASK

Input parameter. The location of the task storage area in your program.

WRITE

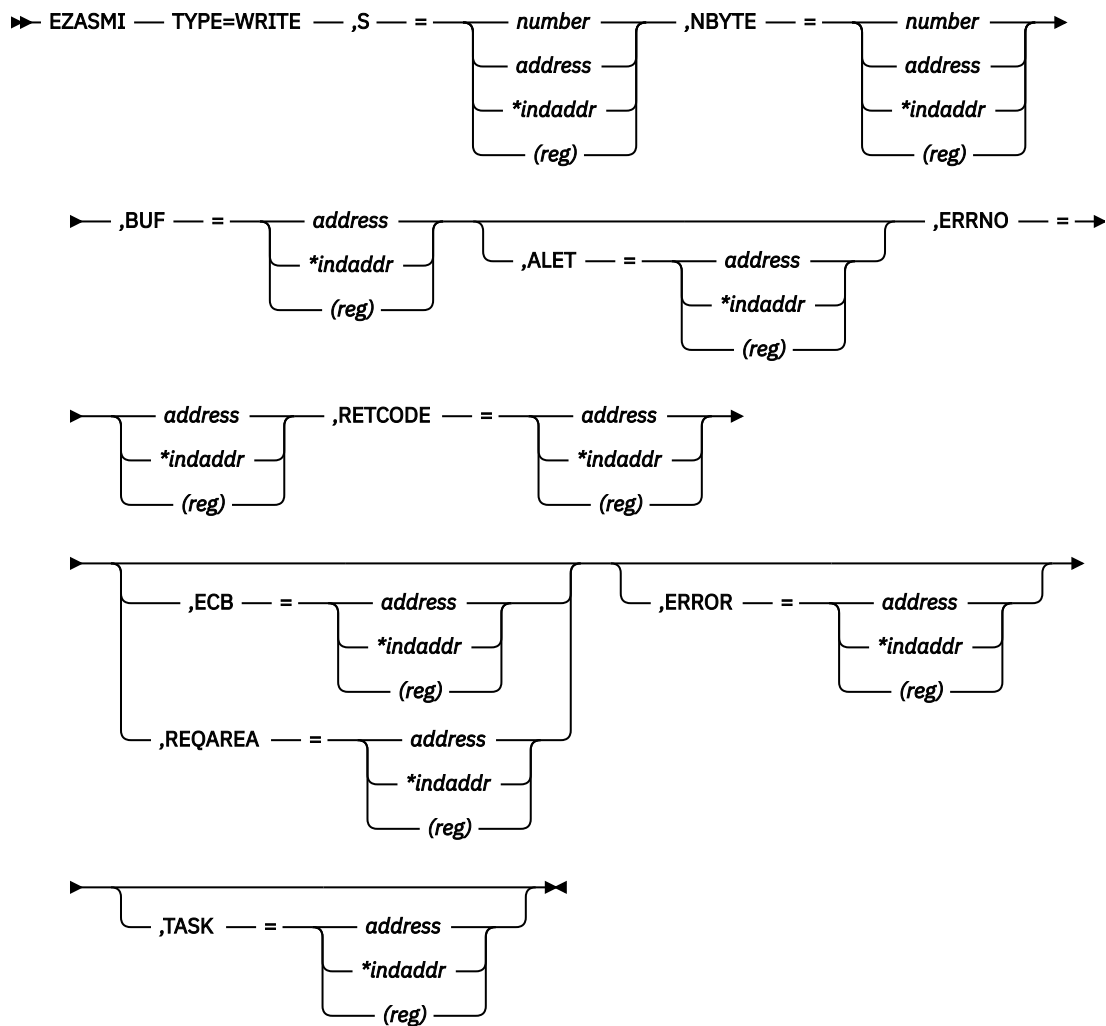
The WRITE macro writes data on a connected socket. The WRITE macro is similar to the SEND macro except that it does not have the control flags that can be used with SEND.

For datagram sockets, this macro writes the entire datagram, if it will fit into one TCP/IP buffer.

For stream sockets, the data is processed as streams of information with no boundaries separating the data. For example, if you want to send 1000 bytes of data, each call to the write macro can send 1 byte, 10 bytes, or the entire 1000 bytes. You should place the WRITE macro in a loop that cycles until all of the data has been sent.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.



Keyword Description

S

Input parameter. A value or the address of a halfword binary number specifying the socket descriptor of the socket to receive the data.

NBYTE

Input parameter. A value or the address of a fullword binary field specifying the number of bytes of data to transmit.

BUF

The address of the data being transmitted. The length of **BUF** must be at least as long as the value of **NBYTE**.

ALET

Optional input parameter. A fullword binary field containing the **ALET** of **BUF**. The default is 0 (primary address space).

If a nonzero **ALET** is specified, the **ALET** must represent a valid entry in the dispatchable unit access list (DU-AL) for the task issuing this call. Note that **ALETs** can be specified only for synchronous socket calls (for example, **ECB/REQAREA** cannot be specified). An exception to this is an **ALET** representing a **SCOPE=COMMON** data space.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, this field contains an error number. See Appendix B, “Socket call error return codes,” on page 743 for information about **ERRNO** return codes.

RETCODE

Output parameter. A fullword binary field.

Value

Description

>0

A successful call. The value is set to the number of bytes transmitted.

0

Connection partner has closed connection.

-1

Check **ERRNO** for an error code.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

This macro writes up to **NBYTE** bytes of data. If there is not enough available buffer space for the socket data to be transmitted, and the socket is in blocking mode, WRITE blocks the caller until additional buffer space is available. If the socket is in nonblocking mode, WRITE returns a -1 and sets **ERRNO** to 35 (EWOULDBLOCK). See [“FCNTL” on page 218](#) or [“IOCTL” on page 278](#) for a description of how to set the nonblocking mode.

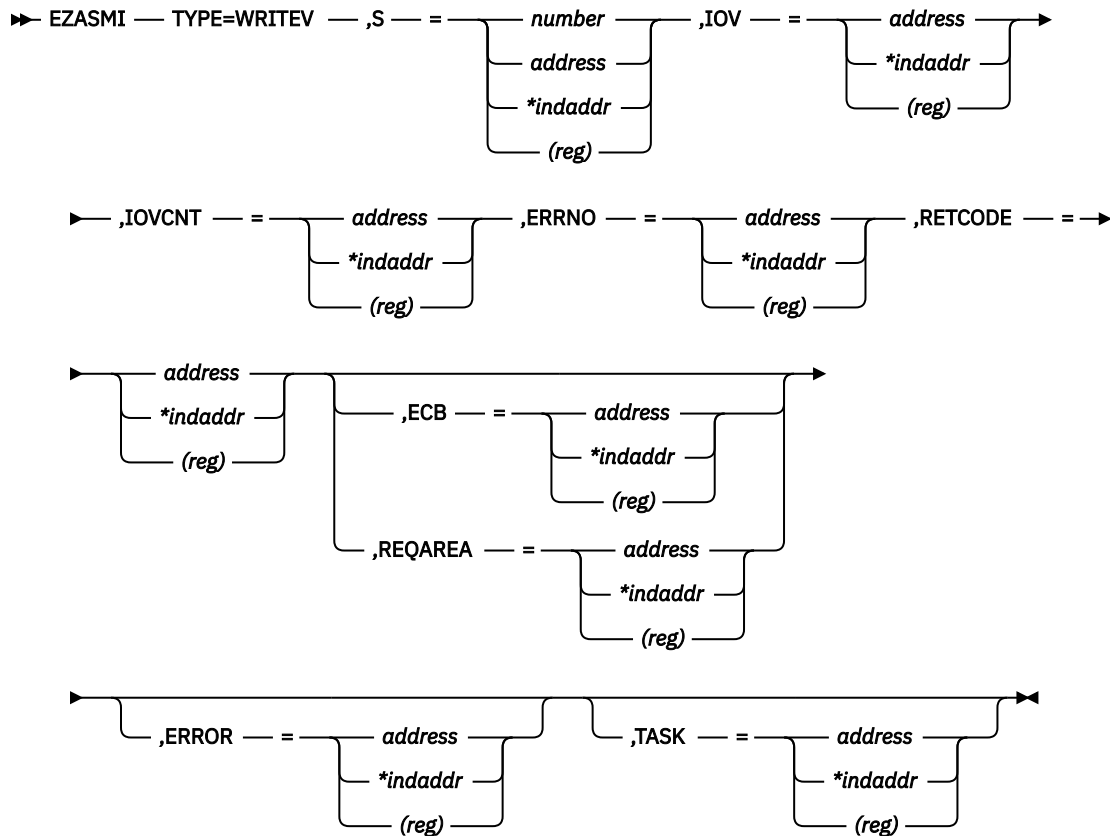
WRITEV

The WRITEV function writes data on a socket from a set of buffers.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389 .
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.

Control parameters:	All parameters must be addressable by the caller and in the primary address space.
---------------------	--



Keyword Description

S

Input parameter. A value or the address of a halfword binary number specifying the descriptor of the socket from which the data is to be written.

IOV

Input parameter. An array of three fullword structures with the number of structures equal to the value in IOVCNT and the format of the structures as follows:

Fullword 1

Input parameter. The address of a data buffer.

Fullword 2

Input parameter. The **ALET** for this buffer. If the buffer is in the primary address space, this should be zeros.

If a nonzero **ALET** is specified, the **ALET** must represent a valid entry in the dispatchable unit access list (DU-AL) for the task issuing this call. Note that **ALETs** can be specified only for synchronous socket calls (for example, ECB/REQAREA cannot be specified). An exception to this is an **ALET** representing a SCOPE=COMMON data space.

Fullword 3

Input parameter. The length of the data buffer referenced in Fullword 1.

IOVCNT

Input parameter. A fullword binary field specifying the number of data buffers provided for this call.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, this contains an error number.

RETCODE

Output parameter. A fullword binary field.

Value**Description****>0**

A successful call. The value is set to the number of bytes transmitted.

0

Connection partner has closed connection.

-1

Check **ERRNO** for an error code.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A 4-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA

A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

The last 100 bytes is a storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted, or the asynchronous exit has been driven.

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

TASK

Input parameter. The location of the task storage area in your program.

Macro interface assembler language sample programs

This information provides sample programs for the macro interface that you can use for assembler language applications. The source code can be found in the SEZAINST data set.

The following sample programs are included:

Program	Description
EZASOKAS	Sample IPv4 macro interface server program
EZASOKAC	Sample IPv4 macro interface client program
EZASO6AS	Sample IPv6 macro interface server program
EZASO6AC	Sample IPv6 macro interface client program

EZASOKAS sample server program for IPv4

The EZASOKAS program is a server program that shows you how to use the following calls provided by the macro socket interface:

- INITAPI
- SOCKET
- GETHOSTID
- BIND
- LISTEN

- ACCEPT
- READ
- WRITE
- CLOSE
- TERMAPI

```

EZASOKAS CSECT
EZASOKAS AMODE ANY
EZASOKAS RMODE ANY
*      PRINT NOGEN
*****
*
*   MODULE NAME:  EZASOKAS Sample server program
*
*   Copyright:    Licensed Materials - Property of IBM
*
*                 "Restricted Materials of IBM"
*
*                 5694-A01
*
*                 (C) Copyright IBM Corp. 1977, 2003
*
*                 US Government Users Restricted Rights -
*                 Use, duplication or disclosure restricted by
*                 GSA ADP Schedule Contract with IBM Corp.
*
*   Status:       CSV1R5
*
*
*   LANGUAGE:     Assembler
*
*   ATTRIBUTES:   NON-REUSABLE
*
*   REGISTER USAGE:
*       R1  =
*       R2  =
*       R3  = BASE REG 1
*       R4  = BASE REG 2 (UNUSED)
*       R5  = FUTURE BASE REG?
*       R6  = TEMP
*       R7  = RETURN REG
*       R8  =
*       R9  = A(WORK AREA)
*       R10 =
*       R11 =
*       R12 =
*       R13 = SAVE AREA
*       R14 =
*       R15 =
*
*   INPUT:  NONE
*   OUTPUT: WTO results of each test case
*
*****
      GBLB  &TRACE  ASSEMBLER VARIABLE TO CONTROL TRACE GENERATION
&TRACE  SETB  1      1=TRACE ON  0=TRACE OFF
R0       EQU  0
R1       EQU  1
R2       EQU  2
R3       EQU  3
R4       EQU  4
R5       EQU  5
R6       EQU  6
R7       EQU  7
R8       EQU  8
R9       EQU  9
R10      EQU 10
R11      EQU 11
R12      EQU 12
R13      EQU 13
R14      EQU 14
R15      EQU 15
*-----*
* START OF EXECUTABLE CODE
*-----*
      USING *,R3,R4          TELL ASSEMBLER OF OTHERS

```

```

        SAVE (14,12),T,*
        LR   R3,R15          COPY EP REG TO FIRST BASE
        LA   R5,2048         GET R5 HALFWAY THERE
        LA   R5,2048(R5)     GET R5 THERE
        LA   R4,0(R5,R3)     GET R4 THERE
        LA   R12,12          JUST FOR FUN!
        ST   R1,PARMADDR     SAVE ADDRESS OF PARAMETER LIST
        L    R1,0(R1)        GET POINTER
        LH   R1,0(R1)        GET LENGTH
*      STC   R1,TRACE        USE IT AS FLAG
        L    R7,=A(SOCSAVE)   GET NEW SAVE AREA
        ST   R7,8(R13)       SAVE ADDRESS OF NEW SAVE AREA
        ST   R13,4(R7)       COMPLETE SAVE AREA CHAIN
        LR   R13,R7          NOW SWAP THEM
        L    R9,=A(MYCB)     POINT TO THE CONTROL BLOCK
        USING MYCB,R9        TELL ASSEMBLER
*-----*
*      BUILD MESSAGE FOR CONSOLE
*-----*
*      INITIALIZE MESSAGE TEXT FIELDS
*
LOOP      EQU   *
        MVC   MSGNUM(8),SUBTASK WHO I AM
        MVC   TYPE,MSGSTART    MOVE 'STARTED' TO MESSAGE
*
        MVC   MSGRSLT1,MSGSUCC ...SUCCESSFUL TEXT
        MVC   MSGRSLT2,BLANK35
*
        STM   R14,R12,12(R13) JUST FOR DEBUGGING
        BAL   R14,WTO SUB     --> DO STARTING WTO
*****
*
*      Issue INITAPI to connect to interface
*
*****
        POST  ECB,1          NEXT IS ALWAYS SYNCH
        MVI   SYNFLAG,1      MOVE A 1 FOR ASYNCH
        MVC   TYPE,MINITAPI  MOVE 'INITAPI' TO MESSAGE
*
        EZASMI TYPE=INITAPI, Issue INITAPI Macro X
                SUBTASK=SUBTASK, SPECIFY SUBTASK IDENTIFIER X
                MAXSOC=MAXSOC, SPECIFY MAXIMUM NUMBER OF SOCKETS X
                MAXSNO=MAXSNO, (HIGHEST SOCKET NUMBER ASSIGNED) X
                ERRNO=ERRNO, (Specify ERRNO field) X
                RETCODE=RETCODE, (Specify RETCODE field) X
                APITYPE=APITYPE, (SPECIFY APITYPE FIELD) X
                ERROR=ERROR, ABEND IF ERROR ON MACRO X
                ASYNC=('EXIT',MYEXIT) (SPECIFY AN EXIT)
*      IDENT=IDENT, TCP ADDR SPACE AND MY ADDR SPACE
*      ASYNC=('ECB') (SPECIFY ECBS)
*
        BAL   R14,RCHECK     --> DID IT WORK?
*****
*
*      Issue SOCKET Macro to obtain a socket descriptor
*      *** INET and STREAM ***
*
*****
        MVC   TYPE,MSOCKET   MOVE 'SOCKET' TO MESSAGE
*
        EZASMI TYPE=SOCKET, Issue SOCKET Macro X
                AF='INET', INET or IUCV X
                SOCTYPE='STREAM', STREAM(TCP) DATAGRAM(UDP) or RAW X
                ERRNO=ERRNO, (Specify ERRNO field) X
                RETCODE=RETCODE, (Specify RETCODE field) X
                REQAREA=REQAREA, IN CASE WE ARE DOING EXITS OR ECBS X
                ERROR=ERROR Abend if Macro error
*
        BAL   R14,RCHECK     CHECK FOR SUCCESSFUL CALL
*
*-----*
*      Get socket descriptor number
*-----*
        STH   R8,S          SAVE RETCODE (=SOCKET DESCRIPTOR)
*****
*
*      ISSUE GETHOSTID CALL
*
*****
        MVC   TYPE,=CL8'GETHOSTI' 'GETHOSTI' TO MESSAGE
        EZASMI TYPE=GETHOSTID,RETCODE=RETCODE,ERRNO=ERRNO, X
                REQAREA=REQAREA IN CASE WE ARE DOING EXITS OR ECBS
        BAL   R14,RCHECK     CHECK FOR SUCCESSFUL CALL

```

```

      ST      R8,ADDR      SAVE OUR ID
*****
*
*      Issue BIND socket
*
*****
      MVC     TYPE,MBIND      MOVE 'BIND' TO MESSAGE
      MVC     PORT(2),PORTS    Load STREAM port #
      MVC     ADDRESS(4),ADDR   Load MVS1 internet address
*
      EZASMI  TYPE=BIND,      Issue Macro
              S=S,            STREAM
              NAME=NAME,      (SOCKET NAME STRUCTURE)
              ERRNO=ERRNO,    (Specify ERRNO field)
              RETCODE=RETCODE, (Specify RETCODE field)
              REQAREA=REQAREA, IN CASE WE ARE DOING EXITS OR ECBS
              ERROR=ERROR     Abend if Macro error
*
      BAL     R14,RCHECK      CHECK FOR SUCCESSFUL CALL
*****
*
*      Issue LISTEN - Backlog = 5
*
*****
      MVC     TYPE,MLISTEN     MOVE 'LISTEN' TO MESSAGE
*
      EZASMI  TYPE=LISTEN,     Issue Macro
              S=S,            STREAM
              BACKLOG=BACKLOG, BACKLOG
              ERRNO=ERRNO,    (Specify ERRNO field)
              RETCODE=RETCODE, (Specify RETCODE field)
              REQAREA=REQAREA, IN CASE WE ARE DOING EXITS OR ECBS
              ERROR=ERROR     Abend if Macro error
*
      BAL     R14,RCHECK      CHECK FOR SUCCESSFUL CALL
*****
*
*      Issue ACCEPT - Block until connection from peer
*
*****
      MVC     TYPE,MACCEPT    MOVE 'ACCEPT' TO MESSAGE
      MVC     PORT(2),PORTS    Load STREAM port #
      MVC     ADDRESS(4),ADDR   Load MVS1 internet address
*
      EZASMI  TYPE=ACCEPT,     Issue Macro
              S=S,            STREAM
              NAME=NAME,      (SOCKET NAME STRUCTURE)
              ERRNO=ERRNO,    (Specify ERRNO field)
              RETCODE=RETCODE, (Specify RETCODE field)
              REQAREA=REQAREA, IN CASE WE ARE DOING EXITS OR ECBS
              ERROR=ERROR     Abend if Macro error
*
      BAL     R14,RCHECK      CHECK FOR SUCCESSFUL CALL
* Message RESULTS text
      STH     R8,SOCDESCA      SAVE RETCODE (SOCKET DESCRIPTOR)
*****
*
*      Issue READ - Read data and store in buffer
*
*****
      MVC     TYPE,MREAD       MOVE 'READ ' TO MESSAGE
*
      EZASMI  TYPE=READ,       Issue Macro
              S=SOCDSCA,      ACCEPT SOCKET
              NBYTE=NBYTE,    SIZE OF BUFFER
              BUF=BUF,        (BUFFER)
              ERRNO=ERRNO,    (Specify ERRNO field)
              RETCODE=RETCODE, (Specify RETCODE field)
              REQAREA=REQAREA, IN CASE WE ARE DOING EXITS OR ECBS
              ERROR=ERROR     Abend if Macro error
*
      BAL     R14,RCHECK      CHECK FOR SUCCESSFUL CALL
      MVC     MSGRSLT1,MSGBUFF
      MVC     MSGRSLT2,BUF
      BAL     R14,WTOSUB      --> PRINT IT
*
*
*****
*
*      Issue WRITE - Write data from buffer
*
*****

```

```

MVC TYPE,MWRITE MOVE 'WRITE ' TO MESSAGE
*
EZASMI TYPE=WRITE, Issue Macro X
S=SOCDESCA, ACCEPT Socket X
NBYTE=NBYTE, SIZE OF BUFFER X
BUF=BUF, (BUFFER) X
ERRNO=ERRNO, (Specify ERRNO field) X
RETCODE=RETCODE, (Specify RETCODE field) X
REQAREA=REQAREA, IN CASE WE ARE DOING EXITS OR ECBS X
ERROR=ERROR Abend if Macro error
*
BAL R14,RCHECK CHECK FOR SUCCESSFUL CALL
*****
* Issue CLOSE for ACCEPT socket *
* *****
MVC TYPE,MCLOSE MOVE 'CLOSE' TO MESSAGE
*
EZASMI TYPE=CLOSE, Issue Macro X
S=SOCDESCA, ACCEPT X
ERRNO=ERRNO, (Specify ERRNO field) X
RETCODE=RETCODE, (Specify RETCODE field) X
REQAREA=REQAREA, IN CASE WE ARE DOING EXITS OR ECBS X
ERROR=ERROR Abend if Macro error
*
MVC MSGRSLT2,BLANK35
BAL R14,RCHECK CHECK FOR SUCCESSFUL CALL
*
*****
* Terminate Connection to API *
* *****
MVC TYPE,MTERMAPI MOVE 'TERMAPI' TO MESSAGE
*
POST ECB,1 FOLLOWING IS ALWAYS SYNCH
EZASMI TYPE=TERMAPI Issue EZASMI Macro for Termapl
*-----*
* Message RESULTS text
MVC MSGRSLT2,BLANK35
*
BAL R14,RCHECK --> CHECK RC
*-----*
* Issue console message for task termination
*-----*
MVC TYPE,MSGEND Move 'ENDED' to message
*
MVC MSGRSLT1,MSGSUCC ...SUCCESSFUL text
MVC MSGRSLT2,BLANK35
*
BAL R14,WTOSUB
LA R14,1 CONSTANT
AH R14,APITYPE ADD
STH R14,APITYPE STORE
CH R14,=H'3' COMPARE
BE LOOP --> LETS DO IT AGAIN!
*-----*
* Return to Caller
*-----*
L R13,4(R13)
RETURN (14,12),T,RC=0
WTOSUB EQU *
LR R7,R14 COPY RETURN REG
MVC MSGCMD(8),TYPE
WTO TEXT=MSG WRITE MESSAGE TO OPERATOR
BR R7 --> RETURN TO CALLER
CNOP 2,4
*
USES R6,R7,R8 RETCODE RETURNED IN R8
RCHECK EQU *
LR R7,R14 COPY TO REAL RETURN REG
MVC MSGRSLT1,MSGSUCC ...SUCCESS TEXT
L R6,RETCODE
LTR R6,R6
BM NOWAIT
CLI SYNFLAG,0 PLAIN CASE?
BE NOWAIT --> SKIP IT
MVC KEY+14(8),SUBTASK
MVC KEY+23(8),TYPE
KEY WTO 'WAIT: XXXXXXXX XXXXXXXX'
WAIT ECB=ECB
NOWAIT EQU *

```

```

*      LA      R15,ECB
*      ST      R15,ECB
*      ST      R9,ECB          MAKE THIS THE TOKEN AGAIN
*      L       R6,RETCODE      CHECK FOR SUCCESSFUL CALL
*      CLC     TYPE,=CL8'GETHOSTI'
*      BE      HOSTIDRC        HANDLE PRINTING HOST ID
*      LTR     R8,R6           SAVE A COPY
*
*      BNL     CONT00
*      EQU     *
*      FAILMSG EQU     *
*      MVC     MSGRSLT1,MSGFAIL ...FAIL TEXT
*      EQU     *
*
*-----*
*      FORMAT THE RETCODE= -XXXXXXX ERRNO= XXXXXXXX MSG RESULTS
*      ***> R6 = RETCODE VALUE ON ENTRY
*-----*
*
*      MVC     MSGRTCT,MSGRETC ' RETCODE= '
*      MVI     MSGRTCS,C'+ '
*      LTR     R6,R6
*      BNM     NOTM            -->
*      MVI     MSGRTCS,C'- '   MOVE SIGN WHICH IS ALWAYS MINUS
*      EQU     *
*      MVC     MSGERRT,MSGERRN ' ERRNO= '
*
*      CVD     R6,DWORK        CONVERT IT TO DECIMAL
*      UNPK    MSGRTCV,DWORK+4(4) UNPACK IT
*      OI      MSGRTCV+6,X'F0'  CORRECT THE SIGN
*      EQU     *
*      L       R6,ERRNO        GET ERRNO VALUE
*      CVD     R6,DWORK        CONVERT IT TO DECIMAL
*      UNPK    MSGERRV,DWORK+4(4) UNPACK IT
*      OI      MSGERRV+6,X'F0'  CORRECT THE SIGN
*
*      MVC     MSGRSLT2(35),MSGRTCD
*
*      MVI     MSGRTHX,X'40'    CLEAR HEX INDICATOR
*      SR      R6,R6            CLEAR OUT...
*      ST      R6,RETCODE      RETCODE AND...
*      ST      R6,ERRNO        ERRNO
*
*
*
*      CLI     TRACE,0
*      BNE     NOTRACE
*      LR      R14,R7          GIVE HIM RETURN REG
*      B       WTOSUB          --> DO WTO
*      EQU     *
*      BR      R7              --> RETURN TO CALLER
*
*      HOSTIDRC EQU     *
*      C       R6,=F'-1'       VALID HOSTID MAY LOOK LIKE NEG. RC
*      BE      FAILMSG         ONLY -1 RC INDICATES FAILURE
*      LR      R8,R6           ...BAD RC, USE STANDARD MSG
*      MVC     MSGRSLT1,MSGSUCC ...NEXT CALL EXPECTS ADDR IN R8
*      UNPK    HEXRC(9),RETCODE(5) ...SUCCESS TEXT
*      TR      HEXRC(8),HEXTAB  PLUS ONE FOR FAKE SIGN
*      MVI     HEXRC+8,X'40'    ...CONVERT UNPK TO PRINTABLE HEX
*      MVI     MSGRTHX,C'X'     ...SPACE OUT FAKED SIGN BYTE
*      B       ERRNOFMT        ...INDICATE INFO IS HEX
*
*      SYNFLAG DC      H'0'     DEFAULT TO SYN
*      TRACE   DC      H'0'     DEFAULT TO TRACE
*      MYEXIT  DC      A(MYEXIT1,SUBTASK)
*      MYEXIT1 SAVE (14,12),T,*
*      LR      R2,R15
*      USING   MYEXIT1,R2
*      LM      R8,R9,0(R1)      GET TWO TOKENS
*      MVC     EXKEY+14(8),0(R8) TELL WHO
*      MVC     EXKEY+23(8),TYPE  TELL WHAT
*      EXKEY   WTO 'EXIT: XXXXXXXX XXXXXXXX'
*      POST    ECB,1
*      RETURN  (14,12),T,RC=0
*      DROP    R2
*
*-----*
*      ABEND PROGRAM AND GET DUMP
*-----*
*      ERROR   ABEND 1,DUMP
*-----*
*      CONSTANTS USED TO RUN PROGRAM
*-----*
*      EZASMGW EZASMI TYPE=GLOBAL,      Storage definition for GWA
*              STORAGE=CSECT

```

```

*-----*
* INITAPI macro parms *
*-----*
SUBTASK DC CL8'EZASOKAS' SUBTASK PARM VALUE
MAXSOC DC AL2(50) MAXSOC PARM VALUE
APITYPE DC H'2' OR A 3
MAXSNO DC F'0' (HIGHEST SOCKET DESCRIPTOR AVAILABLE)
IDENT DC 0CL16' '
DC CL8' ' NAME OF TCP TO WHICH CONNECTING
DC CL8'SOC401CB' MY ADDR SPACE NAME
*-----*
* SOCKET macro parms *
*-----*
S DC H'0' SOCKET DESCRIPTOR FOR STREAM
*-----*
* BIND MACRO PARMS *
*-----*
NAME CNOP 0,4
DC 0CL16' ' SOCKET NAME STRUCTURE
DC AL2(2) FAMILY
PORT DC H'0'
ADDRESS DC F'0'
DC XL8'00' RESERVED
ADDR DC AL1(14),AL1(0),AL1(0),AL1(0) Internet Address
PORTS DC H'11007'
*-----*
* LISTEN PARMS *
*-----*
BACKLOG DC F'5' BACKLOG
*-----*
* READ MACRO PARMS *
*-----*
NBYTE DC F'50' SIZE OF BUFFER
SOCDESCA DC H'0' SOCKET DESCRIPTOR FROM ACCEPT
BUF DC CL50' THIS SHOULD NEVER APPEAR!!! :-(
*-----*
* WTO FRAGMENTS *
*-----*
MINITAPI DC CL8'INITAPI'
MSOCKET DC CL8'SOCKET'
MBIND DC CL8'BIND'
MACCEPT DC CL8'ACCEPT'
MLISTEN DC CL8'LISTEN'
MREAD DC CL8'READ'
MWRITE DC CL8'WRITE'
MCLOSE DC CL8'CLOSE'
MTERMAPI DC CL8'TERMAPI'
MSGSTART DC CL8' STARTED'
MSGEND DC CL8' ENDED '
MSGBUFF DC CL10' BUFFER: ' ...
MSGSUCC DC CL10' SUCCESS ' Command results...
MSGFAIL DC CL10' FAIL: ( ' ...
MSGRETC DC CL10' RETCODE= ' ...
MSGERRN DC CL10' ERRNO= ' ...
BLANK35 DC CL35' '
*-----*
* ERROR NUMBER / RETURN CODE FIELDS *
*-----*
* MESSAGE AREA *
*-----*
MSG DC 0F'0' MESSAGE AREA
DC AL2(MSGE-MSGNUM) LENGTH OF MESSAGE
MSGNUM DC CL10'EZASOKAS:' 'EZASOKASXX:'
MSGCMD DC CL8' ' COMMAND ISSUED
MSGRSLT1 DC CL10' ' COMMAND RESULTS (SUCC, PASS, FAIL)
MSGRSLT2 DC CL35' ' RETURNED VALUES
MSGGE EQU * End of message
*-----*
* MESSAGE RESULTS AREAS (fill in and move to MSGRSLT2) *
*-----*
MSGRTCD DC 0CL35' ' GENERAL RETURNED VALUE
MSGRTCT DC CL9' RETCODE=' ' RETCODE=
MSGRTHX DC CL1' ' 'X' X FOR GETHOSTID
MSGRTCS DC CL1' ' '-' (NEGATIVE SIGN)
HEXRC EQU MSGRTCS HEX RC WILL START AT SIGN LOCATION
MSGRTCV DC CL7' ' RETURNED VALUE (RETCODE)
MSGERRT DC CL10' ERRNO=' ' ERRNO=
MSGERRV DC CL7' ' RETURNED VALUE (ERRNO)
*-----*
PARMADDR DC A(0) PARM ADDRESS SAVE AREA
DWORK DC D'0' WORK AREA

```



```

HEXTAB EQU *-240          TAB TO CONVERT TO PRINTABLE HEX
*                               FIRST 240 BYTES NOT REFERENCED
      DC CL16'0123456789ABCDEF'
      LTORG ,
*-----*
* REG/SAVEAREA *
*-----*
SOCSAVE DC 9D'0'          SAVE AREA
      CNOP 0,8
MYCB EQU *                MY CONTROL BLOCK
REQAREA EQU *
ECB DC A(ECB)            SELF POINTER
      DC CL100'WORK AREA'
MYTIE EZASMI TYPE=TASK,STORAGE=CSECT TIE
      DC CL8'TYPE'
ERRNO DC F'0'
RETCODE DC F'0'
MYNEXT DC A(MYCB)        NEXT IN CHAIN FOR MULTIPLES
      CNOP 0,8
MYLEN EQU *-MYCB
MYCB2 EQU *
      ORG **MYLEN
      CNOP 0,8
      DC CL8'&SYSDATE'
      DC CL8'&SYSTIME'
      END

```

Figure 67. EZASOKAS sample server program for IPv4

EZASOKAC sample client program for IPv4

The EZASOKAC program is a client module that shows you how to use the following calls provided by the macro socket interface:

- INITAPI
- SOCKET
- CONNECT
- GETPEERNAME
- WRITE
- SHUTDOWN
- WRITE
- READ
- TERMAPI

```

EZASOKAC CSECT
EZASOKAC AMODE ANY
EZASOKAC RMODE ANY
      PRINT NOGEN
*****
*
*   MODULE NAME:  EZASOKAC - THIS IS A VERY SIMPLE CLIENT
*
* Copyright:    Licensed Materials - Property of IBM
*
*               "Restricted Materials of IBM"
*
*               5694-A01
*
*               (C) Copyright IBM Corp. 1977, 2003
*
*               US Government Users Restricted Rights -
*               Use, duplication or disclosure restricted by
*               GSA ADP Schedule Contract with IBM Corp.
*
* Status:      CSV1R5
*
*
*   LANGUAGE:   ASSEMBLER
*

```

```

*   ATTRIBUTES: NON-REUSEABLE
*
*   REGISTER USAGE:
*       R1 =
*       R2 =
*       R3 = BASE REG 1
*       R4 = BASE REG 2 (UNUSED)
*       R5 = FUTURE BASE?
*       R6 = TEMP
*       R7 = RETURN REG
*       R8 =
*       R9 = A(WORK AREA)
*       R10 =
*       R11 =
*       R12 =
*       R13 = SAVE AREA
*       R14 =
*       R15 =
*
*   INPUT: ANY PARM TURNS TRACE OFF, NO PARM IS NOISY MODE
*   OUTPUT: WTO RESULTS OF EACH TEST CASE IF TRACING
*           RETURN CODE IS 0 WHETHER IT CONNECTS OR NOT!
*
*****
GBLB  &TRACE  ASSEMBLER VARIABLE TO CONTROL TRACE GENERATION
&TRACE SETB 1 1=TRACE ON 0=TRACE OFF
R0 EQU 0
R1 EQU 1
R2 EQU 2
R3 EQU 3
R4 EQU 4
R5 EQU 5
R6 EQU 6
R7 EQU 7
R8 EQU 8
R9 EQU 9
R10 EQU 10
R11 EQU 11
R12 EQU 12
R13 EQU 13
R14 EQU 14
R15 EQU 15
*-----*
* START OF EXECUTABLE CODE
*-----*
        USING *,R3,R4      TELL ASSEMBLER OF OTHERS
        SAVE (14,12),T,*
        LR R3,R15          COPY EP REG TO FIRST BASE
        LA R5,2048          GET R5 HALFWAY THERE
        LA R5,2048(R5)      GET R5 THERE
        LA R4,0(R5,R3)      GET R4 THERE
        LA R12,12           JUST FOR FUN!
        ST R1,PARMADDR      SAVE ADDRESS OF PARAMETER LIST
        L R1,0(R1)          GET POINTER
        LH R1,0(R1)         GET LENGTH
*       STC R1,TRACE        USE IT AS FLAG
        L R7,=A(SOCSAVE)    GET NEW SAVE AREA
        ST R7,8(R13)        SAVE ADDRESS OF NEW SAVE AREA
        ST R13,4(R7)        COMPLETE SAVE AREA CHAIN
        LR R13,R7           NOW SWAP THEM
        L R9,=A(MYCB)       POINT TO THE CONTROL BLOCK
        USING MYCB,R9       TELL ASSEMBLER
*-----*
* BUILD MESSAGE FOR CONSOLE
*-----*
*                               INITIALIZE MESSAGE TEXT FIELDS
LOOP    EQU *
        MVC MSGNUM(8),SUBTASK WHO I AM
        MVC TYPE,MSGSTART    MOVE 'STARTED' TO MESSAGE
*
        MVC MSGRSLT1,MSGSUCC ...SUCCESSFUL TEXT
        MVC MSGRSLT2,BLANK35
*
        STM R14,R12,12(R13)  JUST FOR DEBUGGING
        BAL R14,WTO SUB      --> DO STARTING WTO
*****
*                               Issue INITAPI to connect to interface
*
*****
        MVC TYPE,MINITAPI    MOVE 'INITAPI' TO MESSAGE
*

```

```

POST ECB,1          FOLLOWING IS SYNC ONLY
MVI SYNFLAG,0       MOVE A 1 FOR ASYNCH
EZASMI TYPE=INITAPI,      ISSUE INITAPI MACRO          X
    SUBTASK=SUBTASK,    SPECIFY SUBTASK IDENTIFIER      X
    MAXSOC=MAXSOC,      SPECIFY MAXIMUM NUMBER OF SOCKETS X
    MAXSNO=MAXSNO,      (HIGHEST SOCKET NUMBER ASSIGNED) X
    ERRNO=ERRNO,        (Specify ERRNO field)            X
    RETCODE=RETCODE,    (Specify RETCODE field)          X
    APITYPE=APITYPE,    (SPECIFY APITYPE FIELD)          X
    ERROR=ERROR        Abend if error on macro
*    IDENT=IDENT,      TCP ADDR SPACE AND MY ADDR SPACE
*
*    ASYNC=('ECB'),      (SPECIFY TO USE ECBS)
*    ASYNC=('EXIT',MYEXIT) (SPECIFY TO USE EXITS)
BAL R14,RCHECK      --> CHECK RESULTS
*****
*    Issue SOCKET Macro to obtain a socket descriptor      *
*    *** INET and STREAM ***                               *
*
*****
MVC TYPE,M_SOCKET    MOVE 'SOCKET' TO MESSAGE
*
EZASMI TYPE=SOCKET,    Issue SOCKET Macro          X
    AF='INET',        INET or IUCV                X
    SOCTYPE='STREAM',  STREAM(TCP) DATAGRAM(UDP) or RAW X
    ERRNO=ERRNO,      (Specify ERRNO field)        X
    RETCODE=RETCODE,  (Specify RETCODE field)      X
    REQAREA=REQAREA,  FOR EXITS (AND ECBS)         X
    ERROR=ERROR      Abend if Macro error
*
BAL R14,RCHECK      --> CHECK RESULTS
STH R8,S            SAVE RETCODE (=SOCKET DESCRIPTOR)
LTR R8,R8           CHECK IT
BM DOSHUTDO        --> WE ARE DONE!
*****
*    ISSUE GETHOSTID CALL                                  *
*
*****
MVC TYPE,=CL8'GETHOSTI'
POST ECB,1          FOLLOWING IS SYNC ONLY
EZASMI TYPE=GETHOSTID,RETCODE=RETCODE,ERRNO=ERRNO
BAL R14,RCHECK      --> CHECK RESULTS
ST R8,ADDR
*****
*    Issue CONNECT Socket                                  *
*
*****
MVC TYPE,MCONNECT    MOVE 'CONNECT' TO MESSAGE
MVC PORT(2),PORTS    Load STREAM port #
*
*
MVC ADDRESS(4),ADDR  LOAD OUR INTERNET ADDRESS
*
EZASMI TYPE=CONNECT,  Issue Macro          X
    S=S,            STREAM                X
    NAME=NAME,      SOCKET NAME STRUCTURE X
    ERRNO=ERRNO,    (Specify ERRNO field) X
    RETCODE=RETCODE, (Specify RETCODE field) X
    REQAREA=REQAREA, FOR EXITS (AND ECBS) X
    ERROR=ERROR    Abend if Macro error
*
BAL R14,RCHECK      --> CHECK RC
LTR R8,R8           RECHECK IT
BM DOSHUTDO        --> WE ARE DONE
*****
*    Issue GETPEERNAME                                    *
*
*****
MVC TYPE,MGETPEER    MOVE 'GTPEERN' TO MESSAGE
*
EZASMI TYPE=GETPEERNAME, Issue Macro          X
    S=S,            STREAM                X
    NAME=NAME,      (SOCKET NAME STRUCTURE) X
    ERRNO=ERRNO,    (Specify ERRNO field) X
    RETCODE=RETCODE, (Specify RETCODE field) X
    REQAREA=REQAREA, FOR EXITS (AND ECBS) X
    ERROR=ERROR    Abend if Macro error
*

```

```

        BAL    R14,RCHECK      --> CHECK RC
*****
*
*      Issue WRITE - Write data from buffer
*
*****
        MVC    TYPE,MWRITE      MOVE 'WRITE ' TO MESSAGE
*
        EZASMI TYPE=WRITE,      Issue Macro
                        S=S,      STREAM SOCKET
                        NBYTE=NBYTE, SIZE OF BUFFER
                        BUF=BUF,   BUFFER
                        ERRNO=ERRNO, (Specify ERRNO field)
                        RETCODE=RETCODE, (Specify RETCODE field)
                        REQAREA=REQAREA, FOR EXITS (AND ECBS)
                        ERROR=ERROR    Abend if Macro error
*
        BAL    R14,RCHECK      --> CHECK RC
*****
*
*      Issue SHUTDOWN - HOW = 1 (end communication TO socket)
*
*****
DOSHUTDO EQU *
        MVC    HOW(4),=F'1'
*
        BAL    R14,SHUTSUB      --> SHUTDOWN
*
        BAL    R14,RCHECK      --> CHECK RC
*****
*
*      Issue READ - Read data and store in buffer
*
*****
        MVC    TYPE,MREAD      MOVE 'READ ' TO MESSAGE
*
        EZASMI TYPE=READ,      Issue Macro
                        S=S,      STREAM SOCKET
                        NBYTE=NBYTE, SIZE OF BUFFER
                        BUF=BUF2, (BUFFER)
                        ERRNO=ERRNO, (Specify ERRNO field)
                        RETCODE=RETCODE, (Specify RETCODE field)
                        REQAREA=REQAREA, FOR EXITS (AND ECBS)
                        ERROR=ERROR    Abend if Macro error
*
        BAL    R14,RCHECK      --> CHECK RC
        MVC    MSGRSLT1,MSGBUFF TITLE
        MVC    MSGRSLT2,BUF2    MOVE THE DATA
        BAL    R14,WTOSUB      --> PRINT IT
*****
*
*      Issue SHUTDOWN - HOW = 0 (end communication FROM socket)
*
*****
        MVC    HOW(4),=F'0'
*
        BAL    R14,SHUTSUB      --> SHUTDOWN
*
        BAL    R14,RCHECK      --> CHECK RC
*****
*
*      Terminate Connection to API
*
*****
        MVC    TYPE,MTERMAPI    MOVE 'TERMAPI' TO MESSAGE
*
        POST   ECB,1            FOLLOWING IS SYNC ONLY
        EZASMI TYPE=TERMAPI    Issue EZASMI Macro for Termapi
*
        BAL    R14,RCHECK      --> CHECK RC
*-----*
*      Issue console message for task termination
*-----*
        MVC    TYPE,MSGEND      Move 'ENDED' to message
*
        MVC    MSGRSLT1,MSGSUCC ...SUCCESSFUL text
        MVC    MSGRSLT2,BLANK35
        BAL    R14,WTOSUB      --> DO WTO
        LA     R14,1            CONSTANT
        AH     R14,APITYPE      ADD
        STH    R14,APITYPE      STORE
        CH     R14,=H'3'        COMPARE

```

```

*      BE      LOOP      --> LETS DO IT AGAIN!
*
*-----*
*      Return to Caller
*-----*
      L      R13,4(R13)
      RETURN (14,12),T,RC=0
WTOSUB  EQU      *
      LR      R7,R14      SAVE RETURN REG
      MVC     MSGCMD,TYPE  COPY COMMAND
      WTO     TEXT=MSG
      BR      R7      --> RETURN
*
SHUTSUB EQU      *
      LR      R7,R14
      MVC     TYPE,MSHUTDOWN  MOVE 'SHUTDOWN' TO MESSAGE
*
      EZASMI TYPE=SHUTDOWN,  Issue Macro                                X
                     S=S,      STREAM                                X
                     HOW=HOW,   End communication in both directions X
                     ERRNO=ERRNO, (Specify ERRNO field)              X
                     RETCODE=RETCODE, (Specify RETCODE field)        X
                     REQAREA=REQAREA, FOR EXITS (AND ECBS)           X
                     ERROR=ERROR Abend if Macro error
*
      BR      R7      --> RETURN TO CALLER
*-----*
*      ABEND PROGRAM AND GET DUMP TO DEBUG!
ERROR   ABEND 1,DUMP
        CNOP 2,4
*
*      USES R6,R7,R8      RETCODE RETURNED IN R8
RCCHECK EQU      *
      LR      R7,R14      COPY TO REAL RETURN REG
      MVC     MSGRSLT1,MSGSUCC ...SUCCESS TEXT
      L      R6,RETCODE
      LTR     R6,R6
      BM      NOWAIT
      CLI     SYNFLAG,0      PLAIN CASE?
      BE      NOWAIT      --> SKIP IT
      MVC     KEY+14(8),SUBTASK
      MVC     KEY+23(8),TYPE
KEY      WTO     'WAIT: XXXXXXXX XXXXXXXX'
NOWAIT   WAIT    ECB=ECB
*
*      LA      R15,ECB
*      ST      R15,ECB
*      ST      R9,ECB      MAKE THIS THE TOKEN AGAIN
      L      R6,RETCODE      CHECK FOR SUCCESSFUL CALL
      CLC     TYPE,=CL8'GETHOSTI'
      BE      HOSTIDRC      HANDLE PRINTING HOST ID
      LTR     R8,R6      SAVE A COPY
*
      BNL     CONT00
FAILMSG  EQU      *
      MVC     MSGRSLT1,MSGFAIL ...FAIL TEXT
CONT00   EQU      *
*-----*
*      FORMAT THE RETCODE= -XXXXXXX ERRNO= XXXXXXXX MSG RESULTS
*      ***> R6 = RETCODE VALUE ON ENTRY
*-----*
      MVC     MSGRTCT,MSGRETC ' RETCODE= '
      MVI     MSGRTCS,C'+'
      LTR     R6,R6
      BNM     NOTM      -->
      MVI     MSGRTCS,C'-'  MOVE SIGN WHICH IS ALWAYS MINUS
NOTM     EQU      *
      MVC     MSGERRT,MSGERRN ' ERRNO= '
*
      CVD     R6,DWORK      CONVERT IT TO DECIMAL
      UNPK    MSGRTCV,DWORK+4(4) UNPACK IT
      OI      MSGRTCV+6,X'F0' CORRECT THE SIGN
*
ERRNOFMT EQU      *
      L      R6,ERRNO      GET ERRNO VALUE
      CVD     R6,DWORK      CONVERT IT TO DECIMAL
      UNPK    MSGERRV,DWORK+4(4) UNPACK IT
      OI      MSGERRV+6,X'F0' CORRECT THE SIGN
*
      MVC     MSGRSLT2(35),MSGRTCD
*
      MVI     MSGRTHX,X'40' CLEAR HEX INDICATOR

```

```

SR      R6,R6          CLEAR OUT...
ST      R6,RETCODE     RETCODE AND...
ST      R6,ERRNO       ERRNO

*
*
      CLI  TRACE,0
      BNE NOTRACE
      LR   R14,R7       GIVE HIM RETURN REG
      B    WTOSUB       --> DO WTO
NOTRACE EQU  *
      BR   R7           --> RETURN TO CALLER

*
HOSTIDRC EQU  *          VALID HOSTID MAY LOOK LIKE NEG. RC
C         R6,=F'-1'     ONLY -1 RC INDICATES FAILURE
BE        FAILMSG       ...BAD RC, USE STANDARD MSG
LR        R8,R6         ...NEXT CALL EXPECTS ADDR IN R8
MVC       MSGRSLT1,MSGSUCC ...SUCCESS TEXT
UNPK      HEXRC(9),RETCODE(5) PLUS ONE FOR FAKE SIGN
TR        HEXRC(8),HEXTAB ...CONVERT UNPK TO PRINTABLE HEX
MVI       HEXRC+8,X'40' ...SPACE OUT FAKED SIGN BYTE
MVI       MSGRTHX,C'X'  ...INDICATE INFO IS HEX
B         ERRNOFMT

*
SYNFLAG  DC   H'0'      DEFAULT TO SYN
TRACE    DC   H'0'      DEFAULT TO TRACE
MYEXIT   DC   A(MYEXIT1,SUBTASK)
MYEXIT1  SAVE (14,12),T,*
      LR   R2,R15
      USING MYEXIT1,R2
      LM   R8,R9,0(R1)   GET TWO TOKENS
      MVC  EXKEY+14(8),0(R8) TELL WHO
      MVC  EXKEY+23(8),TYPE TELL WHAT
EXKEY    WTO  'EXIT: XXXXXXXX XXXXXXXX'
      POST ECB,1
      RETURN (14,12),T,RC=0
      DROP  R2

*-----*
* ELEMENTS USED TO RUN PROGRAM *
*-----*
EZASMGW  EZASMI TYPE=GLOBAL, STORAGE DEFINITION FOR GWA X
          STORAGE=CSECT

*-----*
* INITAPI macro parms *
*-----*
SUBTASK  DC   CL8'EZASOKAC' SUBTASK PARM VALUE
IDENT    DC   0CL16' '
          DC   CL8'TCPV32'  DEFAULT TO FIRST ONE AVAILABLE
          DC   CL8'EZASOKAC' MY ADDR SPACE NAME OR JOBNAME
MAXSNO   DC   F'0'         (HIGHEST SOCKET DESCRIPTOR AVAILABLE)
MAXSOC   DC   AL2(50)      MAXSOC PARM VALUE
APITYPE  DC   H'2'         OR PUT A 3 HERE

*-----*
* SOCKET macro parms *
*-----*
S         DC   H'0'         SOCKET DESCRIPTOR FOR STREAM

*-----*
* CONNECT MACRO PARMS *
*-----*
          CNOP 0,4
NAME      DC   0CL16' '     SOCKET NAME STRUCTURE
          DC   AL2(2)       FAMILY
PORT      DC   H'0'
ADDRESS   DC   F'0'
          DC   XL8'0'       RESERVED
ADDR      DC   AL1(14),AL1(0),AL1(0),AL1(0) Internet Address
PORTS     DC   H'11007'
*ORTS     DC   H'43'

*-----*
* WRITE MACRO PARMS *
*-----*
NBYTE    DC   F'50'        SIZE OF BUFFER
BUF       DC   CL50' THIS IS FROM EZASOKAC!' BUFFER FOR WRITE

*-----*
* SHUTDOWN MACRO PARMS *
*-----*
HOW       DC   F'2'        END COMMUNICATION TO- AND FROM-SOCKET

*-----*
* READ MACRO PARMS *
*-----*
BUF2      DC   CL50'BUF2'  BUFFER FOR READ

*-----*

```

```

MINITAPI DC      CL8'INITAPI'
MSOCKET  DC      CL8'SOCKET'
MCONNECT DC      CL8'CONNECT'
MGETPEER DC      CL8'GETPEERN'
MREAD    DC      CL8'READ'
MWRITE   DC      CL8'WRITE'
MSHUTDOWN DC     CL8'SHUTDOWN'
MTERMAPI DC      CL8'TERMAPI'
MSGSTART DC      CL8' STARTED'
MSGEND   DC      CL8' ENDED '
MSGSUCC  DC      CL10' SUCCESS '      Command results...
MSGFAIL  DC      CL10' FAIL: ( '      ...
MSGRETC  DC      CL10' RETCODE= '      ...
MSGERRN  DC      CL10' ERRNO= '      ...
MSGBUFF  DC      CL10' BUFFER: '      ...
BLANK35  DC      CL35' '

*-----*
* MESSAGE AREA *
*-----*
MSG      DC      0F'0'      MESSAGE AREA
          DC      AL2(MSGE-MSGNUM) LENGTH OF MESSAGE
MSGNUM   DC      CL10'EZASOKAC:' 'EZASOKAC: '
MSGCMD   DC      CL8' '      COMMAND ISSUED
MSGRSLT1 DC      CL10' '      COMMAND RESULTS (SUCC, PASS, FAIL)
MSGRSLT2 DC      CL35' '      RETURNED VALUES
MSGE     EQU      *          End of message
*-----*
* MESSAGE RESULTS AREAS (fill in and move to MSGRSLT2) *
*-----*
*
MSGRTCD  DC      0CL35' '      GENERAL RETURNED VALUE
MSGRTCT  DC      CL9' RETCODE=' ' RETCODE= '
MSGRTHX  DC      CL1' '      'X' X FOR GETHOSTID
MSGRTCS  DC      CL1' '      '-' (NEGATIVE SIGN)
HEXRC    EQU      MSGRTCS      HEX RC WILL START AT SIGN LOCATION
MSGRTCV  DC      CL7' '      RETURNED VALUE (RETCODE)
MSGERRT  DC      CL10' ERRNO=' ' ERRNO= '
MSGERRV  DC      CL7' '      RETURNED VALUE (ERRNO)
DWORK    DC      D'0'          WORK AREA
HEXTAB   EQU      *-240        TAB TO CONVERT TO PRINTABLE HEX
*          FIRST 240 BYTES NOT REFERENCED
*
PARMADDR DC      CL16'0123456789ABCDEF'
          DC      A(0)          PARM ADDRESS SAVE AREA
          LTORG
*-----*
* REG/SAVEAREA *
*-----*
SOCSAVE  DC      9D'0'          SAVE AREA
*-----*
MYCB      CNOP    0,8
          EQU      *          MY CONTROL BLOCK
REQAREA   EQU      *
ECB       DC      A(ECB)      SELF POINTER
          DC      CL100'WORK AREA'
MYTIE     EZASMI  TYPE=TASK,STORAGE=CSECT TIE
TYPE      DC      CL8'TYPE'
ERRNO     DC      F'0'
RETCODE   DC      F'0'
MYNEXT    DC      A(MYCB)      NEXT IN CHAIN FOR MULTIPLES
          CNOP    0,8
MYLEN     EQU      *-MYCB
MYCB2     EQU      *
          ORG      **MYLEN
          CNOP    0,8
          DC      CL8'&SYSDATE'
          DC      CL8'&SYSTIME'
          END

```

Figure 68. EZASOKAC sample client program for IPv4

EZASO6AS sample server program for IPv6

The EZASO6AS program is a server program that shows you how to use the following calls provided by the macro socket interface:

- ACCEPT
- BIND

- CLOSE
- GETADDRINFO
- GETHOSTNAME
- FREEADDRINFO
- INITAPI
- LISTEN
- PTON
- READ
- SOCKET
- TERMAPI
- WRITE

```

EZAS06AS CSECT
EZAS06AS AMODE ANY
EZAS06AS RMODE ANY
*      PRINT NOGEN
*****
*
*   MODULE NAME:  EZAS06AS Sample IPV6 server program
*
*   Copyright:    Licensed Materials - Property of IBM
*
*                 "Restricted Materials of IBM"
*
*                 5694-A01
*
*                 (C) Copyright IBM Corp. 2002, 2003
*
*                 US Government Users Restricted Rights -
*                 Use, duplication or disclosure restricted by
*                 GSA ADP Schedule Contract with IBM Corp.
*
*   Status:       CSV1R5
*
*
*   LANGUAGE:     Assembler
*
*   ATTRIBUTES:   NON-REUSABLE
*
*   REGISTER USAGE:
*       R1 =
*       R2 =
*       R3 = BASE REG 1
*       R4 = BASE REG 2 (UNUSED)
*       R5 = FUTURE BASE REG?
*       R6 = TEMP
*       R7 = RETURN REG
*       R8 =
*       R9 = A(WORK AREA)
*       R10 =
*       R11 =
*       R12 =
*       R13 = SAVE AREA
*       R14 =
*       R15 =
*
*   INPUT:  NONE
*   OUTPUT: WTO results of each test case
*
*****
          GBLB  &TRACE  ASSEMBLER VARIABLE TO CONTROL TRACE GENERATION
&TRACE  SETB  1      1=TRACE ON  0=TRACE OFF
R0       EQU   0
R1       EQU   1
R2       EQU   2
R3       EQU   3
R4       EQU   4
R5       EQU   5
R6       EQU   6
R7       EQU   7
R8       EQU   8

```



```

R9      EQU      9
R10     EQU      10
R11     EQU      11
R12     EQU      12
R13     EQU      13
R14     EQU      14
R15     EQU      15
*-----*
*  START OF EXECUTABLE CODE
*-----*
        USING *,R3,R4          TELL ASSEMBLER OF OTHERS
        SAVE (14,12),T,*
        LR      R3,R15          COPY EP REG TO FIRST BASE
        LA      R5,2048         GET R5 HALFWAY THERE
        LA      R5,2048(R5)     GET R5 THERE
        LA      R4,0(R5,R3)     GET R4 THERE
        LA      R12,12          JUST FOR FUN!
        ST      R1,PARMADDR     SAVE ADDRESS OF PARAMETER LIST
        L       R1,0(R1)        GET POINTER
        LH      R1,0(R1)        GET LENGTH
*       STC     R1,TRACE        USE IT AS FLAG
        L       R7,=A(SOCSAVE)  GET NEW SAVE AREA
        ST      R7,8(R13)       SAVE ADDRESS OF NEW SAVE AREA
        ST      R13,4(R7)       COMPLETE SAVE AREA CHAIN
        LR      R13,R7          NOW SWAP THEM
        L       R9,=A(MYCB)     POINT TO THE CONTROL BLOCK
        USING MYCB,R9          TELL ASSEMBLER
*-----*
*  BUILD MESSAGE FOR CONSOLE
*-----*
*                                     INITIALIZE MESSAGE TEXT FIELDS
LOOP     EQU      *
        MVC     MSGNUM(8),SUBTASK WHO I AM
        MVC     TYPE,MSGSTART    MOVE 'STARTED' TO MESSAGE
*
        MVC     MSGRSLT1,MSGSUCC ...SUCCESSFUL TEXT
        MVC     MSGRSLT2,BLANK35
*
        STM     R14,R12,12(R13) JUST FOR DEBUGGING
        BAL     R14,WTOSUB       --> DO STARTING WTO
*****
*                                     *
*       Issue INITAPI to connect to interface
*                                     *
*****
        POST    ECB,1           NEXT IS ALWAYS SYNCH
        MVI     SYNFLAG,0       MOVE A 1 FOR ASYNCH
        MVC     TYPE,MINITAPI   MOVE 'INITAPI' TO MESSAGE
*
        EZASMI TYPE=INITAPI,    Issue INITAPI Macro
                SUBTASK=SUBTASK, SPECIFY SUBTASK IDENTIFIER
                MAXSOC=MAXSOC,   SPECIFY MAXIMUM NUMBER OF SOCKETS
                MAXSNO=MAXSNO,   (HIGHEST SOCKET NUMBER ASSIGNED)
                ERRNO=ERRNO,     (Specify ERRNO field)
                RETCODE=RETCODE, (Specify RETCODE field)
                APITYPE=APITYPE, (SPECIFY APITYPE FIELD)
                ERROR=ERROR      ABEND IF ERROR ON MACRO
*       ASYNC=('EXIT',MYEXIT), (SPECIFY AN EXIT)
*       IDENT=IDENT,          TCP ADDR SPACE AND MY ADDR SPACE
*       ASYNC=('ECB')         (SPECIFY ECBS)
*
        BAL     R14,RCHECK      --> DID IT WORK?
*****
*                                     *
*       Issue SOCKET Macro to obtain a socket descriptor
*       *** INET and STREAM ***
*                                     *
*****
        MVC     TYPE,MSOCKET    MOVE 'SOCKET' TO MESSAGE
*
        EZASMI TYPE=SOCKET,     Issue SOCKET Macro
                AF='INET6',      INET, IUCV, INET6
                SOCTYPE='STREAM', STREAM(TCP) DATAGRAM(UDP) or RAW
                ERRNO=ERRNO,     (Specify ERRNO field)
                RETCODE=RETCODE, (Specify RETCODE field)
                ERROR=ERROR      Abend if Macro error
*       REQAREA=REQAREA,      IN CASE WE ARE DOING EXITS OR ECBS
*
        BAL     R14,RCHECK      CHECK FOR SUCCESSFUL CALL
*-----*
*       Get socket descriptor number

```

```

*-----*
      STH   R8,S          SAVE RETCODE (=SOCKET DESCRIPTOR)
*****
*
*      ISSUE PTON MACRO
*
*****
      MVC   PRESENTABLE_ADDR,LOOPIPV6 IP ADDRESS TO CONVERT
*
* DISPLAY THE RETURNED ADDRESS IN PRESENTABLE FORMAT
*
      MVC   TYPE,MPTON          MOVE 'PTON ' TO MESSAGE
*
      EZASMI TYPE=PTON,          ISSUE PTON MACRO
              AF='INET6',
              SRCADDR=PRESENTABLE_ADDR,
              SRCLEN=PRESENTABLE_ADDR_LEN,
              DSTADDR=NUMERIC_ADDR,
              ERRNO=ERRNO,      (SPECIFY ERRNO FIELD)
              RETCODE=RETCODE,  (SPECIFY RETCODE FIELD)
              ERROR=ERROR       ABEND IF MACRO ERROR
*
      BAL   R14,RCHECK          CHECK FOR SUCCESSFUL CALL
      MVC   ADDRESS,NUMERIC_ADDR
*****
*
*      ISSUE GETHOSTNAME CALL
*
*****
      MVC   TYPE,MGHOSTN        'GETHOSTN' TO MESSAGE
      EZASMI TYPE=GETHOSTNAME,
              NAMELEN=HOSTNAME_LEN, LENGTH OF HOST NAME FIELD
              NAME=HOSTNAME,        HOST NAME
              ERRNO=ERRNO,          (Specify ERRNO field)
              RETCODE=RETCODE,      (Specify RETCODE field)
              ERROR=ERROR           Abend if Macro error
*
      REQAREA=REQAREA,          IN CASE WE ARE DOING EXITS OR ECBS
*
      BAL   R14,RCHECK          CHECK FOR SUCCESSFUL CALL
*
      MVC   MSGRSLT1,=C'HOST NAME ' INDICATE WHAT WE'RE PROCESSING
      XC    MSGRSLT2,MSGRSLT2
      MVC   MSGRSLT2,HOSTNAME
      STM   R14,R12,12(R13)      JUST FOR DEBUGGING
      BAL   R14,WTOSUB           SEND TO THE CONSOLE
      MVC   NODENAME(24),HOSTNAME
*****
*
*      ISSUE GETADDRINFO MACRO
*
*****
      MVC   TYPE,MGADDRI        MOVE 'GETADDRINFO' TO MESSAGE
      XC    ADDR_INFO(addrinfo_len),ADDR_INFO CLEAR OUT ALL HINTS
      LA    R6,ai_CANONNAMEOK    REQUEST THE CANONICAL NAME
      ST    R6,ai_flags          SAVE THE HINT FLAGS
      LA    R6,ADDR_INFO         POINT TO THE HINTS ADDRINFO
      ST    R6,HINTS             SAVE THE ADDRESS OF THE HINTS
      LA    R6,0                 LENGTH OF SERVICE NAME
      ST    R6,SERVNAME_LEN      SAVE THE SERVICE NAME LENGTH
*
      EZASMI TYPE=GETADDRINFO,    ISSUE GETADDRINFO MACRO
              NODE=NODENAME,      NODE GETTING INFORMATION FOR
              NODELEN=NODENAME_LEN, LENGTH OF NODE NAME
              SERVICE=SERVNAME,    SERVICE GETTING INFORMATION FOR
              SERVLN=SERVNAME_LEN, LENGTH OF SERVICE NAME
              HINTS=HINTS,         HINTS FOR FILTERING
              RES=RESULT_ADDRINFO, RETURNED ADDRESS INFORMATION
              CANNLEN=CANNAME_LEN, LENGTH OF CANONICAL NAME
              ERRNO=ERRNO,        (SPECIFY ERRNO FIELD)
              RETCODE=RETCODE,    (SPECIFY RETCODE FIELD)
              ERROR=ERROR         ABEND IF MACRO ERROR
*
      BAL   R14,RCHECK          CHECK FOR SUCCESSFUL CALL
*
* IF RETURNED SUCCESSFULLY, THEN PROCESS THE ADDRINFO STRUCTURE AND
* THEN CHECK TO SEE IF THERE IS ANOTHER TO PROCESS. CONTINUE UNTIL
* AI_NEXT IS NULL.
*
      ICM   R10,B'1111',RESULT_ADDRINFO EXAMINE RETURNED ADDRINFO
      BZ    NOAIS              IF NOT RETURNED THEN HOST NOT FOUND?
      SEEAIS DS 0H
      MVC   ADDR_INFO(addrinfo_len),0(R10) LOAD ADDRINFO STRUCTURE

```

```

XC      OPNAMELEN,OPNAMELEN  CLEAR NAME LENGTH OUTPUT FIELD
XC      OPCANON,OPCANON      CLEAR CANONICAL NAME OUTPUT FIELD
XC      OPNAME,OPNAME        CLEAR NAME OUTPUT FIELD
XC      OPNEXT,OPNEXT        CLEAR NEXT ADDRINFO OUTPUT FIELD
*
*      CALL  EZACIC09,(RESULT_ADDRINFO,          X
*              OPNAMELEN,          OUTPUT NAME LENGTH      X
*              OPCANON,            OUTPUT CANONICAL NAME    X
*              OPNAME,            OUTPUT NAME              X
*              OPNEXT,            OUTPUT NEXT RESULT ADDRESS INFO  X
*              RETCODE),VL
*
*  FORMAT CANONNAME.
*
*      MVC      MSGRSLT1,=C'CANON NAME' INDICATE WHAT WE'RE PROCESSING
XC      MSGRSLT2,MSGRSLT2
MVC      MSGRSLT2(21),=C' - NO CANON NAME      '
XC      MSGRSLT2,MSGRSLT2
MVC      MSGRSLT2,OPCANON
FMTAISNC DS      0H
STM      R14,R12,12(R13)      JUST FOR DEBUGGING
BAL      R14,WTOSUB          SEND TO THE CONSOLE
FMTAISNCE DS      0H
*
*  IF AI_NEXT IS NULL THEN THIS IS THE LAST STRUCTURE ON THE LIST.
*  TO PROCESS ALL STRUCTURES:
*  1.  GET THE FIRST ONE AND PROCESS THE FIELDS RETURNED.
*  2.  USE THE ADDRESS IN AI_NEXT TO GET THE NEXT ADDRESS IF NOT NULL.
*  3.  PROCESS THE NEW FIELDS IN THE SUBSEQUENT STRUCTURE.
*  4.  GOTO 2.
*
*      ICM      R10,B'1111',ai_next SEE IF NEXT ADDRESS IS NULL...
BP      SEEAIS              NOPE...PARSE IT.
*
*****
*
*      ISSUE FREEADDRINFO MACRO.  MUST BE DRIVEN AFTER A
*      SUCCESSFUL GETADDRINFO; OTHERWISE, RESOLVER STORAGE WILL
*      BE CONSUMED.
*
*****
MVC      TYPE,MFADDRI      MOVE 'FREEADDRINFO' TO MESSAGE
*
EZASMI TYPE=FREEADDRINFO, ISSUE FREEADDRINFO MACRO          X
ADDRINFO=RESULT_ADDRINFO,          X
ERRNO=ERRNO,          (SPECIFY ERRNO FIELD)          X
RETCODE=RETCODE,      (SPECIFY RETCODE FIELD)          X
ERROR=ERROR          ABEND IF MACRO ERROR
*
BAL      R14,RCHECK        CHECK FOR SUCCESSFUL CALL
*
B      ENDAIS
NOAIS   DS      0H
XC      MSGRSLT2,MSGRSLT2
MVC      MSGRSLT2(21),=C'Result not returned. '
BAL      R14,WTOSUB        SEND TO THE CONSOLE
ENDAIS  DS      0H
*
*****
*
*      Issue BIND socket
*
*****
MVC      TYPE,MBIND        MOVE 'BIND' TO MESSAGE
MVC      PORT(2),PORTS      Load STREAM port #
*
EZASMI TYPE=BIND,          Issue Macro          X
S=S,          STREAM          X
NAME=NAME,      (SOCKET NAME STRUCTURE)          X
ERRNO=ERRNO,      (Specify ERRNO field)          X
RETCODE=RETCODE,      (Specify RETCODE field)          X
ERROR=ERROR      Abend if Macro error
*      REQAREA=REQAREA,      IN CASE WE ARE DOING EXITS OR ECBS  X
*
BAL      R14,RCHECK        CHECK FOR SUCCESSFUL CALL
*****
*
*      Issue LISTEN - Backlog = 5
*
*****
MVC      TYPE,MLISTEN      MOVE 'LISTEN' TO MESSAGE
*

```

```

EZASMI TYPE=LISTEN,      Issue Macro                                X
      S=S,                STREAM                                    X
      BACKLOG=BACKLOG,    BACKLOG                                  X
      ERRNO=ERRNO,        (Specify ERRNO field)                    X
      RETCODE=RETCODE,    (Specify RETCODE field)                  X
      ERROR=ERROR         Abend if Macro error
*      REQAREA=REQAREA,    IN CASE WE ARE DOING EXITS OR ECBS      X
*
      BAL   R14,RCHECK     CHECK FOR SUCCESSFUL CALL
*****
*
*      Issue ACCEPT - Block until connection from peer            *
*
*****
      MVC   TYPE,MACCEPT  MOVE 'ACCEPT' TO MESSAGE
      MVC   PORT(2),PORTS   Load STREAM port #
*
      EZASMI TYPE=ACCEPT,   Issue Macro                                X
      S=S,                STREAM                                    X
      NAME=NAME,          (SOCKET NAME STRUCTURE)                  X
      ERRNO=ERRNO,        (Specify ERRNO field)                    X
      RETCODE=RETCODE,    (Specify RETCODE field)                  X
      ERROR=ERROR         Abend if Macro error
*      REQAREA=REQAREA,    IN CASE WE ARE DOING EXITS OR ECBS      X
*
      BAL   R14,RCHECK     CHECK FOR SUCCESSFUL CALL
* Message RESULTS text
      STH   R8,SOCDESCA    SAVE RETCODE (SOCKET DESCRIPTOR)
*****
*
*      ISSUE NTOP MACRO                                           *
*
*****
      MVC   NUMERIC_ADDR,ADDRESS  IP ADDRESS FROM ACCEPT
*
* DISPLAY THE NUMERIC ADDRESS FIRST
*
      MVC   TYPE,MNTOP      MOVE 'NTOP ' TO MESSAGE
*
* TRANSLATE IT TO PRESENTABLE FORM
*
      EZASMI TYPE=NTOP,     ISSUE PTON MACRO                                X
      AF='INET6',          X
      SRCADDR=NUMERIC_ADDR, X
      DSTADDR=PRESENTABLE_ADDR, X
      DSTLEN=PRESENTABLE_ADDR_LEN, X
      ERRNO=ERRNO,         (SPECIFY ERRNO FIELD)                      X
      RETCODE=RETCODE,     (SPECIFY RETCODE FIELD)                    X
      ERROR=ERROR         ABEND IF MACRO ERROR
*
      BAL   R14,RCHECK     CHECK FOR SUCCESSFUL CALL
*
* DISPLAY THE RETURNED ADDRESS IN PRESENTABLE FORMAT
*
      MVC   MSGRSLT1,=C'DSTADDR ' INDICATE WHAT WE'RE PROCESSING
      XC    MSGRSLT2,MSGRSLT2
      MVC   MSGRSLT2(L'PRESENTABLE_ADDR),PRESENTABLE_ADDR
      STM   R14,R12,12(R13) JUST FOR DEBUGGING
      BAL   R14,WTOSUB     SEND TO THE CONSOLE
*****
*
*      Issue READ - Read data and store in buffer                *
*
*****
      MVC   TYPE,MREAD     MOVE 'READ ' TO MESSAGE
*
      EZASMI TYPE=READ,     Issue Macro                                X
      S=SOCDESCA,          ACCEPT SOCKET                            X
      NBYTE=NBYTE,         SIZE OF BUFFER                          X
      BUF=BUF,             (BUFFER)                                X
      ERRNO=ERRNO,         (Specify ERRNO field)                    X
      RETCODE=RETCODE,     (Specify RETCODE field)                  X
      ERROR=ERROR         Abend if Macro error
*      REQAREA=REQAREA,    IN CASE WE ARE DOING EXITS OR ECBS      X
*
      BAL   R14,RCHECK     CHECK FOR SUCCESSFUL CALL
      MVC   MSGRSLT1,MSGBUFF
      MVC   MSGRSLT2,BUF
      BAL   R14,WTOSUB     --> PRINT IT
*
*
*****

```

```

*
*      Issue WRITE - Write data from buffer
*
*****
MVC    TYPE,MWRITE      MOVE 'WRITE ' TO MESSAGE
*
      EZASMI TYPE=WRITE,      Issue Macro
      S=SOCDESCA,            ACCEPT Socket
      NBYTE=NBYTE,          SIZE OF BUFFER
      BUF=BUF,                (BUFFER)
      ERRNO=ERRNO,           (Specify ERRNO field)
      RETCODE=RETCODE,       (Specify RETCODE field)
      ERROR=ERROR            Abend if Macro error
*      REQAREA=REQAREA,      IN CASE WE ARE DOING EXITS OR ECBS
*
      BAL    R14,RCHECK      CHECK FOR SUCCESSFUL CALL
*****
*      Issue CLOSE for ACCEPT socket
*
*****
MVC    TYPE,MCLOSE      MOVE 'CLOSE' TO MESSAGE
*
      EZASMI TYPE=CLOSE,      Issue Macro
      S=SOCDESCA,            ACCEPT
      ERRNO=ERRNO,           (Specify ERRNO field)
      RETCODE=RETCODE,       (Specify RETCODE field)
      ERROR=ERROR            Abend if Macro error
*      REQAREA=REQAREA,      IN CASE WE ARE DOING EXITS OR ECBS
*
      MVC    MSGRSLT2,BLANK35
      BAL    R14,RCHECK      CHECK FOR SUCCESSFUL CALL
*****
*      Terminate Connection to API
*
*****
MVC    TYPE,MTERMAPI     MOVE 'TERMAPI' TO MESSAGE
*
      POST   ECB,1           FOLLOWING IS ALWAYS SYNCH
      EZASMI TYPE=TERMAPI     Issue EZASMI Macro for TermapI
*-----*
* Message RESULTS text
MVC    MSGRSLT2,BLANK35
*
      BAL    R14,RCHECK      --> CHECK RC
*-----*
*      Issue console message for task termination
*-----*
MVC    TYPE,MSGEND       Move 'ENDED' to message
*
MVC    MSGRSLT1,MSGSUCC   ...SUCCESSFUL text
MVC    MSGRSLT2,BLANK35
*
      BAL    R14,WTOSUB
      LA     R14,1           CONSTANT
      AH     R14,APITYPE     ADD
      STH    R14,APITYPE     STORE
      CH     R14,=H'3'       COMPARE
*      BE     LOOP           --> LETS DO IT AGAIN!
*-----*
*      Return to Caller
*-----*
L      R13,4(R13)
RETURN (14,12),T,RC=0
WTOSUB EQU *
      LR     R7,R14          COPY RETURN REG
      MVC    MSGCMD(8),TYPE  WRITE MESSAGE TO OPERATOR
      WTO    TEXT=MSG        --> RETURN TO CALLER
      BR     R7
      CNOP   2,4
*      USES   R6,R7,R8        RETCODE RETURNED IN R8
RCHECK EQU *
      LR     R7,R14          COPY TO REAL RETURN REG
      MVC    MSGRSLT1,MSGSUCC ...SUCCESS TEXT
      L      R6,RETCODE
      LTR    R6,R6
      BM     NOWAIT
      CLI    SYNFLAG,0       PLAIN CASE?
      BE     NOWAIT          --> SKIP IT
      MVC    KEY+14(8),SUBTASK

```

```

KEY      MVC      KEY+23(8),TYPE
WTO      'WAIT: XXXXXXXX XXXXXXXX'
WAIT     ECB=ECB
NOWAIT   EQU      *
*        LA       R15,ECB
*        ST       R15,ECB
*        ST       R9,ECB      MAKE THIS THE TOKEN AGAIN
*        L        R6,RETCODE  CHECK FOR SUCCESSFUL CALL
*        CLC      TYPE,=CL8'GETHOSTI'
*        BE       HOSTIDRC    HANDLE PRINTING HOST ID
*        LTR      R8,R6       SAVE A COPY
*
*        BNL      CONT00
FAILMSG  EQU      *
MVC      MSGRSLT1,MSGFAIL    ...FAIL TEXT
CONT00   EQU      *
*
*-----*
*        FORMAT THE RETCODE= -XXXXXXX ERRNO= XXXXXXX MSG RESULTS
*        ***> R6 = RETCODE VALUE ON ENTRY
*-----*
*
MVC      MSGRTCT,MSGRETC     ' RETCODE= '
MVI      MSGRTCS,C'+ '
LTR      R6,R6
BNM      NOTM               -->
MVI      MSGRTCS,C'- '      MOVE SIGN WHICH IS ALWAYS MINUS
EQU      *
MVC      MSGERRT,MSGERRN     ' ERRNO= '
*
*        CVD      R6,DWORK    CONVERT IT TO DECIMAL
UNPK     MSGRTCV,DWORK+4(4)  UNPACK IT
OI       MSGRTCV+6,X'F0'    CORRECT THE SIGN
ERRNOFMT EQU      *
L        R6,ERRNO           GET ERRNO VALUE
CVD      R6,DWORK           CONVERT IT TO DECIMAL
UNPK     MSGERRV,DWORK+4(4)  UNPACK IT
OI       MSGERRV+6,X'F0'    CORRECT THE SIGN
*
*        MVC      MSGRSLT2(35),MSGRTCD
*
MVI      MSGRTHX,X'40'      CLEAR HEX INDICATOR
SR       R6,R6              CLEAR OUT...
ST       R6,RETCODE         RETCODE AND...
ST       R6,ERRNO          ERRNO
*
*
*        CLI      TRACE,0
BNE      NOTRACE
LR       R14,R7             GIVE HIM RETURN REG
B        WTOSUB             --> DO WTO
NOTRACE  EQU      *
BR       R7                 --> RETURN TO CALLER
*
*        HOSTIDRC EQU      *      VALID HOSTID MAY LOOK LIKE NEG. RC
C        R6,=F'-1'          ONLY -1 RC INDICATES FAILURE
BE       FAILMSG            ...BAD RC, USE STANDARD MSG
LR       R8,R6              ...NEXT CALL EXPECTS ADDR IN R8
MVC      MSGRSLT1,MSGSUCC   ...SUCCESS TEXT
UNPK     HEXRC(9),RETCODE(5) PLUS ONE FOR FAKE SIGN
TR       HEXRC(8),HEXTAB    ...CONVERT UNPK TO PRINTABLE HEX
MVI      HEXRC+8,X'40'      ...SPACE OUT FAKED SIGN BYTE
MVI      MSGRTHX,C'X'       ...INDICATE INFO IS HEX
B        ERRNOFMT
*
*        SYNFLAG  DC       H'0'      DEFAULT TO SYN
TRACE    DC       H'0'      DEFAULT TO TRACE
MYEXIT   DC       A(MYEXIT1,SUBTASK)
MYEXIT1  SAVE     (14,12),T,*
LR       R2,R15
USING    MYEXIT1,R2
LM       R8,R9,0(R1)        GET TWO TOKENS
MVC      EXKEY+14(8),0(R8)   TELL WHO
MVC      EXKEY+23(8),TYPE    TELL WHAT
EXKEY    WTO      'EXIT: XXXXXXXX XXXXXXXX'
POST     ECB,1
RETURN   (14,12),T,RC=0
DROP     R2
*-----*
*        ABEND PROGRAM AND GET DUMP
*-----*
*        ERROR    ABEND 1,DUMP
*-----*

```

```

* CONSTANTS USED TO RUN PROGRAM *
*-----*
EZASMGW  EZASMI TYPE=GLOBAL,      Storage definition for GWA      X
          STORAGE=CSECT
*-----*
* INITAPI macro parms *
*-----*
SUBTASK  DC    CL8'EZAS06AS'      SUBTASK PARM VALUE
MAXSOC   DC    AL2(50)            MAXSOC PARM VALUE
APITYPE  DC    H'2'              OR A 3
MAXSNO   DC    F'0'              (HIGHEST SOCKET DESCRIPTOR AVAILABLE)
IDENT    DC    0CL16' '
          DC    CL8' '            NAME OF TCP TO WHICH CONNECTING
          DC    CL8'SOC401CB'      MY ADDR SPACE NAME
*-----*
* SOCKET macro parms *
*-----*
S         DC    H'0'              SOCKET DESCRIPTOR FOR STREAM
*-----*
* BIND MACRO PARMS *
*-----*
          CNOP  0,4
NAME      DC    0CL28' '          SOCKET IPV6 NAME STRUCTURE
          DC    AL1(16)           Address Length
          DC    AL1(19)           Family
PORT      DC    H'0'
FLOWINFO  DC    XL4'00'
ADDRESS   DC    XL16'FF'
          DC    XL4'00'           SCOPEID
ADDR      DC    XL16'00000000000000000000000000000001' Internet Address
PORTS     DC    H'11007'
*-----*
* LISTEN PARMS *
*-----*
BACKLOG   DC    F'5'             BACKLOG
*-----*
* READ MACRO PARMS *
*-----*
NBYTE     DC    F'50'            SIZE OF BUFFER
SOCDESCA  DC    H'0'            SOCKET DESCRIPTOR FROM ACCEPT
BUF        DC    CL50' THIS SHOULD NEVER APPEAR!!! : ('
*-----*
* WTO FRAGMENTS *
*-----*
MNTOP     DC    CL8'NTOP '
MPTON     DC    CL8'PTON '
MFADDRI   DC    CL8'FADDRI '
MGADDRI   DC    CL8'GADDRI '
MGHOSTN   DC    CL8'GETHOSTN'
MGNAMEI   DC    CL8'GNAMEI '
MINITAPI  DC    CL8'INITAPI'
MSOCKET   DC    CL8'SOCKET'
MBIND     DC    CL8'BIND'
MACCEPT DC    CL8'ACCEPT'
MLISTEN   DC    CL8'LISTEN'
MREAD     DC    CL8'READ'
MWRITE    DC    CL8'WRITE'
MCLOSE    DC    CL8'CLOSE'
MTERMAPI  DC    CL8'TERMAPI'
MSGSTART  DC    CL8' STARTED'
MSGEND    DC    CL8' ENDED '
MSGBUFF   DC    CL10' BUFFER: '
MSGSUCC   DC    CL10' SUCCESS '      Command results...
MSGFAIL   DC    CL10' FAIL: ( '      ...
MSGRETC   DC    CL10' RETCODE= '    ...
MSGERRN   DC    CL10' ERRNO= '      ...
BLANK35   DC    CL35' '
*-----*
* ERROR NUMBER / RETURN CODE FIELDS *
*-----*
* MESSAGE AREA *
*-----*
MSG        DC    0F'0'            MESSAGE AREA
          DC    AL2(MSGE-MSGNUM)  LENGTH OF MESSAGE
MSGNUM     DC    CL10'EZAS06AS:'  'EZAS06ASXX:'
MSGCMD     DC    CL8' '           COMMAND ISSUED
MSGRSLT1   DC    CL10' '         COMMAND RESULTS (SUCC, PASS, FAIL)
MSGRSLT2   DC    CL35' '         RETURNED VALUES
MSGE       EQU    *              End of message
*-----*
* MESSAGE RESULTS AREAS (fill in and move to MSGRSLT2) *

```

```

*-----*
MSGRTCD DC 0CL35' ' GENERAL RETURNED VALUE
MSGRTCT DC CL9' RETCODE=' ' RETCODE= '
MSGRTHX DC CL1' ' 'X' X FOR GETHOSTID
MSGRTCS DC CL1' ' '-' (NEGATIVE SIGN)
HEXRC EQU MSGRTCS HEX RC WILL START AT SIGN LOCATION
MSGRTCV DC CL7' ' RETURNED VALUE (RETCODE)
MSGERRT DC CL10' ERRNO=' ' ERRNO= '
MSGERRV DC CL7' ' RETURNED VALUE (ERRNO)
*-----*
PARMADDR DC A(0) PARM ADDRESS SAVE AREA
DWORK DC D'0' WORK AREA
HEXTAB EQU *-240 TAB TO CONVERT TO PRINTABLE HEX
* FIRST 240 BYTES NOT REFERENCED
DC CL16'0123456789ABCDEF'
EZBREHST DSECT=NO,LIST=YES,HOSTENT=NO,ADRINFO=YES
LTORG ,
*-----*
* REG/SAVEAREA *
*-----*
SOCSAVE DC 9D'0' SAVE AREA
CNOP 0,8
MYCB EQU * MY CONTROL BLOCK
REQAREA EQU *
ECB DC A(ECB) SELF POINTER
DC CL100'WORK AREA'
MYTIE EZASMI TYPE=TASK,STORAGE=CSECT TIE
TYPE DC CL8'TYPE'
ERRNO DC F'0'
RETCODE DC F'0'
*
REQARG DC F'1'
RETARG DS 0H
*
* FOR NTOP AND PTON
*
NUMERIC_ADDR DS CL16 IP ADDRESS IN NUMERIC FORM
PRESENTABLE_ADDR DS CL45 IP ADDRESS IN PRESENTABLE FORM
PRESENTABLE_ADDR_LEN DC AL2(L'PRESENTABLE_ADDR) LENGTH OF PRESENTABLE X
IP ADDRESS
LOOPIPV6 DC CL45'::1' IPV6 LOOPBACK ADDRESS
*
* FOR GETHOSTNAME, GETADDRINFO, and FREEADDRINFO
*
HOSTNAME DC CL24' '
NODENAME DC CL255' ' FOR THE RETURNED HOST NAME
SERVNAME DC C' ' SERVICE BEING RESOLVED
CNOP 0,4
HOSTNAME_LEN DC AL4(L'HOSTNAME) LENGTH OF THE HOST NAME
NODENAME_LEN DC AL4(L'NODENAME) LENGTH OF THE NODE NAME
SERVNAME_LEN DC F'0' LENGTH OF THE SERVICE NAME
RESULT_ADDRINFO DC F'0' RETURNED ADDRINFO
CANNONAME_LEN DC F'0' CANNONICAL NAME LENGTH IN ADDRINFO
HINTS DC F'0' ADDRESS OF HINTS ADDRINFO
*
* For EZACIC09 processing
*
OPNAMELEN DS F SOCKET ADDRESS STRUCTURE LENGTH
OPCANON DS CL256 CANONICAL NAME
OPNAME DS CL28 SOCKET ADDRESS STRUCTURE
OPNEXT DS F NEXT RESULT ADDRESS INFO IN CHAIN
*
MYNEXT DC A(MYCB) NEXT IN CHAIN FOR MULTIPLES
CNOP 0,8
MYLEN EQU *-MYCB
MYCB2 EQU *
ORG **MYLEN
CNOP 0,8
DC CL8'&SYSDATE'
DC CL8'&SYSTIME'
BPXYSOCK DSECT=NO,LIST=YES
END

```

Figure 69. EZASO6AS sample server program for IPv6

EZAS06AC sample client program for IPv6

The EZAS06AC program is a client module that shows you how to use the following calls provided by the macro socket interface:

- INITAPI
- SOCKET
- CONNECT
- GETPEERNAME
- GETNAMEINFO
- GLOBAL
- WRITE
- READ
- TASK
- TERMAPI
- SHUTDOWN

```
EZAS06AC CSECT
EZAS06AC AMODE ANY
EZAS06AC RMODE ANY
PRINT NOGEN
*****
*
*   MODULE NAME:  EZAS06AC - THIS IS A VERY SIMPLE IPV6 CLIENT
*
* Copyright:    Licensed Materials - Property of IBM
*
*               "Restricted Materials of IBM"
*
*               5694-A01
*
*               (C) Copyright IBM Corp. 2002, 2003
*
*               US Government Users Restricted Rights -
*               Use, duplication or disclosure restricted by
*               GSA ADP Schedule Contract with IBM Corp.
*
* Status:      CSV1R5
*
*
*   LANGUAGE:   ASSEMBLER
*
*   ATTRIBUTES: NON-REUSEABLE
*
*   REGISTER USAGE:
*       R1 =
*       R2 =
*       R3 = BASE REG 1
*       R4 = BASE REG 2 (UNUSED)
*       R5 = FUTURE BASE?
*       R6 = TEMP
*       R7 = RETURN REG
*       R8 =
*       R9 = A(WORK AREA)
*       R10 =
*       R11 =
*       R12 =
*       R13 = SAVE AREA
*       R14 =
*       R15 =
*
*   INPUT: ANY PARM TURNS TRACE OFF, NO PARM IS NOISY MODE
*   OUTPUT: WTO RESULTS OF EACH TEST CASE IF TRACING
*           RETURN CODE IS 0 WHETHER IT CONNECTS OR NOT!
*
*****
GBLB  &TRACE  ASSEMBLER VARIABLE TO CONTROL TRACE GENERATION
&TRACE SETB  1      1=TRACE ON  0=TRACE OFF
R0     EQU   0
R1     EQU   1
```

```

R2      EQU      2
R3      EQU      3
R4      EQU      4
R5      EQU      5
R6      EQU      6
R7      EQU      7
R8      EQU      8
R9      EQU      9
R10     EQU      10
R11     EQU      11
R12     EQU      12
R13     EQU      13
R14     EQU      14
R15     EQU      15
-----*
*  START OF EXECUTABLE CODE
*-----*
          USING *,R3,R4          TELL ASSEMBLER OF OTHERS
          SAVE (14,12),T,*
          LR      R3,R15          COPY EP REG TO FIRST BASE
          LA      R5,2048          GET R5 HALFWAY THERE
          LA      R5,2048(R5)      GET R5 THERE
          LA      R4,0(R5,R3)      GET R4 THERE
          LA      R12,12           JUST FOR FUN!
          ST      R1,PARMADDR      SAVE ADDRESS OF PARAMETER LIST
          L       R1,0(R1)         GET POINTER
          LH      R1,0(R1)         GET LENGTH
*          STC     R1,TRACE         USE IT AS FLAG
          L       R7,=A(SOCSAVE)   GET NEW SAVE AREA
          ST      R7,8(R13)        SAVE ADDRESS OF NEW SAVE AREA
          ST      R13,4(R7)        COMPLETE SAVE AREA CHAIN
          LR      R13,R7           NOW SWAP THEM
          L       R9,=A(MYCB)      POINT TO THE CONTROL BLOCK
          USING MYCB,R9           TELL ASSEMBLER
-----*
*  BUILD MESSAGE FOR CONSOLE
*-----*
*                                     INITIALIZE MESSAGE TEXT FIELDS
LOOP      EQU      *
          MVC     MSGNUM(8),SUBTASK WHO I AM
          MVC     TYPE,MSGSTART     MOVE 'STARTED' TO MESSAGE
*
          MVC     MSGRSLT1,MSGSUCC ...SUCCESSFUL TEXT
          MVC     MSGRSLT2,BLANK35
*
          STM     R14,R12,12(R13)  JUST FOR DEBUGGING
          BAL     R14,WTO SUB      --> DO STARTING WTO
*****
*                                     *
*      Issue INITAPI to connect to interface
*                                     *
*****
          MVC     TYPE,MINITAPI     MOVE 'INITAPI' TO MESSAGE
*
          POST    ECB,1             FOLLOWING IS SYNC ONLY
          MVI     SYNFLAG,0         MOVE A 1 FOR ASYNCH
          EZASMI  TYPE=INITAPI,     ISSUE INITAPI MACRO
                                SUBTASK=SUBTASK, SPECIFY SUBTASK IDENTIFIER
                                MAXSOC=MAXSOC,   SPECIFY MAXIMUM NUMBER OF SOCKETS
                                MAXSNO=MAXSNO,   (HIGHEST SOCKET NUMBER ASSIGNED)
                                ERRNO=ERRNO,     (Specify ERRNO field)
                                RETCODE=RETCODE, (Specify RETCODE field)
                                APITYPE=APITYPE, (SPECIFY APITYPE FIELD)
                                ERROR=ERROR      Abend if error on macro
*                                IDENT=IDENT,     TCP ADDR SPACE AND MY ADDR SPACE
*
*                                ASYNC=('ECB'),   (SPECIFY TO USE ECBS)
*                                ASYNC=('EXIT',MYEXIT) (SPECIFY TO USE EXITS)
          BAL     R14,RCHECK        --> CHECK RESULTS
*****
*                                     *
*      Issue SOCKET Macro to obtain a socket descriptor
*      *** INET and STREAM ***
*                                     *
*****
          MVC     TYPE,MSOCKET      MOVE 'SOCKET' TO MESSAGE
*
          EZASMI  TYPE=SOCKET,      Issue SOCKET Macro
                                AF='INET6',     INET, IUCV, or INET6
                                SOCTYPE='STREAM', STREAM(TCP) DATAGRAM(UDP) or RAW
                                ERRNO=ERRNO,    (Specify ERRNO field)
                                RETCODE=RETCODE, (Specify RETCODE field)

```

```

REQAREA=REQAREA,    FOR EXITS (AND ECBS)          X
ERROR=ERROR          Abend if Macro error

*
    BAL    R14,RCHECK    --> CHECK RESULTS
    STH    R8,S          SAVE RETCODE (=SOCKET DESCRIPTOR)
    LTR    R8,R8         CHECK IT
    BM     DOSHUTDO      --> WE ARE DONE!
*****
*
*       Issue CONNECT Socket
*
*****
    MVC    TYPE,MCONNECT    MOVE 'CONNECT' TO MESSAGE
    MVC    PORT(2),PORTS    Load STREAM port #
    MVC    ADDRESS(16),ADDR  LOAD THE INTERNET ADDRESS
*
    EZASMI TYPE=CONNECT,    Issue Macro          X
           S=S,            STREAM              X
           NAME=NAME,      SOCKET NAME STRUCTURE
           ERRNO=ERRNO,    (Specify ERRNO field)
           RETCODE=RETCODE,(Specify RETCODE field)
           REQAREA=REQAREA, FOR EXITS (AND ECBS)
           ERROR=ERROR     Abend if Macro error
*
    BAL    R14,RCHECK    --> CHECK RC
    LTR    R8,R8         RECHECK IT
    BM     DOSHUTDO      --> WE ARE DONE
*****
*
*       Issue GETPEERNAME
*
*****
    MVC    TYPE,MGETPEER    MOVE 'GTPEERN' TO MESSAGE
*
    EZASMI TYPE=GETPEERNAME, Issue Macro          X
           S=S,            STREAM              X
           NAME=NAME,      (SOCKET NAME STRUCTURE)
           ERRNO=ERRNO,    (Specify ERRNO field)
           RETCODE=RETCODE,(Specify RETCODE field)
           REQAREA=REQAREA, FOR EXITS (AND ECBS)
           ERROR=ERROR     Abend if Macro error
*
    BAL    R14,RCHECK    --> CHECK RC
*****
*
*       ISSUE GETNAMEINFO MACRO
*
*****
    MVC    TYPE,MNAMEI      MOVE 'GETNAMEINFO' TO MESSAGE
    LA     R6,NI_NAMEREQD
    ST     R6,FLAGS
*
    EZASMI TYPE=GETNAMEINFO, ISSUE GETNAMEINFO MACRO
           NAME=NAME,
           NAMELEN=NAMELEN,
           HOST=HOSTNAME,
           HOSTLEN=HOSTNAME,
           SERVICE=SERVNAME,
           SERVLN=SERVNAME,
           FLAGS=FLAGS,
           ERRNO=ERRNO,    (SPECIFY ERRNO FIELD)
           RETCODE=RETCODE,(SPECIFY RETCODE FIELD)
           ERROR=ERROR     ABEND IF MACRO ERROR
*
    BAL    R14,RCHECK    CHECK FOR SUCCESSFUL CALL
*
*       DISPLAY HOSTNAME
*
    MVC    MSGRSLT1,=C'HOST NAME ' INDICATE WHAT WERE PROCESSING
    XC     MSGRSLT2,MSGRSLT2
    MVC    MSGRSLT2,HOSTNAME    LOAD UP THE DATA
    STM    R14,R12,12(R13)    JUST FOR DEBUGGING
    BAL    R14,WTOSUB        SEND TO THE CONSOLE
*
*       DISPLAY SERVNAME
*
    MVC    MSGRSLT1,=C'SERV NAME ' INDICATE WHAT WERE PROCESSING
    XC     MSGRSLT2,MSGRSLT2
    MVC    MSGRSLT2,SERVNAME    LOAD UP THE DATA
    STM    R14,R12,12(R13)    JUST FOR DEBUGGING
    BAL    R14,WTOSUB        SEND TO THE CONSOLE
*****

```

```

*
*      Issue WRITE - Write data from buffer
*
*****
MVC   TYPE,MWRITE      MOVE 'WRITE ' TO MESSAGE
*
      EZASMI TYPE=WRITE,      Issue Macro
                                S=S,      STREAM SOCKET
                                NBYTE=NBYTE,  SIZE OF BUFFER
                                BUF=BUF,      BUFFER
                                ERRNO=ERRNO,  (Specify ERRNO field)
                                RETCODE=RETCODE, (Specify RETCODE field)
                                REQAREA=REQAREA, FOR EXITS (AND ECBS)
                                ERROR=ERROR    Abend if Macro error
*
      BAL   R14,RCHECK      --> CHECK RC
*****
*
*      Issue SHUTDOWN - HOW = 1 (end communication TO socket)
*
*****
DOSHUTDO EQU *
MVC   HOW(4),=F'1'
*
      BAL   R14,SHUTSUB      --> SHUTDOWN
*
      BAL   R14,RCHECK      --> CHECK RC
*****
*
*      Issue READ - Read data and store in buffer
*
*****
MVC   TYPE,MREAD      MOVE 'READ ' TO MESSAGE
*
      EZASMI TYPE=READ,      Issue Macro
                                S=S,      STREAM SOCKET
                                NBYTE=NBYTE,  SIZE OF BUFFER
                                BUF=BUF2,      (BUFFER)
                                ERRNO=ERRNO,  (Specify ERRNO field)
                                RETCODE=RETCODE, (Specify RETCODE field)
                                REQAREA=REQAREA, FOR EXITS (AND ECBS)
                                ERROR=ERROR    Abend if Macro error
*
      BAL   R14,RCHECK      --> CHECK RC
      MVC   MSGRSLT1,MSGBUFF TITLE
      MVC   MSGRSLT2,BUF2    MOVE THE DATA
      BAL   R14,WTOSUB      --> PRINT IT
*****
*
*      Issue SHUTDOWN - HOW = 0 (end communication FROM socket)
*
*****
MVC   HOW(4),=F'0'
*
      BAL   R14,SHUTSUB      --> SHUTDOWN
*
      BAL   R14,RCHECK      --> CHECK RC
*****
*
*      Terminate Connection to API
*
*****
MVC   TYPE,MTERMAPI      MOVE 'TERMAPI' TO MESSAGE
*
      POST  ECB,1          FOLLOWING IS SYNC ONLY
      EZASMI TYPE=TERMAPI  Issue EZASMI Macro for Termap
*
      BAL   R14,RCHECK      --> CHECK RC
*-----*
*      Issue console message for task termination
*-----*
MVC   TYPE,MSGEND      Move 'ENDED' to message
*
      MVC   MSGRSLT1,MSGSUCC ..SUCCESSFUL text
      MVC   MSGRSLT2,BLANK35
      BAL   R14,WTOSUB      --> DO WTO
      LA    R14,1          CONSTANT
      AH    R14,APITYPE     ADD
      STH   R14,APITYPE     STORE
      CH    R14,=H'3'       COMPARE
*
      BE    LOOP            --> LETS DO IT AGAIN!
*

```

```

*-----*
*      Return to Caller
*-----*
WTOSUB   L      R13,4(R13)
         RETURN (14,12),T,RC=0
         EQU    *
         LR     R7,R14          SAVE RETURN REG
         MVC    MSGCMD,TYPE     COPY COMMAND
         WTO    TEXT=MSG
         BR     R7              --> RETURN

*
SHUTSUB  EQU    *
         LR     R7,R14
         MVC    TYPE,MSHUTDOWN MOVE 'SHUTDOWN' TO MESSAGE

*
         EZASMI TYPE=SHUTDOWN, Issue Macro
                     S=S,      STREAM
                     HOW=HOW,  End communication in both directions
                     ERRNO=ERRNO, (Specify ERRNO field)
                     RETCODE=RETCODE, (Specify RETCODE field)
                     REQAREA=REQAREA, FOR EXITS (AND ECBS)
                     ERROR=ERROR    Abend if Macro error
                                     X
                                     X
                                     X
                                     X
                                     X

*
         BR     R7              --> RETURN TO CALLER
*-----*
*      ABEND PROGRAM AND GET DUMP TO DEBUG!
*      ERROR
         ABEND 1,DUMP
         CNOB 2,4
*      RCHECK
         USES R6,R7,R8          RETCODE RETURNED IN R8
         EQU    *
         LR     R7,R14          COPY TO REAL RETURN REG
         MVC    MSGRSLT1,MSGSUCC ...SUCCESS TEXT
         L      R6,RETCODE
         LTR    R6,R6
         BM     NOWAIT
         CLI    SYNFLAG,0      PLAIN CASE?
         BE     NOWAIT         --> SKIP IT
         MVC    KEY+14(8),SUBTASK
         MVC    KEY+23(8),TYPE
KEY      WTO    'WAIT: XXXXXXXX XXXXXXXX'
         WAIT   ECB=ECB
NOWAIT   EQU    *
*      LA      R15,ECB
*      ST      R15,ECB
*      ST      R9,ECB          MAKE THIS THE TOKEN AGAIN
         L      R6,RETCODE      CHECK FOR SUCCESSFUL CALL
         CLC    TYPE,=CL8'GETHOSTI'
         BE     HOSTIDRC        HANDLE PRINTING HOST ID
         LTR    R8,R6          SAVE A COPY

*
         BNL    CONT00
FAILMSG  EQU    *
         MVC    MSGRSLT1,MSGFAIL ...FAIL TEXT
CONT00   EQU    *
*-----*
*      FORMAT THE RETCODE= -XXXXXXX ERRNO= XXXXXXXX MSG RESULTS
*      ***> R6 = RETCODE VALUE ON ENTRY
*-----*
         MVC    MSGRTCT,MSGRETC ' RETCODE= '
         MVI    MSGRTCS,C'+'
         LTR    R6,R6
         BNM    NOTM           -->
         MVI    MSGRTCS,C'-'   MOVE SIGN WHICH IS ALWAYS MINUS
NOTM     EQU    *
         MVC    MSGERRT,MSGERRN ' ERRNO= '

*
         CVD    R6,DWORK        CONVERT IT TO DECIMAL
         UNPK   MSGRTCV,DWORK+4(4) UNPACK IT
         OI     MSGRTCV+6,X'F0'  CORRECT THE SIGN

*
ERRNOFMT EQU    *
         L      R6,ERRNO        GET ERRNO VALUE
         CVD    R6,DWORK        CONVERT IT TO DECIMAL
         UNPK   MSGGERRV,DWORK+4(4) UNPACK IT
         OI     MSGGERRV+6,X'F0' CORRECT THE SIGN

*
         MVC    MSGRSLT2(35),MSGRTCD

*
         MVI    MSGRTHX,X'40'   CLEAR HEX INDICATOR
         SR     R6,R6           CLEAR OUT...
         ST     R6,RETCODE      RETCODE AND...

```

```

*      ST      R6,ERRNO      ERRNO
*
*      CLI      TRACE,0
*      BNE      NOTRACE
*      LR      R14,R7      GIVE HIM RETURN REG
*      B      WTOSUB      --> DO WTO
NOTRACE EQU *
*      BR      R7      --> RETURN TO CALLER
*
HOSTIDRC EQU *      VALID HOSTID MAY LOOK LIKE NEG. RC
*      C      R6,=F'-1'      ONLY -1 RC INDICATES FAILURE
*      BE      FAILMSG      ...BAD RC, USE STANDARD MSG
*      LR      R8,R6      ...NEXT CALL EXPECTS ADDR IN R8
*      MVC      MSGRSLT1,MSGSUCC ...SUCCESS TEXT
*      UNPK      HEXRC(9),RETCODE(5) PLUS ONE FOR FAKE SIGN
*      TR      HEXRC(8),HEXTAB ...CONVERT UNPK TO PRINTABLE HEX
*      MVI      HEXRC+8,X'40' ...SPACE OUT FAKED SIGN BYTE
*      MVI      MSGRTHX,C'X' ...INDICATE INFO IS HEX
*      B      ERNOFMT
*
SYNFLAG DC H'0'      DEFAULT TO SYN
TRACE DC H'0'      DEFAULT TO TRACE
MYEXIT DC A(MYEXIT1,SUBTASK)
MYEXIT1 SAVE (14,12),T,*
*      LR      R2,R15
*      USING MYEXIT1,R2
*      LM      R8,R9,0(R1)      GET TWO TOKENS
*      MVC      EXKEY+14(8),0(R8) TELL WHO
*      MVC      EXKEY+23(8),TYPE TELL WHAT
EXKEY WTO 'EXIT: XXXXXXXX XXXXXXXX'
*      POST ECB,1
*      RETURN (14,12),T,RC=0
*      DROP R2
*-----*
* ELEMENTS USED TO RUN PROGRAM *
*-----*
EZASMGW EZASMI TYPE=GLOBAL,      STORAGE DEFINITION FOR GWA      X
*      STORAGE=CSECT
*-----*
* INITAPI macro parms *
*-----*
SUBTASK DC CL8'EZAS06AC'      SUBTASK PARM VALUE
IDENT DC 0CL16' '
*      DC      CL8'TCPV32'      DEFAULT TO FIRST ONE AVAILABLE
*      DC      CL8'EZAS06AC' MY ADDR SPACE NAME OR JOBNAME
MAXSNO DC F'0'      (HIGHEST SOCKET DESCRIPTOR AVAILABLE)
MAXSOC DC AL2(50)      MAXSOC PARM VALUE
APITYPE DC H'2'      OR PUT A 3 HERE
*-----*
* SOCKET macro parms *
*-----*
S DC H'0'      SOCKET DESCRIPTOR FOR STREAM
*-----*
* CONNECT MACRO PARMS *
*-----*
CNOPI 0,4
NAME DC 0CL28' '      SOCKET IPV6 NAME STRUCTURE
*      DC      AL1(16) Address Length
*      DC      AL1(19) Family
PORT DC H'0'
FLOWINFO DC XL4'00'
ADDRESS DC XL16'FF'
*      DC      XL4'00' SCOPEID
ADDR DC XL16'00000000000000000000000000000001' Internet Address
PORTS DC H'11007'
*-----*
* WRITE MACRO PARMS *
*-----*
NBYTE DC F'50'      SIZE OF BUFFER
BUF DC CL50' THIS IS FROM EZAS06AC!' BUFFER FOR WRITE
*-----*
* SHUTDOWN MACRO PARMS *
*-----*
HOW DC F'2'      END COMMUNICATION TO- AND FROM-SOCKET
*-----*
* READ MACRO PARMS *
*-----*
BUF2 DC CL50'BUF2'      BUFFER FOR READ
*-----*
*-----*
MNTOP DC CL8'NTOP '

```

```

MPTON      DC      CL8'PTON      '
MFADDRI    DC      CL8'FADDRI    '
MGADDRI    DC      CL8'GADDRI    '
MGNAMEI    DC      CL8'GNAMEI    '
MINITAPI   DC      CL8'INITAPI   '
MSOCKET     DC      CL8'SOCKET    '
MCONNECT   DC      CL8'CONNECT   '
MGETPEER   DC      CL8'GETPEERN   '
MREAD       DC      CL8'READ      '
MWRITE      DC      CL8'WRITE     '
MSHUTDOWN  DC      CL8'SHUTDOWN   '
MTERMAPI   DC      CL8'TERMAPI   '
MSGSTART    DC      CL8' STARTED  '
MSGEND      DC      CL8' ENDED    '
MSGSUCC     DC      CL10' SUCCESS '      Command results...
MSGFAIL     DC      CL10' FAIL:-('      ...
MSGRETC     DC      CL10' RETCODE='      ...
MSGERRN     DC      CL10' ERRNO='      ...
MSGBUFF     DC      CL10' BUFFER:'      ...
BLANK35     DC      CL35' '

*-----*
* MESSAGE AREA *
*-----*
MSG         DC      0F'0'          MESSAGE AREA
          DC      AL2(MSGE-MSGNUM)  LENGTH OF MESSAGE
MSGNUM      DC      CL10'EZAS06AC:' 'EZAS06AC:'
MSGCMD      DC      CL8' '          COMMAND ISSUED
MSGRSLT1    DC      CL10' '          COMMAND RESULTS (SUCC, PASS, FAIL)
MSGRSLT2    DC      CL35' '          RETURNED VALUES
MSGE        EQU      *              End of message
*-----*
* MESSAGE RESULTS AREAS (fill in and move to MSGRSLT2) *
*-----*
*
MSGRTCD     DC      0CL35' '          GENERAL RETURNED VALUE
MSGRTCT     DC      CL9' RETCODE='    ' RETCODE='
MSGRTHX     DC      CL1' '           'X' X FOR GETHOSTID
MSGRTCS     DC      CL1' '           '-' (NEGATIVE SIGN)
HEXRC       EQU      MSGRTCS         HEX RC WILL START AT SIGN LOCATION
MSGRTCV     DC      CL7' '           RETURNED VALUE (RETCODE)
MSGERRT     DC      CL10' ERRNO='    ' ERRNO='
MSGERRV     DC      CL7' '           RETURNED VALUE (ERRNO)
DWORK       DC      D'0'             WORK AREA
HEXTAB      EQU      *-240           TAB TO CONVERT TO PRINTABLE HEX
*          *              FIRST 240 BYTES NOT REFERENCED
          DC      CL16'0123456789ABCDEF'
PARMADDR    DC      A(0)             PARM ADDRESS SAVE AREA
          EZBREHST DSECT=NO,LIST=YES,HOSTENT=NO,ADRINFO=YES
          LTORG
*-----*
* REG/SAVEAREA *
*-----*
SOCSAVE     DC      9D'0'             SAVE AREA
*-----*
          CNOP      0,8
MYCB        EQU      *              MY CONTROL BLOCK
REQAREA     EQU      *
ECB         DC      A(ECB)           SELF POINTER
          DC      CL100'WORK AREA'
MYTIE       EZASMI  TYPE=TASK,STORAGE=CSECT  TIE
TYPE        DC      CL8'TYPE'
ERRNO       DC      F'0'
RETCODE     DC      F'0'
*
HOSTNAME    DS      CL255             HOST NAME FOR GETNAMEINFO
SERVNAME    DS      CL32             SERVICE NAME FOR GETNAMEINFO
          CNOP      0,4
HOSTNAMEL   DC      AL4(L'HOSTNAME)  LENGTH OF HOST NAME
SERVNAMEL   DC      AL4(L'SERVNAME)  LENGTH OF SERVICE NAME
NAMELEN     DC      AL4(L'NAME)      LENGTH OF NAME
FLAGS       DC      F'0'            GETNAMEINFO FLAGS
*
MYNEXT      DC      A(MYCB)           NEXT IN CHAIN FOR MULTIPLES
          CNOP      0,8
MYLEN       EQU      *-MYCB
MYCB2       EQU      *
          ORG      **MYLEN
          CNOP      0,8
          DC      CL8'&SYSDATE'
          DC      CL8'&SYSTIME'
          END

```

Figure 70. EZASO6AC sample client program for IPv6

Chapter 13. CALL instruction application programming interface

This information describes the CALL instruction API for IPv4 or IPv6 socket applications. The following topics are included:

- [“CALL instruction API environmental restrictions and programming requirements” on page 389](#)
- [“CALL instruction API output register information” on page 390](#)
- [“CALL instruction API compatibility considerations” on page 390](#)
- [“CALL instruction application programming interface \(API\)” on page 391](#)
- [“Understanding COBOL, Assembler, and PL/I call formats” on page 391](#)
- [“Converting parameter descriptions” on page 392](#)
- [“Diagnosing problems in applications using the CALL instruction API” on page 392](#)
- [“CALL instruction API error messages and return codes” on page 393](#)
- [“Code CALL instructions” on page 393](#)
- [“Using data translation programs for socket call interface” on page 522](#)
- [“Call interface sample programs” on page 534](#)

CALL instruction API environmental restrictions and programming requirements

The following restrictions apply to both the Macro Socket API and the Callable Socket API:

Function	Restriction
SRB mode	These APIs can only be invoked in TCB mode (task mode).
Cross-memory mode	These APIs can only be invoked in a non-cross-memory environment (PASN=SASN=HASN).
Functional Recovery Routine (FRR)	Do not invoke these APIs with an FRR set. This causes system recovery routines to be bypassed and severely damage the system.
Locks	No locks should be held when issuing these calls.
INITAPI and TERMAPI socket commands	The INITAPI and TERMAPI socket commands must be issued under the same task.
Storage	Storage acquired for the purpose of containing data returned from a socket call must be obtained in the same key as the application program status word (PSW) at the time of the socket call.
Nested socket API calls	You cannot issue nested API calls within the same task. That is, if a request block (RB) issues a socket API call and is interrupted by an interrupt request block (IRB) in an STIMER exit, any additional socket API calls that the IRB attempts to issue are detected and flagged as errors.

Function	Restriction
Addressability mode (Amode) considerations	The EZASOCKET API can be invoked while the caller is in either 31-bit or 24-bit Amode. However, if the application is running in 24-bit addressability mode at the time of the call, all addresses of parameters passed by the application must be addressable in 31-bit Amode. This implies that even if the addresses being passed reside in storage below the 16 MB line (and therefore addressable by 24-bit Amode programs) the high-order byte of these addresses needs to be 0.
Use of z/OS UNIX System Services	Address spaces using the EZASOCKET API should not use any z/OS UNIX System Services socket API facilities such as z/OS UNIX Assembler Callable Services or Language Environment for z/OS C/C++. Doing so can yield unpredictable results.

CALL instruction API output register information

When control returns to the caller, the general purpose registers (GPRs) contain:

Register Contents

0-1

Used as work registers by the system

2-13

Unchanged

14

Used as a work register by the system

15

Contains the entry point address EZASOCKET

When control returns to the caller, the access registers (ARs) contain:

Register Contents

0-1

Used as work registers by the system

2-14

Unchanged

15

Used as a work register by the system.

If a caller depends on register contents to remain the same before and after issuing a service, the caller must save the contents of a register before issuing the service and must restore them after the system returns control.

CALL instruction API compatibility considerations

Unless noted in [z/OS Communications Server: New Function Summary](#), an application program compiled and link edited on a release of z/OS Communications Server IP can be used on higher level releases. That is, the API is upward compatible.

Application programs that are compiled and link edited on a release of z/OS Communications Server IP cannot be used on older releases. That is, the API is not downward compatible.

CALL instruction application programming interface (API)

This information describes the CALL instruction API for TCP/IP application programs written in the COBOL, PL/I, or System/370 Assembler language. The format and parameters are described for each socket call.

Notes:

- Unless your program is running in a CICS environment, reentrant code and multithread applications are not directly supported by this interface. Each task or thread needs to invoke its own copy of EZASOKET.
- Modules that use EZASOKET must be non-reusable (NREUS) or serially reusable (REUS). If the module can be invoked again after its original task ends, the application needs to perform the TERMAPI call at the end of its function. Otherwise, unpredictable results are produced.
- For a PL/I program, include the following statement before your first call instruction.

```
DCL EZASOKET ENTRY OPTIONS(ASM,INTER) EXT;
```

- If you use the CALL instruction from code that will run as a part of a CICS transaction, see the [z/OS Communications Server: IP CICS Sockets Guide](#) for additional considerations.
- The Sockets Extended module (EZASOKET) is located in the hlq.SEZATCP(EZASOKET) load module and should be resolved from there when it is processed by the binder. You can use the linkage editor MAP parameter to produce the module map report to verify where EZASOKET is resolved.

Understanding COBOL, Assembler, and PL/I call formats

This API is invoked by calling the EZASOKET program and performs the same functions as the C language calls. The parameters look different because of the differences in the programming languages.

COBOL language call format

The following syntax shows the EZASOKET call format for COBOL language programs:

➤ CALL 'EZASOKET' USING SOC-FUNCTION — *parm1, parm2, ..* — ERRNO,RETCODE. ➤

SOC-FUNCTION

A 16-byte character field, left-justified and padded on the right with blanks. Set to the name of the call. SOC-FUNCTION is case specific. It must be in uppercase.

parm*n*

A variable number of parameters depending on the type call.

ERRNO

If RETCODE is negative, there is an error number in ERRNO. This field is used in most, but not all, of the calls. It corresponds to the value returned by the `tcperror()` function in C.

RETCODE

A fullword binary variable containing a code returned by the EZASOKET call. This value corresponds to the normal return value of a C function.

Assembler language call format

The following syntax shows the EZASOKET call format for assembler language programs.

➤ CALL EZASOKET,(SOC-FUNCTION, — *parm1, parm2, ..* — ERRNO,RETCODE),VL ➤

PL/I language call format

The following syntax shows the EZASOKET call format for PL/I language programs:

➡ CALL EZASOKET (SOC-FUNCTION — *parm1, parm2, ...* — ERRNO, RETCODE); ➡

SOC-FUNCTION

A 16-byte character field, left-justified and padded on the right with blanks. Set to the name of the call.

parm*n*

A variable number of parameters depending on the type call.

ERRNO

If RETCODE is negative, there is an error number in ERRNO. This field is used in most, but not all, of the calls. It corresponds to the value returned by the `tcperror()` function in C.

RETCODE

A fullword binary variable containing a code returned by the EZASOKET call. This value corresponds to the normal return value of a C function.

Converting parameter descriptions

The parameter descriptions in this information are written using the VS COBOL II PIC language syntax and conventions, but you should use the syntax and conventions that are appropriate for the language you want to use.

Figure 71 on page 392 shows examples of storage definition statements for COBOL, PL/I, and assembler languages.

VS COBOL II PIC

PIC S9(4) BINARY	HALFWORD BINARY VALUE
PIC S9(8) BINARY	FULLWORD BINARY VALUE
PIC X(n)	CHARACTER FIELD OF N BYTES

COBOL PIC

PIC S9(4) COMP	HALFWORD BINARY VALUE
PIC S9(4) BINARY	HALFWORD BINARY VALUE
PIC S9(8) COMP	FULLWORD BINARY VALUE
PIC S9(8) BINARY	FULLWORD BINARY VALUE
PIC X(n)	CHARACTER FIELD OF N BYTES

PL/I DECLARE STATEMENT

DCL HALF	FIXED BIN(15),	HALFWORD BINARY VALUE
DCL FULL	FIXED BIN(31),	FULLWORD BINARY VALUE
DCL CHARACTER	CHAR(n)	CHARACTER FIELD OF n BYTES

ASSEMBLER DECLARATION

DS H	HALFWORD BINARY VALUE
DS F	FULLWORD BINARY VALUE
DS CLn	CHARACTER FIELD OF n BYTES

Figure 71. Storage definition statement examples

Diagnosing problems in applications using the CALL instruction API

TCP/IP provides a trace facility that can be helpful in diagnosing problems in applications using the CALL instruction API. The trace is implemented using the TCP/IP Component Trace (CTRACE) SOCKAPI trace option. The SOCKAPI trace option allows all Call instruction socket API calls issued by an application to be traced in the TCP/IP CTRACE. The SOCKAPI trace records include information such as the type of socket call, input, and output parameters and return codes. This trace can be helpful in isolating failing socket API calls and in determining the nature of the error or the history of socket API calls that may be the cause of an error. For more information about the SOCKAPI trace option, see [z/OS Communications Server: IP Diagnosis Guide](#).

CALL instruction API error messages and return codes

For information about error messages, see [z/OS Communications Server: IP Messages Volume 1 \(EZA\)](#).

For information about error codes that are returned by TCP/IP, see [Appendix B, “Socket call error return codes,”](#) on page 743.

Code CALL instructions

This information contains the description, syntax, parameters , and other related information for each call instruction included in this API.

ACCEPT

A server issues the ACCEPT call to accept a connection request from a client. The call points to a socket that was previously created with a SOCKET call and marked by a LISTEN call.

The ACCEPT call is a blocking call. 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 s, and returns its descriptor in RETCODE. The original sockets remain available to the calling program to accept more connection requests.
3. The address of the client is returned in NAME for use by subsequent server calls.

Notes:

1. The blocking or nonblocking mode of a socket affects the operation of certain commands. The default is blocking; nonblocking mode can be established by use of the FCNTL and IOCTL calls. When a socket is in blocking mode, an I/O call waits for the completion of certain events. For example, a READ call will block until the buffer contains input data. When an I/O call is issued:
 - If the socket is blocking, program processing is suspended until the event completes.
 - If the socket is nonblocking, program processing continues.
2. If the queue has no pending connection requests, ACCEPT blocks the socket unless the socket is in nonblocking mode. The socket can be set to nonblocking by calling FCNTL or IOCTL.
3. When multiple socket calls are issued, a SELECT call can be issued prior to the ACCEPT to ensure that a connection request is pending. Using this technique ensures that subsequent ACCEPT calls will not block.
4. TCP/IP does not provide a function for screening clients. As a result, it is up to the application program to control which connection requests it accepts, but it can close a connection immediately after discovering the identity of the client.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.

Control parameters:	All parameters must be addressable by the caller and in the primary address space.
---------------------	--

Figure 72 on page 394 shows an example of ACCEPT call instructions.

```

WORKING-STORAGE SECTION.
  01 SOC-FUNCTION    PIC X(16) VALUE IS 'ACCEPT'.
  01 S               PIC 9(4) BINARY.
  * IPv4 socket address structure.
  01 NAME.
    03 FAMILY       PIC 9(4) BINARY.
    03 PORT         PIC 9(4) BINARY.
    03 IP-ADDRESS   PIC 9(8) BINARY.
    03 RESERVED     PIC X(8).
  * IPv6 socket address structure.
  01 NAME.
    03 FAMILY       PIC 9(4) BINARY.
    03 PORT         PIC 9(4) BINARY.
    03 FLOWINFO     PIC 9(8) BINARY.
    03 IP-ADDRESS.
      10 FILLER     PIC 9(16) BINARY.
      10 FILLER     PIC 9(16) BINARY.
    03 SCOPE-ID     PIC X(8) BINARY.
  01 ERRNO          PIC 9(8) BINARY.
  01 RETCODE        PIC S9(8) BINARY.

PROCEDURE DIVISION.
  CALL 'EZASOKET' USING SOC-FUNCTION S NAME ERRNO RETCODE.

```

Figure 72. ACCEPT call instructions example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions”](#) on page 392.

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing ACCEPT. Left-justify the field and pad it on the right with blanks.

S

A halfword binary number specifying the descriptor of a socket that was previously created with a SOCKET call. In a concurrent server, this is the socket upon which the server listens.

Parameter values returned to the application

NAME

An IPv4 socket address structure that contains the client’s socket address.

FAMILY

A halfword binary field specifying the IPv4 addressing family. The call returns the value decimal 2 for AF_INET.

PORT

A halfword binary field that is set to the client’s port number.

IP-ADDRESS

A fullword binary field that is set to the 32-bit IPv4 IP address, in network byte order, of the client’s host machine.

RESERVED

Specifies 8 bytes of binary zeros. This field is required, but not used.

An IPv6 socket address structure that contains the client’s socket address.

FAMILY

A halfword binary field specifying the IPv6 addressing family. For TCP/IP the value is decimal 19, indicating AF_INET6.

PORT

A halfword binary field that is set to the client’s port number.

FLOWINFO

A fullword binary field specifying the traffic class and flow label. This value of this field is undefined.

IP-ADDRESS

A 16-byte binary field that is set to the 128-bit IPv6 IP address, in network-byte-order, of the client's host machine.

SCOPE-ID

A fullword binary field which identifies a set of interfaces as appropriate for the scope of the address carried in the IPv6-ADDRESS field. For a link scope IPv6-ADDRESS, SCOPE-ID contains the link index for the IPv6-ADDRESS. For all other address scopes, SCOPE-ID is undefined.

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See [Appendix B, "Socket call error return codes,"](#) on page 743 for information about ERRNO return codes.

RETCODE

If the RETCODE value is positive, the RETCODE value is the new socket number.

If the RETCODE value is negative, check the ERRNO field for an error number.

Value**Description**

> 0

Successful call.

-1

Check **ERRNO** for an error code.

BIND

In a typical server program, the BIND call follows a SOCKET call and completes the process of creating a new socket.

The BIND socket command can 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 the socket address to use when issuing a CONNECT, SENDTO, or SENDMSG request.

In addition to the port, the application also specifies an IP address on the BIND socket command. Most applications typically specify a value of 0 for the IP address, which allows these applications to accept new TCP connections or receive UDP datagrams that arrive over any of the network interfaces of the local host. This enables client applications to contact the application using any of the IP addresses associated with the local host.

Alternatively, an application can indicate that it is interested in receiving new TCP connections or UDP datagrams that are targeted towards only a specific IP address associated with the local host. This can be accomplished by specifying the IP address in the appropriate field of the socket address structure passed on the NAME parameter.

Tip: Even if an application specifies the value 0 for the IP address on the BIND, the system administrator can override that value by specifying the BIND parameter on the PORT reservation statement in the TCP/IP profile. The effect of this override is similar to the effect of the application specifying an explicit IP address on the BIND macro. For more information, see [z/OS Communications Server: IP Configuration Reference](#).

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.

Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 73 on page 396 shows an example of BIND call instructions.

```

WORKING-STORAGE SECTION.
  01 SOC-FUNCTION    PIC X(16)  VALUE IS 'BIND'.
  01 S               PIC 9(4)  BINARY.

  * IPv4 socket address structure.
  01 NAME.
    03 FAMILY       PIC 9(4)  BINARY.
    03 PORT         PIC 9(4)  BINARY.
    03 IP-ADDRESS   PIC 9(8)  BINARY.
    03 RESERVED     PIC X(8).

  * IPv6 socket address structure.
  01 NAME.
    03 FAMILY       PIC 9(4)  BINARY.
    03 PORT         PIC 9(4)  BINARY.
    03 FLOWINFO     PIC 9(8)  BINARY.
    03 IP-ADDRESS.
      10 FILLER     PIC 9(16) BINARY.
      10 FILLER     PIC 9(16) BINARY.
    03 SCOPE-ID     PIC 9(8)  BINARY.

  01 ERRNO          PIC 9(8)  BINARY.
  01 RETCODE        PIC S9(8) BINARY.

PROCEDURE DIVISION.
  CALL 'EZASOKET' USING SOC-FUNCTION S NAME ERRNO RETCODE.

```

Figure 73. BIND call instruction example

For equivalent PL/I and assembler language declarations, see ["Converting parameter descriptions"](#) on page 392.

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing BIND. The field is left-justified and padded to the right with blanks.

S

A halfword binary number specifying the socket descriptor for the socket to be bound.

NAME

See [Chapter 3, "Designing an iterative server program,"](#) on page 23 for more information.

Specifies the IPv4 socket address structure for the socket that is to be bound.

FAMILY

A halfword binary field specifying the IPv4 addressing family. The value is always set to decimal 2, indicating AF_INET.

PORT

A halfword binary field that is set to the port number to which you want the socket to be bound.

Note: To determine the assigned port number, call the GETSOCKNAME command after calling the BIND command.

IP-ADDRESS

A fullword binary field that is set to the 32-bit IPv4 IP address (network byte order) of the socket to be bound.

RESERVED

Specifies an 8-byte character field that is required but not used.

Specifies the IPv6 socket address structure for the socket that is to be bound.

FAMILY

A halfword binary field specifying the IPv6 addressing family. For TCP/IP the value is decimal 19, indicating AF_INET6.

PORT

A halfword binary field that is set to the port number to which you want the socket to be bound.

Note: To determine the assigned port number, call the GETSOCKNAME command after calling the BIND command.

FLOWINFO

A fullword binary field specifying the traffic class and flow label. This field must be set to 0.

IP-ADDRESS

A 16-byte binary field that is set to the 128-bit IPv6 IP address (network byte order) of the socket to be bound.

SCOPE-ID

A fullword binary field which identifies a set of interfaces as appropriate for the scope of the address carried in the IPv6-ADDRESS field. A value of 0 indicates the SCOPE-ID field does not identify the set of interfaces to be used, and may be specified for any address types and scopes. For a link scope IPv6-ADDRESS, SCOPE-ID may specify a link index which identifies a set of interfaces. For all other address scopes, SCOPE-ID must be set to 0.

Parameter values returned to the application

ERRNO

A fullword binary field. If RETCODE is negative, this field contains an error number. See [Appendix B, “Socket call error return codes,”](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check **ERRNO** for an error code.

BIND2ADDRSEL

The BIND2ADDRSEL call binds a socket to the local IP address that would be selected by the stack to communicate with the input destination IP address.

Use the BIND2ADDRSEL call when the application must verify that the local IP address assigned by the stack meets its address selection criteria as specified by the IPV6_ADDR_PREFERENCES socket option before the stack sends any packets to the remote host. In a TCP or UDP application, the BIND2ADDRSEL call usually follows the SETSOCKOPT call with option IPV6_ADDR_PREFERENCES and precedes any communication with a remote host.

Result: The stack attempts to select a local IP address according to your application preferences. However, a successful BIND2ADDRSEL call does not guarantee that all of your source IP address selection preferences were met.

Guidelines:

- Use the SETSOCKOPT call to set the IPV6_ADDR_PREFERENCES option to indicate your selection preferences of source IP address before binding the socket and before allowing an implicit bind of the socket to occur.

Result: If a socket has not been explicitly bound to a local IP address with a BIND or BIND2ADDRSEL call when a CONNECT, SENDTO, or SENDMSG call is issued, an implicit bind occurs. The stack chooses the local IP address used for outbound packets.

Requirement: When your application is using stream sockets, and must prevent the stack from sending any packets whatsoever (such as SYN) to the remote host before it can verify that the local IP address meets the values specified for the IPV6_ADDR_PREFERENCES option, do not allow the CONNECT call to implicitly bind the socket to a local IP address. Instead, bind the socket with the BIND2ADDRSEL call and test the local IP address assigned with the INET6_IS_SRCADDR call. If the assigned local IP address is satisfactory, you can then use the CONNECT call to establish communication with the remote host.

- After you successfully issue the BIND2ADDRSEL call, use the GETSOCKNAME call to obtain the local IP address that is bound to the socket. When the local IP address is obtained, use the INET6_IS_SRCADDR call to verify that the local IP address meets your address selection criteria.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

[Figure 74 on page 399](#) shows an example of BIND2ADDRSEL call instructions.

```

WORKING-STORAGE SECTION.
    01 SOC-FUNCTION    PIC X(16) VALUE IS 'BIND2ADDRSEL'.
    01 S              PIC 9(4) BINARY.
    * IPv6 socket address structure.
    01 NAME.
        03 FAMILY      PIC 9(4) BINARY.
        03 PORT        PIC 9(4) BINARY.
        03 FLOWINFO    PIC 9(8) BINARY.
        03 IP-ADDRESS.
            10 FILLER   PIC 9(16) BINARY.
            10 FILLER   PIC 9(16) BINARY.
        03 SCOPE-ID    PIC 9(8) BINARY.
    01 ERRNO          PIC 9(8) BINARY.
    01 RETCODE        PIC S9(8) BINARY.

PROCEDURE DIVISION.
    CALL 'EZASOKET' USING SOC-FUNCTION S NAME ERRNO RETCODE.

```

Figure 74. BIND2ADDRSEL call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions” on page 392](#).

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing BIND2ADDRSEL. The field is left-justified and padded to the right with blanks.

S

A halfword binary number specifying the socket descriptor for the socket that is to be bound.

Requirement: The socket must be an AF_INET6 socket. The type can be SOCK_STREAM or SOCK_DGRAM.

NAME

Specifies the IPv6 socket address structure of the remote host that the socket will communicate with. The IPv6 socket structure must specify the following fields:

FAMILY

A halfword binary field specifying the IPv6 addressing family. This field must be set to the decimal value 19, indicating AF_INET6.

PORT

A halfword binary field. This field is ignored by BIND2ADDRSEL processing.

Tip: To determine the assigned port number, issue the GETSOCKNAME call after the BIND2ADDRSEL call completes.

FLOWINFO

A fullword binary field. This field is ignored by BIND2ADDRSEL processing.

IP-ADDRESS

A 16-byte binary field that is set to the 128-bit IPv6 IP address (network byte order) of the remote host that the socket will communicate with.

Rule: Specify an IPv4 address by using its IPv4-mapped IPv6 format.

SCOPE-ID

A fullword binary field that identifies a set of appropriate interfaces for the scope of the address that is specified in the IPv6-ADDRESS field. The value 0 indicates that the SCOPE-ID field does not identify the set of interfaces to be used.

Requirement: The SCOPE-ID value must be nonzero if the address is a link-local address. For all other address scopes, SCOPE-ID must be set to 0.

Parameter values returned to the application

ERRNO

A fullword binary field. If RETCODE is negative, this field contains an error number. See [Appendix B, “Socket call error return codes,”](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value	Description
-------	-------------

0	Successful call.
-1	Check ERRNO for an error code.

CLOSE

The CLOSE call performs the following functions:

- The CLOSE call shuts down a socket and frees all resources allocated to it. If the socket refers to an open TCP connection, the connection is closed.
- The CLOSE call is also issued by a concurrent server after it gives a socket to a child server program. After issuing the GIVESOCKET and receiving notification that the client child has successfully issued a TAKESOCKET, the concurrent server issues the close command to complete the passing of ownership. In high-performance, transaction-based systems the timeout associated with the CLOSE call can cause performance problems. In such systems you should consider the use of a SHUTDOWN call before you issue the CLOSE call. See [“SHUTDOWN”](#) on page 513 for more information.

Notes:

1. If a stream socket is closed while input or output data is queued, the TCP connection is reset and data transmission may be incomplete. The SETSOCKOPT call can be used to set a *linger* condition, in which TCP/IP will continue to attempt to complete data transmission for a specified period of time after the CLOSE call is issued. See SO-LINGER in the description of [“SETSOCKOPT”](#) on page 497.
2. A concurrent server differs from an iterative server. An iterative server provides services for one client at a time; a concurrent server receives connection requests from multiple clients and creates child servers that actually serve the clients. When a child server is created, the concurrent server obtains a new socket, passes the new socket to the child server, and then dissociates itself from the connection. The CICS Listener is an example of a concurrent server.
3. After an unsuccessful socket call, a close should be issued and a new socket should be opened. An attempt to use the same socket with another call results in a nonzero return code.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.

Control parameters:	All parameters must be addressable by the caller and in the primary address space.
---------------------	--

Figure 75 on page 401 shows an example of CLOSE call instructions.

```
WORKING-STORAGE SECTION.
    01 SOC-FUNCTION PIC X(16) VALUE IS 'CLOSE'.
    01 S PIC 9(4) BINARY.
    01 ERRNO PIC 9(8) BINARY.
    01 RETCODE PIC S9(8) BINARY.

PROCEDURE DIVISION.
    CALL 'EZASOKET' USING SOC-FUNCTION S ERRNO RETCODE.
```

Figure 75. CLOSE call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions” on page 392](#).

Parameter values set by the application

SOC-FUNCTION

A 16-byte field containing CLOSE. Left-justify the field and pad it on the right with blanks.

S

A halfword binary field containing the descriptor of the socket to be closed.

Parameter values returned to the application

ERRNO

A fullword binary field. If RETCODE is negative, this field contains an error number. See [Appendix B, “Socket call error return codes,” on page 743](#) for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check **ERRNO** for an error code.

CONNECT

The CONNECT call is issued by a client to establish a connection between a local socket and a remote socket.

Stream sockets

For stream sockets, the CONNECT call is issued by a client to establish connection with a server. The call performs two tasks:

- It completes the binding process for a stream socket if a BIND call has not been previously issued.
- It attempts to make a connection to a remote socket. This connection is necessary before data can be transferred.

UDP sockets

For UDP sockets, a CONNECT call need not precede an I/O call, but if issued, it allows you to send messages without specifying the destination.

The call sequence issued by the client and server for stream sockets is:

1. The *server* issues BIND and LISTEN to create a passive open socket.
2. The *client* issues CONNECT to request the connection.
3. The *server* accepts the connection on the passive open socket, creating a new connected socket.

The blocking mode of the CONNECT call conditions its operation.

- If the socket is in blocking mode, the CONNECT call blocks the calling program until the connection is established, or until an error is received.
- If the socket is in nonblocking mode, the return code indicates whether the connection request was successful.
 - A 0 RETCODE indicates that the connection was completed.
 - A nonzero RETCODE with an ERRNO of 36 (EINPROGRESS) indicates that the connection is not completed, but since the socket is nonblocking, the CONNECT call returns normally.

The caller must test the completion of the connection setup by calling SELECT and testing for the ability to write to the socket.

The completion cannot be checked by issuing a second CONNECT. For more information, see [“SELECT” on page 481](#).

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389 .
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

[Figure 76 on page 403](#) shows an example of CONNECT call instructions.

```

WORKING-STORAGE SECTION.
  01 SOC-FUNCTION PIC X(16) VALUE IS 'CONNECT'.
  01 S PIC 9(4) BINARY.

  * IPv4 socket address structure.
  01 NAME.
    03 FAMILY PIC 9(4) BINARY.
    03 PORT PIC 9(4) BINARY.
    03 IP-ADDRESS PIC 9(8) BINARY.
    03 RESERVED PIC X(8).

  * IPv6 socket address structure.
  01 NAME.
    03 FAMILY PIC 9(4) BINARY.
    03 PORT PIC 9(4) BINARY.
    03 IP-ADDRESS PIC 9(8) BINARY.
    03 FLOWINFO PIC 9(8) BINARY.
    03 IP-ADDRESS.
      10 FILLER PIC 9(16) BINARY.
      10 FILLER PIC 9(16) BINARY.
    03 SCOPE-ID PIC 9(8) BINARY.
  01 ERRNO PIC 9(8) BINARY.
  01 RETCODE PIC S9(8) BINARY.

PROCEDURE DIVISION.

  CALL 'EZASOKET' USING SOC-FUNCTION S NAME ERRNO RETCODE.

```

Figure 76. CONNECT call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions” on page 392](#).

Parameter values set by the application

SOC-FUNCTION

A 16-byte field containing CONNECT. Left-justify the field and pad it on the right with blanks.

S

A halfword binary number specifying the socket descriptor of the socket that is to be used to establish a connection.

NAME

An IPv4 socket address structure that contains the IPv4 socket address of the target to which the local, client socket is to be connected.

FAMILY

A halfword binary field specifying the IPv4 addressing family. The value must be decimal 2 for AF_INET.

PORT

A halfword binary field that is set to the server’s port number in network byte order. For example, if the port number is 5000 in decimal, it is stored as X'1388' in hex.

IP-ADDRESS

A fullword binary field that is set to the 32-bit IPv4 IP address of the server’s host machine in network byte order. For example, if the IP address is 129.4.5.12 in dotted decimal notation, it would be represented as X'8104050C' in hex.

RESERVED

Specifies an 8-byte reserved field. This field is required, but is not used.

An IPv6 socket address structure that contains the IPv6 socket address of the target to which the local, client socket is to be connected.

FAMILY

A halfword binary field specifying the IPv6 addressing family. For TCP/IP the value is decimal 19 for AF_INET6.

PORT

A halfword binary field that is set to the server's port number in network byte order. For example, if the port number is 5000 in decimal, it is stored as X'1388' in hex.

FLOWINFO

A fullword binary field specifying the traffic class and flow label. This field must be set to 0.

IP-ADDRESS

A 16-byte binary field that is set to the 128-bit IPv6 IP address of the server's host machine in network byte order. For example, if the IPv6 IP address is 12ab:0:0:cd30:123:4567:89ab:cedf in colon hex notation, it is set to X'12AB00000000CD300123456789ABCDEF'.

SCOPE-ID

A fullword binary field which identifies a set of interfaces as appropriate for the scope of the address carried in the IPv6-ADDRESS field. A value of 0 indicates the SCOPE-ID field does not identify the set of interfaces to be used, and may be specified for any address types and scopes. For a link scope IPv6-ADDRESS, SCOPE-ID may specify a link index which identifies a set of interfaces. For all other address scopes, SCOPE-ID must be set to 0.

Parameter values returned to the application

ERRNO

A fullword binary field. If RETCODE is negative, this field contains an error number. See [Appendix B, "Socket call error return codes,"](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value**Description****0**

Successful call.

-1Check **ERRNO** for an error code.

FCNTL

The blocking mode of a socket can either be queried or set to nonblocking using the FNDELAY flag described in the FCNTL call. You can query or set the FNDELAY flag even though it is not defined in your program.

See ["IOCTL" on page 455](#) for another way to control a socket's blocking mode.

Values for commands that are supported by the z/OS UNIX Systems Services fcntl callable service will also be accepted. See [z/OS UNIX System Services Programming: Assembler Callable Services Reference](#) for more information.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.

Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 77 on page 405 shows an example of FCNTL call instructions.

```

WORKING-STORAGE SECTION
01  SOC-FUNCTION    PIC X(16)  VALUE IS 'FCNTL'.
01  S               PIC 9(4)   BINARY.
01  COMMAND        PIC 9(8)   BINARY.
01  REQARG         PIC 9(8)   BINARY.
01  ERRNO          PIC 9(8)   BINARY.
01  RETCODE        PIC S9(8)  BINARY.

PROCEDURE DIVISION
CALL 'EZASOKET' USING SOC-FUNCTION S COMMAND REQARG
ERRNO RETCODE.

```

Figure 77. FCNTL call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions” on page 392](#).

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing FCNTL. The field is left-justified and padded on the right with blanks.

S

A halfword binary number specifying the socket descriptor for the socket that you want to unblock or query.

COMMAND

A fullword binary number with the following values:

Value

Description

3

Query the blocking mode of the socket.

4

Set the mode to blocking or nonblocking for the socket.

REQARG

A fullword binary field containing a mask that TCP/IP uses to set the FNDELAY flag.

- If COMMAND is set to 3 ('query') the REQARG field should be set to 0.
- If COMMAND is set to 4 ('set')
 - Set REQARG to 4 to turn the FNDELAY flag on. This places the socket in nonblocking mode.
 - Set REQARG to 0 to turn the FNDELAY flag off. This places the socket in blocking mode.

Parameter values returned to the application

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See [Appendix B, “Socket call error return codes,” on page 743](#) for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

- If COMMAND was set to 3 (query), a bit string is returned.

- If RETCODE contains X'00000004', the socket is nonblocking. (The FNDELAY flag is on.)
- If RETCODE contains X'00000000', the socket is blocking. (The FNDELAY flag is off.)
- If COMMAND was set to 4 (set), a successful call is indicated by 0 in this field. In both cases, a RETCODE of -1 indicates an error (check the ERRNO field for the error number).

FREEADDRINFO

The FREEADDRINFO call frees all the address information structures returned by GETADDRINFO in the RES parameter.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 78 on page 406 shows an example of FREEADDRINFO call instructions.

```

WORKING-STORAGE SECTION.
  01  SOC-FUNCTION    PIC X(16)  VALUE IS 'FREEADDRINFO'.
  01  ADDRINFO       PIC 9(8)  BINARY.
  01  ERRNO          PIC 9(8)  BINARY.
  01  RETCODE        PIC S9(8)  BINARY.

PROCEDURE DIVISION.
  CALL 'EZASOKET' USING SOC-FUNCTION ADDRINFO
                      ERRNO RETCODE.

```

Figure 78. FREEADDRINFO call instruction example

Parameter values set by the application

Keyword

Description

SOC-FUNCTION

A 16-byte character field containing FREEADDRINFO. The field is left-justified and padded on the right with blanks.

ADDRINFO

Input parameter. The address of a set of address information structures returned by the GETADDRINFO RES argument.

Parameter values returned to the application

Keyword

Description

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, **ERRNO** contains a valid error number. Otherwise, ignore the **ERRNO** field.

See Appendix B, “Socket call error return codes,” on page 743 for information about **ERRNO** return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check **ERRNO** for an error code.

GETADDRINFO

The GETADDRINFO call translates either the name of a service location (for example, a host name), a service name, or both, and returns a set of socket addresses and associated information to be used in creating a socket with which to address the specified service or sending a datagram to the specified service.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

[Figure 79 on page 408](#) shows an example of GETADDRINFO call instructions.

```

WORKING-STORAGE SECTION.
01 SOC-FUNCTION PIC X(16) VALUE IS 'GETADDRINFO'.
01 NODE PIC X(255).
01 NODELEN PIC 9(8) BINARY.
01 SERVICE PIC X(32).
01 SERVLN PIC 9(8) BINARY.
01 AI-PASSIVE PIC 9(8) BINARY VALUE 1.
01 AI-CANONNAMEOK PIC 9(8) BINARY VALUE 2.
01 AI-NUMERICHOST PIC 9(8) BINARY VALUE 4.
01 AI-NUMERICSERV PIC 9(8) BINARY VALUE 8.
01 AI-V4MAPPED PIC 9(8) BINARY VALUE 16.
01 AI-ALL PIC 9(8) BINARY VALUE 32.
01 AI-ADDRCONFIG PIC 9(8) BINARY VALUE 64.
01 AI-EXTFLAGS PIC 9(8) BINARY VALUE 128.
01 HINTS USAGE IS POINTER.
01 RES USAGE IS POINTER.
01 CANNLEN PIC 9(8) BINARY.
01 ERRNO PIC 9(8) BINARY.
01 RETCODE PIC S9(8) BINARY.

LINKAGE SECTION.
01 HINTS-ADDRINFO.
03 FLAGS PIC 9(8) BINARY.
03 AF PIC 9(8) BINARY.
03 SOCTYPE PIC 9(8) BINARY.
03 PROTO PIC 9(8) BINARY.
03 FILLER PIC 9(8) BINARY.
03 FILLER PIC X(4).
03 FILLER PIC X(4).
03 FILLER PIC 9(8) BINARY.
03 FILLER PIC X(4).
03 FILLER PIC 9(8) BINARY.
03 FILLER PIC X(4).
03 FILLER PIC 9(8) BINARY.
03 EFLAGS PIC 9(8) BINARY.
01 RES-ADDRINFO.
03 FLAGS PIC 9(8) BINARY.
03 AF PIC 9(8) BINARY.
03 SOCTYPE PIC 9(8) BINARY.
03 PROTO PIC 9(8) BINARY.
03 NAMELEN PIC 9(8) BINARY.
03 FILLER PIC X(4).
03 FILLER PIC X(4).
03 CANONNAME USAGE IS POINTER.
03 FILLER PIC X(4).
03 NAME USAGE IS POINTER.
03 FILLER PIC X(4).
03 NEXT USAGE IS POINTER.
03 FILLER PIC 9(8) BINARY.

PROCEDURE DIVISION.
    MOVE 'www.hostname.com' TO NODE.
    MOVE 16 TO HOSTLEN.
    MOVE 'TELNET' TO SERVICE.
    MOVE 6 TO SERVLN.
    SET HINTS TO ADDRESS OF HINTS-ADDRINFO.
    CALL 'EZASOKET' USING SOC-FUNCTION NODE NODELEN SERVICE SERVLN HINTS
    RES CANNLEN ERRNO RETCODE.

```

Figure 79. GETADDRINFO call instruction example

Parameter values set by the application

Keyword

Description

SOC-FUNCTION

A 16-byte character field containing GETADDRINFO. The field is left-justified and padded on the right with blanks.

NODE

An input parameter. Storage up to 255 bytes long that contains the host name being queried. If the AI-NUMERICHOST flag is specified in the storage pointed to by the HINTS field, then NODE should contain the queried host's IP address in presentation form. This is an optional field but if specified you

must also code NODELEN. The NODE name being queried will consist of up to NODELEN or up to the first binary 0.

You can append scope information to the host name, using the format *node%scope information*. The combined information must be 255 bytes or less. For more information, see [z/OS Communications Server: IPv6 Network and Appl Design Guide](#).

NODELEN

An input parameter. A fullword binary field set to the length of the host name specified in the NODE field and should not include extraneous blanks. This is an optional field but if specified you must also code NODE.

SERVICE

An input parameter. Storage up to 32 bytes long that contains the service name being queried. If the AI-NUMERICSERV flag is specified in the storage pointed to by the HINTS field, then SERVICE should contain the queried port number in presentation form. This is an optional field but if specified you must also code SERLEN. The SERVICE name being queried will consist of up to SERLEN or up to the first binary 0.

SERLEN

An input parameter. A fullword binary field set to the length of the service name specified in the SERVICE field and should not include extraneous blanks. This is an optional field but if specified you must also code SERVICE.

HINTS

An input parameter. If the HINTS argument is specified, it contains the address of an addrinfo structure containing input values that may direct the operation by providing options and limiting the returned information to a specific socket type, address family, or protocol. If the HINTS argument is not specified, then the information returned will be as if it referred to a structure containing the value 0 for the FLAGS, SOCTYPE and PROTO fields, and AF_UNSPEC for the AF field. Include the EZBREHST resolver macro so that your assembler program will contain the assembler mappings for the ADDR_INFO structure. The EZBREHST assembler macro is stored in the SYS1.MACLIB library. The macro defines the resolver hostent (host entry), address information (addrinfo) mappings, and services return codes. Copy definitions from the EZACOBOL sample module to your COBOL program for mapping the ADDRINFO structure. The EZACOBOL sample module is stored in the *hlq*.SEZAINST library. Copy definitions from the CBLOCK sample module to your PL/I program for mapping the ADDRINFO structure. The CBLOCK sample module is stored in *hlq*.SEZAINST library.

This is an optional field.

The address information structure has the following fields:

Field

Description

FLAGS

A fullword binary field. Must have the value of 0 or the bitwise OR of one or more of the following values:

AI-PASSIVE (X'00000001') or the decimal value 1.

- Specifies how to specify the NAME pointed to by the returned RES.
- If this flag is specified, then the returned address information will be suitable for use in binding a socket for accepting incoming connections for the specified service (for example, the BIND call). In this case, if the NODE argument is not specified, then the IP address portion of the socket address structure pointed to by the returned RES will be set to INADDR_ANY for an IPv4 address or to the IPv6 unspecified address (in6addr_any) for an IPv6 address.
- If this flag is not set, the returned address information will be suitable for the CONNECT call (for a connection-mode protocol) or for a CONNECT, SENDTO, or SENDMSG call (for a connectionless protocol). In this case, if the NODE argument is not specified, then the IP address portion of the socket address structure pointed to by the returned RES will be set

to the default loopback address for an IPv4 address (127.0.0.1) or the default loopback address for an IPv6 address (::1).

- This flag is ignored if the NODE argument is specified.

AI-CANONNAMEOK (X'00000002') or the decimal value 2.

- If this flag is specified and the NODE argument is specified, then the GETADDRINFO call attempts to determine the canonical name corresponding to the NODE argument.

AI-NUMERICHOST (X'00000004') or the decimal value 4.

- If this flag is specified then the NODE argument must be a numeric host address in presentation form. Otherwise, an error of host not found [EAI_NONAME] is returned.

AI-NUMERICSERV (X'00000008') or the decimal value 8.

- If this flag is specified, the SERVICE argument must be a numeric port in presentation form. Otherwise, an error [EAI_NONAME] is returned.

AI-V4MAPPED (X'00000010') or the decimal value 16.

- If this flag is specified along with the AF field with the value of AF_INET6 or a value of AF_UNSPEC when IPv6 is supported, the caller will accept IPv4-mapped IPv6 addresses.
 - If the AF field is AF_INET6, a query for IPv4 addresses is made if the AI-ALL flag is specified or if no IPv6 addresses are found. Any IPv4 addresses that are found are returned as IPv4-mapped IPv6 addresses.
 - If the AF field is AF_UNSPEC, queries are made for both IPv6 and IPv4 addresses. If IPv4 addresses are found and IPv6 is supported, the IPv4 addresses are returned as IPv4-mapped IPv6 addresses.
- Otherwise, this flag is ignored.

AI-ALL (X'00000020') or the decimal value 32.

- When the AF field has a value of AF_INET6 and AI-ALL is set, the AI-V4MAPPED flag must also be set to indicate that the caller will accept all addresses: IPv6 and IPv4-mapped IPv6 addresses.
- If the AF field has a value of AF_UNSPEC, AI-ALL is accepted, but has no impact on the processing. No matter if AI-ALL is specified or not, the caller accepts both IPv6 and IPv4 addresses. A query is first made for IPv6 addresses and if successful, the IPv6 addresses are returned. Another query is then made for IPv4 addresses:
 - If the AI-V4MAPPED flag is also specified and the system supports IPv6, the IPv4 addresses are returned as IPv4-mapped IPv6 addresses.
 - If the AI-V4MAPPED flag is not specified or the system does not support IPv6, the IPv4 addresses are returned as IPv4 addresses.
- Otherwise, this flag is ignored.

AI-ADDRCONFIG (X'00000040') or the decimal value 64.

If this flag is specified, then a query on the name in NODE will occur if the Resolver determines whether either of the following conditions is true:

- If the system is IPv6 enabled and has at least one IPv6 interface, then the Resolver will make a query for IPv6 (AAAA or A6 DNS) records.
- If the system is IPv4 enabled and has at least one IPv4 interface, then the Resolver will make a query for IPv4 (A DNS) records.

The loopback address is not considered in this case as a valid interface.

AI-EXTFLAGS (X'00000080') or the decimal value 128.

Specifies this flag to request the extended form of the getaddrinfo function. The extended form allows additional hints to be passed to the resolver for determining the order of destination addresses that are returned. If this flag is specified, the EFLAGS field is required.

Tip: To perform the binary OR'ing of the flags above in a COBOL program, simply add the necessary COBOL statements as in the example below. Note that the value of the FLAGS field after the COBOL ADD is a decimal 80 or a X'00000050', which is the sum of OR'ing AI-V4MAPPED and AI-ADDRCONFIG or X'00000010' and X'00000040':

```
01 AI-V4MAPPED    PIC 9(8) BINARY VALUE 16.
01 AI-ADDRCONFIG  PIC 9(8) BINARY VALUE 64.

ADD AI-V4MAPPED TO FLAGS.
ADD AI-ADDRCONF  TO FLAGS.
```

AF

A fullword binary field. Used to limit the returned information to a specific address family. The value of AF_UNSPEC means that the caller will accept any protocol family. The value of a decimal 0 indicates AF_UNSPEC. The value of a decimal 2 indicates AF_INET, and the value of a decimal 19 indicates AF_INET6.

SOCTYPE

A fullword binary field. Used to limit the returned information to a specific socket type. A value of 0 means that the caller will accept any socket type. If a specific socket type is not given (for example, a value of 0) then information on all supported socket types will be returned.

The following table shows the acceptable socket types:

Type name	Decimal value	Description
SOCK_STREAM	1	for stream socket
SOCK_DGRAM	2	for datagram socket
SOCK_RAW	3	for raw-protocol interface

Anything else will fail with return code EAI_SOCTYPE. Note that although SOCK_RAW will be accepted, it will be valid only when SERVICE is numeric (for example, SERVICE=23). A lookup for a SERVICE name will never occur in the appropriate services file (for example, *hlq.ETC.SERVICES*) using any protocol value other than SOCK_STREAM or SOCK_DGRAM.

If PROTO is not 0 and SOCTYPE is 0, then the only acceptable input values for PROTO are IPPROTO_TCP and IPPROTO_UDP. Otherwise, the GETADDRINFO call will be failed with return code of EAI_BADFLAGS.

If SOCTYPE and PROTO are both specified as 0, then GETADDRINFO will proceed as follows:

- If SERVICE is null, or if SERVICE is numeric, then any returned addrinfos will default to a specification of SOCTYPE as SOCK_STREAM.
- If SERVICE is specified as a service name (for example, SERVICE=FTP), the GETADDRINFO call will search the appropriate services file (for example, *hlq.ETC.SERVICES*) twice. The first search will use SOCK_STREAM as the protocol, and the second search will use SOCK_DGRAM as the protocol. No default socket type is provided in this case.

If both SOCTYPE and PROTO are specified as nonzero, then they should be compatible, regardless of the value specified by SERVICE. In this context, *compatible* can have one of the following meanings:

- SOCTYPE=SOCK_STREAM and PROTO=IPPROTO_TCP
- SOCTYPE=SOCK_DGRAM and PROTO=IPPROTO_UDP
- SOCTYPE is specified as SOCK_RAW, in which case PROTO can be anything

PROTO

A fullword binary field. Used to limit the returned information to a specific protocol. A value of 0 means that the caller will accept any protocol.

The following table shows the acceptable protocols:

Protocol name	Decimal value	Description
IPPROTO_TCP	6	TCP
IPPROTO_UDP	17	user datagram

If SOCTYPE is 0 and PROTO is nonzero, the only acceptable input values for PROTO are IPPROTO_TCP and IPPROTO_UDP. Otherwise, the GETADDRINFO call will be failed with return code of EAI_BADFLAGS.

If PROTO and SOCTYPE are both specified as 0, then GETADDRINFO will proceed as follows:

- If SERVICE is null, or if SERVICE is numeric, then any returned addrinfos will default to a specification of SOCTYPE as SOCK_STREAM.
- If SERVICE is specified as a service name (for example, SERVICE=FTP), the GETADDRINFO will search the appropriate services file (for example, *hlq.ETC.SERVICE*) twice. The first search will use SOCK_STREAM as the protocol, and the second search will use SOCK_DGRAM as the protocol. No default socket type is provided in this case.

If both PROTO and SOCTYPE are specified as nonzero, they should be compatible, regardless of the value specified by SERVICE. In this context, *compatible* can have one of the following meanings:

- SOCTYPE=SOCK_STREAM and PROTO=IPPROTO_TCP
- SOCTYPE=SOCK_DGRAM and PROTO=IPPROTO_UDP
- SOCTYPE=SOCK_RAW, in which case PROTO can be anything

If the lookup for the value specified in SERVICE fails [for example, the service name does not appear in an appropriate service file (such as, *hlq.ETC.SERVICES*) using the input protocol], then the GETADDRINFO call will be failed with return code of EAI_SERVICE.

NAMELEN

A fullword binary field followed by 8 padding bytes. On input, this field must be 0.

CANONNAME

A fullword binary field followed by 4 padding bytes. On input, this field must be 0.

NAME

A fullword binary field followed by 4 padding bytes. On input, this field must be 0.

NEXT

A fullword binary field. On input, this field must be 0.

EFLAGS

A fullword binary field that specifies the source IPv6 address selection preferences. This field is required if the value AI-EXTFLAGS is specified in the FLAGS field.

This field must contain the value 0 or the bitwise OR of one or more of the following values:

IPv6_PREFER_SRC_HOME (X'00000001') or the decimal value 1

Indicates that home source IPv6 addresses are preferred over care-of source IPv6 addresses.

IPv6_PREFER_SRC_COA (X'00000002') or the decimal value 2

Indicates that care-of source IPv6 addresses are preferred over home source IPv6 addresses.

IPv6_PREFER_SRC_TMP (X'00000004') or the decimal value 4

Indicates that temporary source IPv6 addresses are preferred over public source IPv6 addresses.

IPv6_PREFER_SRC_PUBLIC (X'00000008') or the decimal value 8

Indicates that public source IPv6 addresses are preferred over temporary source IPv6 addresses.

IPv6_PREFER_SRC_CGA (X'00000010') or the decimal value 16

Indicates that cryptographically generated source IPv6 addresses are preferred over non-cryptographically generated source IPv6 addresses.

IPV6_PREFER_SRC_NONCGA (X'00000020') or the decimal value 32

Indicates that non-cryptographically generated source IPv6 addresses are preferred over cryptographically generated source IPv6 addresses.

Guidelines:

- If contradictory EFLAGS (for example, IPV6_PREFER_SRC_TMP and IPV6_PREFER_SRC_PUBLIC) or invalid EFLAGS (for example, X'00000040' or the decimal value 64) are specified, then the GETADDRINFO call fails with RETCODE -1 and ERRNO EAI_BADEXTFLAGS (decimal value 11).
- The COBOL constants for EFLAGS use hyphens instead of underscores.

RES

Initially a fullword binary field. On a successful return, this field contains a pointer to a chain of one or more address information structures. The structures are allocated in the key of the calling application. The structures that are returned on a GETADDRINFO call are serially reusable storage for the z/OS UNIX process. They can be used or referenced between process threads, but should not be used or referenced between processes. When you finish using the structures, explicitly release their storage by specifying the returned pointer on a FREEADDRINFO call. Include the EZBREHST resolver macro so that your assembler program contains the assembler mappings for the ADDR_INFO structure. The EZBREHST assembler macro is stored in the SYS1.MACLIB library. Copy definitions from the EZACOBOL sample module to your COBOL program for mapping the ADDRINFO structure. The EZACOBOL sample module is stored in the *hlq*.SEZAINST library. Copy definitions from the CBLOCK sample module to your PL/I program for mapping the ADDRINFO structure. The CBLOCK sample module is stored in the *hlq*.SEZAINST library.

The address information structure contains the following fields:

Field**Description****FLAGS**

A fullword binary field that is not used as output.

AF

A fullword binary field. The value returned in this field may be used as the AF argument on the SOCKET call to create a socket suitable for use with the returned address NAME.

SOCTYPE

A fullword binary field. The value returned in this field may be used as the SOCTYPE argument on the SOCKET call to create a socket suitable for use with the returned address NAME.

PROTO

A fullword binary field. The value returned in this field may be used as the PROTO argument on the SOCKET call to create a socket suitable for use with the returned address ADDR.

NAMELEN

A fullword binary field followed by 8 padding bytes. The length of the NAME socket address structure.

CANONNAME

A fullword binary field followed by 4 padding bytes. The canonical name for the value specified by NODE. If the NODE argument is specified, and if the AI-CANONNAMEOK flag was specified by the HINTS argument, then the CANONNAME field in the first returned address information structure will contain the address of storage containing the canonical name corresponding to the input NODE argument. If the canonical name is not available, then the CANONNAME field will refer to the NODE argument or a string with the same contents. The CANNLEN field contains the length of the returned canonical name.

NAME

A fullword binary field followed by 4 padding bytes. The address of the returned socket address structure. The value returned in this field can be used as the arguments for the CONNECT, BIND, or BIND2ADDRSEL call with such a socket, according to the AI-PASSIVE flag.

NEXT

A fullword binary field. Contains the address of the next address information structure on the list, or 0's if it is the last structure on the list.

EFLAGS

A fullword binary field that is not used as output.

CANNLEN

Initially an input parameter. A fullword binary field used to contain the length of the canonical name returned by the RES CANONNAME field. This is an optional field.

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, ERRNO contains a valid error number. Otherwise, ignore the ERRNO field.

See Appendix B, “Socket call error return codes,” on page 743 for information about ERRNO return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value**Description****0**

Successful call.

-1

Check ERRNO for an error code.

The ADDRINFO structure uses indirect addressing to return a variable number of NAMES. If you are coding in PL/I or assembler language, this structure can be processed in a relatively straight-forward manner. If you are coding in COBOL, this structure may be difficult to interpret. You can use the subroutine EZACIC09 to simplify interpretation of the information returned by the GETADDRINFO calls.

GETCLIENTID

GETCLIENTID call returns the identifier by which the calling application is known to the TCP/IP address space in the calling program. The CLIENT parameter is used in the GIVESOCKET and TAKESOCKET calls. See “GIVESOCKET” on page 448 for a discussion of the use of GIVESOCKET and TAKESOCKET calls.

Do not be confused by the terminology; when GETCLIENTID is called by a server, the identifier of the *caller* (not necessarily the *client*) is returned.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit.
	Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 80 on page 415 shows an example of GETCLIENTID call instructions.

```
WORKING-STORAGE SECTION.  
  01 SOC-FUNCTION    PIC X(16)  VALUE IS 'GETCLIENTID'.  
  01 CLIENT.  
    03 DOMAIN        PIC 9(8)  BINARY.  
    03 NAME           PIC X(8).  
    03 TASK           PIC X(8).  
    03 RESERVED       PIC X(20).  
  01 ERRNO           PIC 9(8)  BINARY.  
  01 RETCODE          PIC S9(8) BINARY.  
  
PROCEDURE DIVISION.  
  CALL 'EZASOKET' USING SOC-FUNCTION CLIENT ERRNO RETCODE.
```

Figure 80. GETCLIENTID call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions” on page 392](#).

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing GETCLIENTID. The field is left-justified and padded to the right with blanks.

Parameter values returned to the application

CLIENT

A client-ID structure that describes the application that issued the call.

DOMAIN

This is a fullword binary number specifying the domain of the client. On input this is an optional parameter for AF_INET, and required parameter for AF_INET6 to specify the domain of the client. For TCP/IP the value is a decimal 2, indicating AF_INET, or a decimal 19, indicating AF_INET6. On output, this is the returned domain of the client.

NAME

An 8-byte character field set to the caller’s address space name.

TASK

An 8-byte field set to the task identifier of the caller.

RESERVED

Specifies 20-byte character reserved field. This field is required, but not used.

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See [Appendix B, “Socket call error return codes,” on page 743](#) for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check **ERRNO** for an error code.

GETHOSTBYADDR

The GETHOSTBYADDR call returns the domain name and alias name of a host whose IPv4 IP address is specified in the call. A given TCP/IP host can have multiple alias names and multiple host IPv4 IP addresses. The address resolution attempted depends on how the resolver is configured and if any

local host tables exist. See [z/OS Communications Server: IP Configuration Guide](#) for information about configuring the resolver and how local host tables can be used.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state. The PSW key must match the key in which the MVS application task was attached
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 81 on page 416 shows an example of GETHOSTBYADDR call instructions.

```

WORKING-STORAGE SECTION.
  01 SOC-FUNCTION    PIC X(16)  VALUE IS 'GETHOSTBYADDR'.
  01 HOSTADDR        PIC 9(8)   BINARY.
  01 HOSTENT         PIC 9(8)   BINARY.
  01 RETCODE         PIC S9(8)  BINARY.

PROCEDURE DIVISION.
  CALL 'EZASOKET' USING SOC-FUNCTION HOSTADDR HOSTENT RETCODE.

```

Figure 81. GETHOSTBYADDR call instruction example

For equivalent PL/I and assembler language declarations, see ["Converting parameter descriptions"](#) on page 392.

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing GETHOSTBYADDR. The field is left-justified and padded on the right with blanks.

HOSTADDR

A fullword binary field set to the IP address (specified in network byte order) of the host whose name is being sought. See [Appendix B, "Socket call error return codes,"](#) on page 743 for information about ERRNO return codes.

Parameter values returned to the application

HOSTENT

A fullword containing the address of the HOSTENT structure.

RETCODE

A fullword binary field that returns one of the following values:

Value	Description
-------	-------------

0

Successful call.

-1

Check **ERRNO** for an error code.

GETHOSTBYADDR returns the HOSTENT structure shown in [Figure 82 on page 417](#).

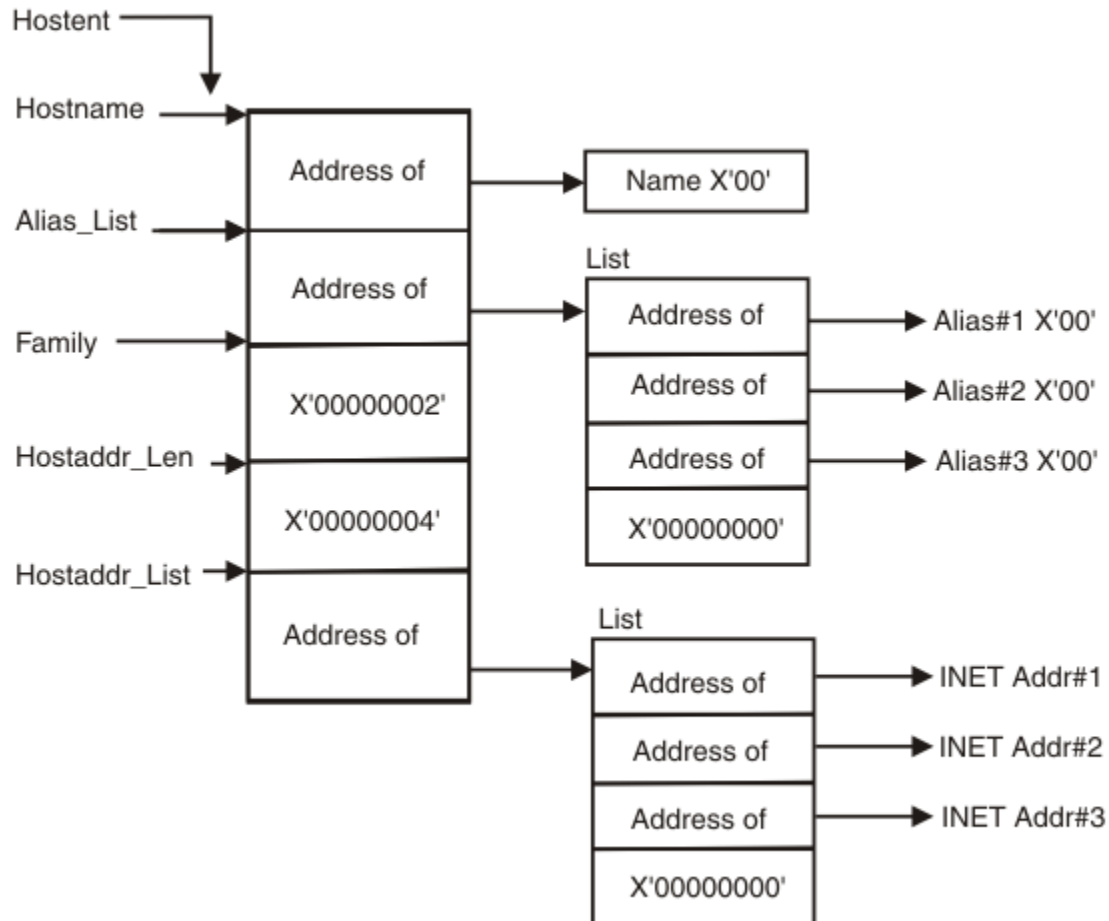


Figure 82. HOSTENT structure that is returned by the GETHOSTBYADDR call

GETHOSTBYADDR returns the HOSTENT structure shown in figure Figure 82 on page 417. The HOSTENT structure is a task's serially reusable storage area. It should not be used or referenced between MVS tasks. The storage is freed when the task terminates. The assembler mapping of the structure is defined in macro EZBREHST, which is installed in the data set specified on your SMP/E DDDEF for MACLIB. The EZBREHST assembler macro is stored in the SYS1.MACLIB library. The macro defines the resolver hostent (host entry), address information (addrinfo) mappings, and services return codes. This structure contains:

- The address of the host name that is returned by the call. The name length is variable and is ended by X'00'.
- The address of a list of addresses that point to the alias names returned by the call. This list is ended by the pointer X'00000000'. Each alias name is a variable length field ended by X'00'.
- The value returned in the FAMILY field is always 2 for AF_INET.
- The length of the host IP address returned in the HOSTADDR_LEN field is always 4 for AF_INET.
- The address of a list of addresses that point to the host IP addresses returned by the call. The list is ended by the pointer X'00000000'. If the call cannot be resolved, the HOSTENT structure contains the ERRNO 10214.

The HOSTENT structure uses indirect addressing to return a variable number of alias names and IP addresses. If you are coding in PL/I or assembler language, this structure can be processed in a relatively straight-forward manner. If you are coding in COBOL, this structure may be difficult to interpret. You can use the subroutine EZACIC08 to simplify interpretation of the information returned by the GETHOSTBYADDR and GETHOSTBYNAME calls. For more information about EZACIC08, see [“EZACIC08 ” on page 527.](#)

GETHOSTBYNAME

The GETHOSTBYNAME call returns the alias name and the IPv4 IP address of a host whose domain name is specified in the call. A given TCP/IP host can have multiple alias names and multiple host IPv4 IP addresses.

The name resolution attempted depends on how the resolver is configured and if any local host tables exist. See [z/OS Communications Server: IP Configuration Guide](#) for information about configuring the resolver and how local host tables can be used.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state. The PSW key must match the key in which the MVS application task was attached.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

[Figure 83 on page 418](#) shows an example of GETHOSTBYNAME call instructions.

```

WORKING-STORAGE SECTION.
  01 SOC-FUNCTION      PIC X(16)  VALUE IS 'GETHOSTBYNAME'.
  01 NAMELEN           PIC 9(8)   BINARY.
  01 NAME              PIC X(255).
  01 HOSTENT           PIC 9(8)   BINARY.
  01 RETCODE           PIC S9(8)  BINARY.

PROCEDURE DIVISION.
  CALL 'EZASOKET' USING SOC-FUNCTION NAMELEN NAME
                      HOSTENT RETCODE.

```

Figure 83. GETHOSTBYNAME call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions” on page 392.](#)

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing GETHOSTBYNAME. The field is left-justified and padded on the right with blanks.

NAMELEN

A value set to the length of the host name. The maximum length is 255.

NAME

A character string, up to 255 characters, set to a host name. Any trailing blanks will be removed from the specified name prior to trying to resolve it to an IP address. This call returns the address of the HOSTENT structure for this name.

Parameter values returned to the application

HOSTENT

A fullword binary field that contains the address of the HOSTENT structure.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

An error occurred.

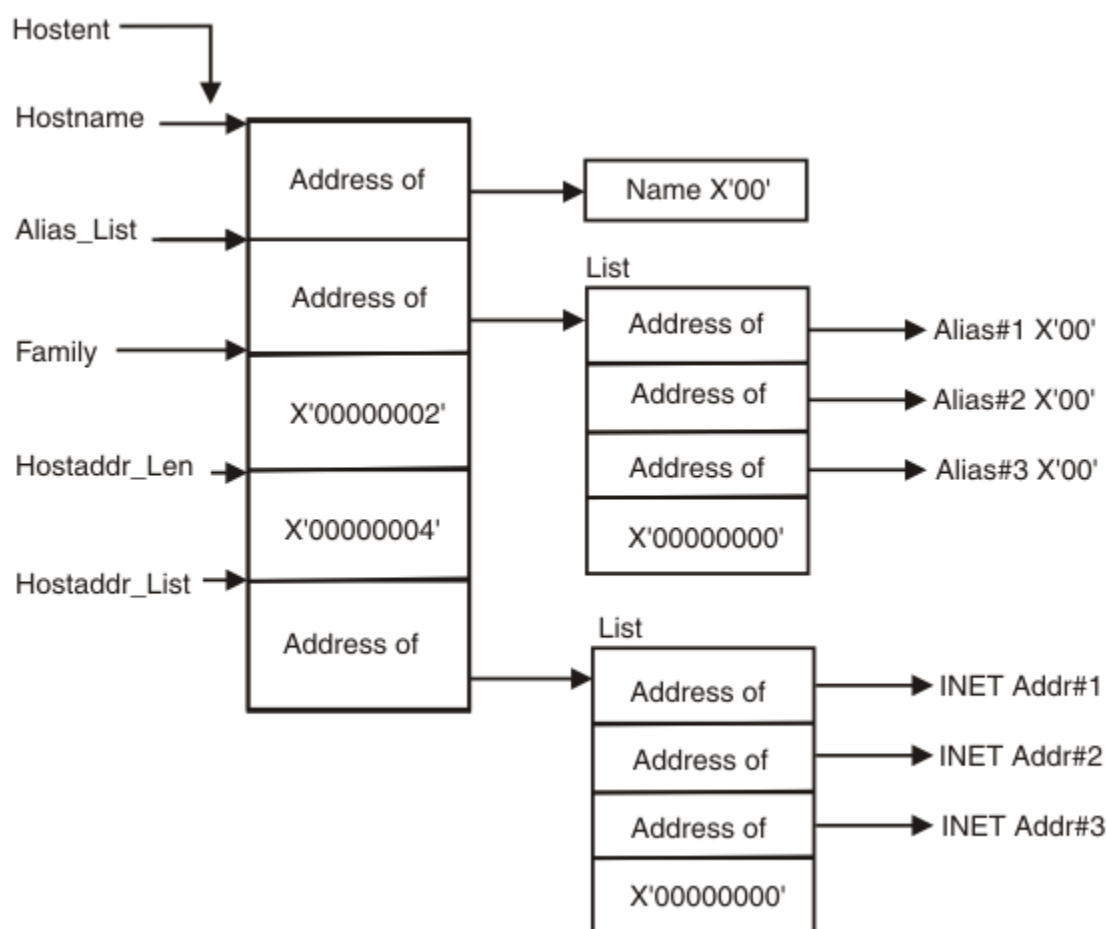


Figure 84. HOSTENT structure returned by the GETHOSTBYNAME call

GETHOSTBYNAME returns the HOSTENT structure shown in Figure 84 on page 419. The HOSTENT structure is a task's serially reusable storage area. It should not be used or referenced between MVS

tasks. The storage is freed when the task terminates. The assembler mapping of the structure is defined in macro EZBREHST, which is installed in the data set specified on your SMP/E DDDEF for MACLIB. The EZBREHST assembler macro is stored in the SYS1.MACLIB library. The macro defines the resolver hostent (host entry), address information (addrinfo) mappings, and services return codes. This structure contains:

- The address of the host name that is returned by the call. The name length is variable and is ended by X'00'.
- The address of a list of addresses that point to the alias names returned by the call. This list is ended by the pointer X'00000000'. Each alias name is a variable length field ended by X'00'.
- The value returned in the FAMILY field is always 2 for AF_INET.
- The length of the host IP address returned in the HOSTADDR_LEN field is always 4 for AF_INET.
- The address of a list of addresses that point to the host IP addresses returned by the call. The list is ended by the pointer X'00000000'. If the call cannot be resolved, the HOSTENT structure contains the ERRNO 10214.

The HOSTENT structure uses indirect addressing to return a variable number of alias names and IP addresses. If you are coding in PL/I or assembler language, this structure can be processed in a relatively straight-forward manner. If you are coding in COBOL, this structure may be difficult to interpret. You can use the subroutine EZACIC08 to simplify interpretation of the information returned by the GETHOSTBYADDR and GETHOSTBYNAME calls. For more information about EZACIC08, see [“EZACIC08 ” on page 527](#).

GETHOSTID

The GETHOSTID call returns the 32-bit IP address for the current host.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389 .
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

[Figure 85 on page 420](#) shows an example of GETHOSTID call instructions.

```

WORKING-STORAGE SECTION.
  01 SOC-FUNCTION    PIC X(16) VALUE IS 'GETHOSTID'.
  01 RETCODE         PIC S9(8) BINARY.

PROCEDURE DIVISION.
  CALL 'EZASOKET' USING SOC-FUNCTION RETCODE.

```

Figure 85. GETHOSTID call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions” on page 392](#).

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing GETHOSTID. The field is left-justified and padded on the right with blanks.

RETCODE

Returns a fullword binary field containing the 32-bit IP address of the host. There is no ERRNO parameter for this call.

GETHOSTNAME

The GETHOSTNAME call returns the domain name of the local host.

Note: The host name returned is the host name the TCPIP stack learned at startup from the TCPIP.DATA file that was found.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

[Figure 86 on page 421](#) shows an example of GETHOSTNAME call instructions.

```
WORKING-STORAGE SECTION.  
  01 SOC-FUNCTION    PIC X(16) VALUE IS 'GETHOSTNAME'.  
  01 NAMELEN         PIC 9(8) BINARY.  
  01 NAME            PIC X(24).  
  01 ERRNO           PIC 9(8) BINARY.  
  01 RETCODE         PIC S9(8) BINARY.  
  
PROCEDURE DIVISION.  
  CALL 'EZASOKET' USING SOC-FUNCTION NAMELEN NAME  
                      ERRNO RETCODE.
```

Figure 86. GETHOSTNAME call instruction example

For equivalent PL/I and assembler language declarations, see ["Converting parameter descriptions"](#) on page 392.

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing GETHOSTNAME. The field is left-justified and padded on the right with blanks.

NAMELEN

A fullword binary field set to the length of the NAME field. The minimum length of the NAME field is 1 character. The maximum length of the NAME field is 255 characters.

Parameter values returned to the application

NAME

Indicates the receiving field for the host name. If the host name is shorter than the NAMELEN value, the NAME field is filled with binary zeros after the host name. If the host name is longer than the NAMELEN value, the name is truncated.

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See [Appendix B, “Socket call error return codes,” on page 743](#) for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check **ERRNO** for an error code.

GETIBMOPT

The GETIBMOPT call returns the number of TCP/IP images installed on a given MVS system and their status, versions, and names. With this information, the caller can dynamically choose the TCP/IP image with which to connect by using the INITAPI call. The GETIBMOPT call is optional. If you do not use the GETIBMOPT call, follow the standard method to determine the connecting TCP/IP image:

- Connect to the TCP/IP specified by TCPIPJOBNAME in the active TCPIP.DATA file.
- Locate TCPIP.DATA using the search order described in the [z/OS Communications Server: IP Configuration Reference](#).

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389 .
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

[Figure 87 on page 423](#) shows an example of GETIBMOPT call instructions.

```

WORKING-STORAGE SECTION.
01 SOC-FUNCTION      PIC X(16)  VALUE IS 'GETIBMOPT'.
01 COMMAND           PIC 9(8)   BINARY VALUE IS 1.
01 BUF.
03 NUM-IMAGES        PIC 9(8) COMP.
03 TCP-IMAGE          OCCURS 8 TIMES.
05 TCP-IMAGE-STATUS  PIC 9(4) BINARY.
05 TCP-IMAGE-VERSION PIC 9(4) BINARY.
05 TCP-IMAGE-NAME     PIC X(8)
01 ERRNO             PIC 9(8)   BINARY.
01 RETCODE            PIC S9(8)  BINARY.

PROCEDURE DIVISION.

CALL 'EZASOKET' USING SOC-FUNCTION COMMAND BUF ERRNO RETCODE.

```

Figure 87. GETIBMOPT call instruction example

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing GETIBMOPT. The field is left-justified and padded on the right with blanks.

COMMAND

A value or the address of a fullword binary number specifying the command to be processed. The only valid value is 1.

Parameter values returned to the application

BUF

A 100-byte buffer into which each active TCP/IP image status, version, and name are placed.

On successful return, these buffer entries contain the status, names, and versions of up to eight active TCP/IP images. The following layout shows the BUF field upon completion of the call.

The NUM_IMAGES field indicates how many entries of TCP_IMAGE are included in the total BUF field. If the NUM_IMAGES returned is 0, there are no TCP/IP images present.

The status field can have a combination of the following information:

Status field

Meaning

X'8xxx'

Active

X'4xxx'

Terminating

X'2xxx'

Down

X'1xxx'

Stopped or stopping

Note: In the above status fields, xxx is reserved for IBM use and can contain any value.

When the status field is returned with a combination of Down and Stopped, TCP/IP abended. Stopped, when returned alone, indicates that TCP/IP was stopped.

The following table shows the examples that are returned on version field:

Version	Field
TCP/IP z/OS Communications Server V1R13	X'061D'
TCP/IP z/OS Communications Server V2R1	X'0621'

The name field is the PROC name, left-justified, and padded with blanks.

NUM_IMAGES (4 bytes)		
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)

Figure 88. Example of name field

ERRNO

A fullword binary field. If RETCODE is negative, this contains an error number. See [Appendix B, “Socket call error return codes,”](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field with the following values:

Value

Description

-1

Call returned error. See ERRNO field.

0

Successful call.

GETNAMEINFO

The GETNAMEINFO call returns the node name and service location of a socket address that is specified in the call. On successful completion, GETNAMEINFO returns the node and service named, if requested, in the buffers provided.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389.
ASC mode:	Primary address space control (ASC) mode.

Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

```
WORKING-STORAGE SECTION.
  01 SOC-FUNCTION      PIC X(16) VALUE IS 'GETNAMEINFO'.
  01 NAMELEN          PIC 9(8) BINARY.
  01 HOST             PIC X(255).
  01 HOSTLEN          PIC 9(8) BINARY.
  01 SERVICE          PIC X(32).
  01 SERVLN          PIC 9(8) BINARY.
  01 FLAGS            PIC 9(8) BINARY VALUE 0.
  01 NI-NOFQDN        PIC 9(8) BINARY VALUE 1.
  01 NI-NUMERICHOST   PIC 9(8) BINARY VALUE 2.
  01 NI-NAMEREQD      PIC 9(8) BINARY VALUE 4.
  01 NI-NUMERICSERVER PIC 9(8) BINARY VALUE 8.
  01 NI-DGRAM         PIC 9(8) BINARY VALUE 16.
  01 NI-NUMERICSCOPE  PIC 9(8) BINARY VALUE 32.

* IPv4 socket structure.
  01 NAME.
    03 FAMILY        PIC 9(4) BINARY.
    03 PORT          PIC 9(4) BINARY.
    03 IP-ADDRESS    PIC 9(8) BINARY.
    03 RESERVED      PIC X(8).

* IPv6 socket structure.
  01 NAME.
    03 FAMILY        PIC 9(4) BINARY.
    03 PORT          PIC 9(4) BINARY.
    03 FLOWINFO      PIC 9(8) BINARY.
    03 IP-ADDRESS.
      10 FILLER      PIC 9(16) BINARY.
      10 FILLER      PIC 9(16) BINARY.
    03 SCOPE-ID      PIC 9(8) BINARY.

  01 ERRNO           PIC 9(8) BINARY.
  01 RETCODE         PIC S9(8) BINARY.

PROCEDURE DIVISION.

  MOVE 28 TO NAMELEN.
  MOVE 255 TO HOSTLEN.
  MOVE 32 TO SERVLN.
  MOVE NI-NAMEREQD TO FLAGS.
  CALL 'EZASOKET' USING SOC-FUNCTION NAME NAMELEN HOST
    HOSTLEN SERVICE SERVLN FLAGS ERRNO RETCODE.
```

Figure 89. GETNAMEINFO call instruction example

Parameter values set by the application

Keyword

Description

SOC-FUNCTION

A 16-byte character field containing GETNAMEINFO. The field is left-justified and padded on the right with blanks.

NAME

An input parameter. A socket address structure to be translated which has the following fields:
The IPv4 socket address structure must specify the following fields:

Field

Description

FAMILY

A halfword binary number specifying the IPv4 addressing family. For TCP/IP the value is a decimal 2, indicating AF_INET.

PORT

A halfword binary number specifying the port number.

IP-ADDRESS

A fullword binary number specifying the 32-bit IPv4 IP address.

RESERVED

An 8-byte reserved field. This field is required, but is not used.

The IPv6 socket address structure specifies the following fields:

Field**Description****FAMILY**

A halfword binary field specifying the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT

A halfword binary number specifying the port number.

FLOWINFO

A fullword binary field specifying the traffic class and flow label. This value of this field is undefined.

IP-ADDRESS

A 16-byte binary field specifying the 128-bit IPv6 IP address, in network byte order.

SCOPE-ID

A fullword binary field that identifies a set of interfaces as appropriate for the scope of the address carried in the IPv6-ADDRESS field. For a link-local scope IPv6-ADDRESS, SCOPE-ID contains the interface index for the IPv6-ADDRESS. For all other address scopes, SCOPE-ID is undefined and is ignored by the resolver.

NAMELEN

An input parameter. A fullword binary field. The length of the socket address structure pointed to by the NAME argument.

HOST

On input, storage capable of holding the returned resolved host name, which may be up to 255 bytes long, for the input socket address. If inadequate storage is specified to contain the resolved host name, then the resolver will return the host name up to the storage specified and truncation may occur. If the host's name cannot be located, the numeric form of the host's address is returned instead of its name. However, if the NI_NAMEREQD option is specified and no host name is located then an error is returned. This is an optional field, but if you specify it, you also must code HOSTLEN. One or both of the following groups of parameters are required:

- The HOST and HOSTLEN parameters
- The SERVICE and SERVLN parameters

Otherwise, an error occurs.

If the IPv6-ADDRESS value is a link-local address, and the SCOPE-ID interface index is nonzero, scope information is appended to the resolved host name using the format *host%scope information*. The scope information can be either the numeric form of the SCOPE-ID interface index or the interface name associated with the SCOPE-ID interface index. Use the NI_NUMERICSCOPE option to select which form should be returned. The combined host name and scope information will still be at most 255 bytes long. For more information about scope information and GETNAMEINFO processing, see [z/OS Communications Server: IPv6 Network and Appl Design Guide](#).

HOSTLEN

An output parameter. A fullword binary field that contains the length of the host storage used to contain the returned resolved host name. The HOSTLEN value must be equal to or greater than the

length of the longest host name, or host name and scope information combination, to be returned. The GETNAMEINFO call returns the host name, or host name and scope information combination, up to the length specified by the HOSTLEN value. On output, the HOSTLEN value contains the length of the returned resolved host name or host name and scope information combination. If HOSTLEN is 0 on input, then the resolved host name is not returned. This is an optional field but if specified you must also code the HOST value. One or both of the following groups of parameters are required:

- The HOST and HOSTLEN parameters
- The SERVICE and SERVLLEN parameters

Otherwise, an error occurs.

SERVICE

On input, storage capable of holding the returned resolved service name, which may be up to 32 bytes long, for the input socket address. If inadequate storage is specified to contain the resolved service name, then the resolver will return the service name up to the storage specified and truncation may occur. If the service name cannot be located, or if NI_NUMERICSERV was specified in the FLAGS operand, then the numeric form of the service address is returned instead of its name. This is an optional field, but if you specify it, you also must code the SERVLLEN value. One or both of the following groups of parameters are required:

- The HOST and HOSTLEN parameters
- The SERVICE and SERVLLEN parameters

Otherwise, an error occurs.

SERVLLEN

An output parameter. A fullword binary field. The length of the SERVICE storage used to contain the returned resolved service name. SERVLLEN must be equal to or greater than the length of the longest service name to be returned. GETNAMEINFO will return the service name up to the length specified by SERVLLEN. On output, SERVLLEN will contain the length of the returned resolved service name. If SERVLLEN is 0 on input, then the service name information will not be returned. This is an optional field, but if you specify it, you also must code the SERVICE value. One or both of the following groups of parameters are required:

- The HOST and HOSTLEN parameters
- The SERVICE and SERVLLEN parameters

Otherwise, an error occurs.

FLAGS

An input parameter. A fullword binary field. This is an optional field. The FLAGS field must contain either a binary value or decimal value, depending on the programming language used:

Flag name	Binary value	Decimal value	Description
'NI_NOFQDN'	X'00000001'	1	Return the NAME portion of the fully qualified domain name.
'NI_NUMERICHOST'	X'00000002'	2	Only return the numeric form of host's address.
'NI_NAMEREQD'	X'00000004'	4	Return an error if the host's name cannot be located.
'NI_NUMERICSERV'	X'00000008'	8	Only return the numeric form of the service address.
'NI_DGRAM'	X'00000010'	16	Indicates that the service is a datagram service. The default behavior is to assume that the service is a stream service.
'NI_NUMERICSERVICE'	X'00000020'	32	Only return the numeric form of the scope information, when applicable

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, ERRNO contains a valid error number. Otherwise, ignore the ERRNO field.

See Appendix B, “Socket call error return codes,” on page 743 for information about ERRNO return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check **ERRNO** for an error code.

GETPEERNAME

The GETPEERNAME call returns the name of the remote socket to which the local socket is connected.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

[Figure 90 on page 429](#) shows an example of GETPEERNAME call instructions.


```

WORKING-STORAGE SECTION.
  01 SOC-FUNCTION PIC X(16) VALUE IS 'GETPEERNAME'.
  01 S PIC 9(4) BINARY.

  * IPv4 socket structure.
  01 NAME.
    03 FAMILY PIC 9(4) BINARY.
    03 PORT PIC 9(4) BINARY.
    03 IP-ADDRESS PIC 9(8) BINARY.
    03 RESERVED PIC X(8).

  * IPv6 socket structure.
  01 NAME.
    03 FAMILY PIC 9(4) BINARY.
    03 PORT PIC 9(4) BINARY.
    03 FLOWINFO PIC 9(8) BINARY.
    03 IP-ADDRESS.
      10 FILLER PIC 9(16) BINARY.
      10 FILLER PIC 9(16) BINARY.
    03 SCOPE-ID PIC 9(8) BINARY.

  01 ERRNO PIC 9(8) BINARY.
  01 RETCODE PIC S9(8) BINARY.

PROCEDURE DIVISION.
  CALL 'EZASOKET' USING SOC-FUNCTION S NAME ERRNO RETCODE.

```

Figure 90. GETPEERNAME call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions” on page 392](#).

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing GETPEERNAME. The field is left-justified and padded on the right with blanks.

S

A halfword binary number set to the socket descriptor of the local socket connected to the remote peer whose address is required.

Parameter Values Returned to the Application

NAME

An IPv4 socket address structure to contain the peer name. The structure that is returned is the socket address structure for the remote socket connected to the local socket specified in field **S**.

FAMILY

A halfword binary field containing the connection peer’s IPv4 addressing family. The call always returns the value decimal 2, indicating AF_INET.

PORT

A halfword binary field set to the connection peer’s port number.

IP-ADDRESS

A fullword binary field set to the 32-bit IPv4 IP address of the connection peer’s host machine.

RESERVED

Specifies an 8-byte reserved field. This field is required, but not used.

An IPv6 socket address structure to contain the peer name. The structure that is returned is the socket address structure for the remote socket that is connected to the local socket specified in field **S**.

FAMILY

A halfword binary field containing the connection peer’s IPv6 addressing family. The call always returns the value decimal 19, indicating AF_INET6.

PORT

A halfword binary field set to the connection peer's port number.

FLOWINFO

A fullword binary field specifying the traffic class and flow label. This value of this field is undefined.

IP-ADDRESS

A 16-byte binary field set to the 128-bit IPv6 IP address of the connection peer's host machine.

SCOPE-ID

A fullword binary field which identifies a set of interfaces as appropriate for the scope of the address carried in the IPv6-ADDRESS field. For a link scope IPv6-ADDRESS, SCOPE-ID contains the link index for the IPv6-ADDRESS. For all other address scopes, SCOPE-ID is undefined.

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See [Appendix B, "Socket call error return codes,"](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value**Description****0**

Successful call.

-1

Check **ERRNO** for an error code.

GETSOCKNAME

The GETSOCKNAME call returns the address currently bound to a specified socket. If the socket is not currently bound to an address, the call returns with the FAMILY field set, and the rest of the structure set to 0.

Since a stream socket is not assigned a name until after a successful call to either BIND, CONNECT, or ACCEPT, the GETSOCKNAME call can be used after an implicit bind to discover which port was assigned to the socket.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

[Figure 91 on page 431](#) shows an example of GETSOCKNAME call instructions.

```

WORKING-STORAGE SECTION.
  01 SOC-FUNCTION PIC X(16) VALUE IS 'GETSOCKNAME'.
  01 S PIC 9(4) BINARY.

  * IPv4 socket address structure.
  01 NAME.
    03 FAMILY PIC 9(4) BINARY.
    03 PORT PIC 9(4) BINARY.
    03 IP-ADDRESS PIC 9(8) BINARY.
    03 RESERVED PIC X(8).

  * IPv6 socket address structure.
  01 NAME.
    03 FAMILY PIC 9(4) BINARY.
    03 PORT PIC 9(4) BINARY.
    03 FLOWINFO PIC 9(8) BINARY.
    03 IP-ADDRESS.
      10 FILLER PIC 9(16) BINARY.
      10 FILLER PIC 9(16) BINARY.
      03 SCOPE-ID PIC 9(8) BINARY.
  01 ERRNO PIC 9(8) BINARY.
  01 RETCODE PIC S9(8) BINARY.

PROCEDURE DIVISION.
  CALL 'EZASOKET' USING SOC-FUNCTION S NAME ERRNO RETCODE.

```

Figure 91. GETSOCKNAME call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions”](#) on page 392.

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing GETSOCKNAME. The field is left-justified and padded on the right with blanks.

S

A halfword binary number set to the descriptor of a local socket whose address is required.

Parameter values returned to the application

NAME

Specifies the IPv4 socket address structure returned by the call.

FAMILY

A halfword binary field containing the IPv4 addressing family. The call always returns the value decimal 2, indicating AF_INET.

PORT

A halfword binary field set to the port number bound to this socket. If the socket is not bound, 0 is returned.

IP-ADDRESS

A fullword binary field set to the 32-bit IP address of the local host machine.

RESERVED

Specifies 8 bytes of binary zeros. This field is required but not used.

NAME

Specifies the IPv6 socket address structure returned by the call.

FAMILY

A halfword binary field containing the IPv6 addressing family. The call always returns the value decimal 19, indicating AF_INET6.

PORT

A halfword binary field set to the port number bound to this socket. If the socket is not bound, 0 is returned.

FLOWINFO

A fullword binary field specifying the traffic class and flow label. This value of this field is undefined.

IP-ADDRESS

A 16 byte binary field set to the 128-bit IPv6 IP address in network byte order, of the local host machine.

SCOPE-ID

A fullword binary field which identifies a set of interfaces as appropriate for the scope of the address carried in the IPv6-ADDRESS field. For a link scope IPv6-ADDRESS, SCOPE-ID contains the link index for the IPv6-ADDRESS. For all other address scopes, SCOPE-ID is undefined.

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See [Appendix B, "Socket call error return codes," on page 743](#) for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value**Description****0**

Successful call.

-1

Check **ERRNO** for an error code.

GETSOCKOPT

The GETSOCKOPT call queries the options that are set by the SETSOCKOPT call.

Several options are associated with each socket. These options are described below. You must specify the option to be queried when you issue the GETSOCKOPT call.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389 .
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

[Figure 92 on page 433](#) shows an example of GETSOCKOPT call instructions.

```

WORKING-STORAGE SECTION.
  01 SOC-FUNCTION PIC X(16) VALUE IS 'GETSOCKOPT'.
  01 S PIC 9(4) BINARY.
  01 OPTNAME PIC 9(8) BINARY.

  01 OPTVAL PIC 9(8) BINARY.
  If OPNAME = SO-LINGER then
  01 OPTVAL PIC X(16).

  01 OPTLEN PIC 9(8) BINARY.
  01 ERRNO PIC 9(8) BINARY.
  01 RETCODE PIC S9(8) BINARY.

PROCEDURE DIVISION.
  CALL 'EZASOKET' USING SOC-FUNCTION S OPTNAME
    OPTVAL OPTLEN ERRNO RETCODE.

```

Figure 92. GETSOCKOPT call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions”](#) on page 392.

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing GETSOCKOPT. The field is left-justified and padded on the right with blanks.

S

A halfword binary number specifying the socket descriptor for the socket requiring options.

OPTNAME

Set **OPTNAME** to the required option before you issue GETSOCKOPT. See the following table for a list of the options and their unique requirements.

See [Appendix D, “GETSOCKOPT/SETSOCKOPT command values,”](#) on page 767 for the numeric values of **OPTNAME**.

Note: COBOL programs cannot contain field names with the underbar character. Fields representing the option name should contain dashes instead.

OPTLEN

Input parameter. A fullword binary field containing the length of the data returned in **OPTVAL**. See the following table for determining on what to base the value of **OPTLEN**.

Parameter values returned to the application

OPTVAL

For the GETSOCKOPT API, **OPTVAL** will be an output parameter. See the following table for a list of the options and their unique requirements.

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See [Appendix B, “Socket call error return codes,”](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check ERRNO for an error code.

Table 18. OPTNAME options for GETSOCKOPT and SETSOCKOPT

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IP_ADD_MEMBERSHIP</p> <p>Use this option to enable an application to join a multicast group on a specific interface. An interface has to be specified with this option. Only applications that want to receive multicast datagrams need to join multicast groups.</p> <p>This is an IPv4-only socket option.</p>	<p>Contains the IP_MREQ structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 interface address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IP-MREQ.</p>	<p>N/A</p>
<p>IP_ADD_SOURCE_MEMBERSHIP</p> <p>Use this option to enable an application to join a source multicast group on a specific interface and a specific source address. You must specify an interface and a source address with this option. Applications that want to receive multicast datagrams need to join source multicast groups.</p> <p>This is an IPv4-only socket option.</p>	<p>Contains the IP_MREQ_SOURCE structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ_SOURCE structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 source address and a 4-byte IPv4 interface address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ_SOURCE.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IP-MREQ-SOURCE.</p>	<p>N/A</p>
<p>IP_BLOCK_SOURCE</p> <p>Use this option to enable an application to block multicast packets that have a source address that matches the given IPv4 source address. You must specify an interface and a source address with this option. The specified multicast group must have been joined previously.</p> <p>This is an IPv4-only socket option.</p>	<p>Contains the IP_MREQ_SOURCE structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ_SOURCE structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 source address and a 4-byte IPv4 interface address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ_SOURCE.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IP-MREQ-SOURCE.</p>	<p>N/A</p>

Table 18. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IP_DROP_MEMBERSHIP</p> <p>Use this option to enable an application to exit a multicast group or to exit all sources for a multicast group.</p> <p>This is an IPv4-only socket option.</p>	<p>Contains the IP_MREQ structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 interface address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IP-MREQ.</p>	<p>N/A</p>
<p>IP_DROP_SOURCE_MEMBERSHIP</p> <p>Use this option to enable an application to exit a source multicast group.</p> <p>This is an IPv4-only socket option.</p>	<p>Contains the IP_MREQ_SOURCE structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ_SOURCE structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 source address and a 4-byte IPv4 interface address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ_SOURCE.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IP-MREQ-SOURCE.</p>	<p>N/A</p>
<p>IP_MULTICAST_IF</p> <p>Use this option to set or obtain the IPv4 interface address used for sending outbound multicast datagrams from the socket application.</p> <p>This is an IPv4-only socket option.</p> <p>Note: Multicast datagrams can be transmitted only on one interface at a time.</p>	<p>A 4-byte binary field containing an IPv4 interface address.</p>	<p>A 4-byte binary field containing an IPv4 interface address.</p>
<p>IP_MULTICAST_LOOP</p> <p>Use this option to control or determine whether a copy of multicast datagrams is looped back for multicast datagrams sent to a group to which the sending host itself belongs. The default is to loop the datagrams back.</p> <p>This is an IPv4-only socket option.</p>	<p>A 1-byte binary field.</p> <p>To enable, set to 1.</p> <p>To disable, set to 0.</p>	<p>A 1-byte binary field.</p> <p>If enabled, will contain a 1.</p> <p>If disabled, will contain a 0.</p>

Table 18. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IP_MULTICAST_TTL</p> <p>Use this option to set or obtain the IP time-to-live of outgoing multicast datagrams. The default value is '01'x meaning that multicast is available only to the local subnet.</p> <p>This is an IPv4-only socket option.</p>	<p>A 1-byte binary field containing the value of '00'x to 'FF'x.</p>	<p>A 1-byte binary field containing the value of '00'x to 'FF'x.</p>
<p>IP_UNBLOCK_SOURCE</p> <p>Use this option to enable an application to unblock a previously blocked source for a given IPv4 multicast group. You must specify an interface and a source address with this option.</p> <p>This is an IPv4-only socket option.</p>	<p>Contains the IP_MREQ_SOURCE structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ_SOURCE structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 source address and a 4-byte IPv4 interface address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ_SOURCE.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IP-MREQ-SOURCE.</p>	

Table 18. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IPV6_ADDR_PREFERENCES</p> <p>Use this option to query or set IPv6 address preferences of a socket. The default source address selection algorithm considers these preferences when it selects an IP address that is appropriate to communicate with a given destination address.</p> <p>This is an AF_INET6-only socket option.</p> <p>Result: These flags are only preferences. The stack could assign a source IP address that does not conform to the IPV6_ADDR_PREFERENCES flags that you specify.</p> <p>Guideline: Use the INET6_IS_SRCADDR function to test whether the source IP address matches one or more IPV6_ADDR_PREFERENCES flags.</p>	<p>Contains the 4-byte flags field IPV6_ADDR_PREFERENCES_FLAGS that is defined in SYS1.MACLIB(BPXYSOCK) with the following flags:</p> <p>IPV6_PREFER_SRC_HOME (X'00000001') Prefer home address</p> <p>IPV6_PREFER_SRC_COA (X'00000002') Prefer care-of address</p> <p>IPV6_PREFER_SRC_TMP (X'00000004') Prefer temporary address</p> <p>IPV6_PREFER_SRC_PUBLIC (X'00000008') Prefer public address</p> <p>IPV6_PREFER_SRC_CGA (X'00000010') Prefer cryptographically generated address</p> <p>IPV6_PREFER_SRC_NONCGA (X'00000020') Prefer non-cryptographically generated address</p> <p>Some of these flags are contradictory. Combining contradictory flags, such as IPV6_PREFER_SRC_CGA and IPV6_PREFER_SRC_NONCGA, results in error code EINVAL.</p> <p>See IPV6_ADDR_PREFERENCES and Mapping of GAI_HINTS/GAI_ADDRINFO EFLAGS in SEZAINST(CBLOCK) for the PL/I example of the OPTNAME and flag definitions.</p> <p>See IPV6_ADDR_PREFERENCES and AI_EFLAGS mappings in SEZAINST(EZACOBOL) for the COBOL example of the OPTNAME and flag definitions.</p>	<p>Contains the 4-byte flags field IPV6_ADDR_PREFERENCES_FLAGS that is defined in SYS1.MACLIB(BPXYSOCK) with the following flags:</p> <p>IPV6_PREFER_SRC_HOME (X'00000001') Prefer home address</p> <p>IPV6_PREFER_SRC_COA (X'00000002') Prefer care-of address</p> <p>IPV6_PREFER_SRC_TMP (X'00000004') Prefer temporary address</p> <p>IPV6_PREFER_SRC_PUBLIC (X'00000008') Prefer public address</p> <p>IPV6_PREFER_SRC_CGA (X'00000010') Prefer cryptographically generated address</p> <p>IPV6_PREFER_SRC_NONCGA (X'00000020') Prefer non-cryptographically generated address</p> <p>See IPV6_ADDR_PREFERENCES and Mapping of GAI_HINTS/GAI_ADDRINFO EFLAGS in SEZAINST(CBLOCK) for the PL/I example of the OPTNAME and flag definitions.</p> <p>See IPV6_ADDR_PREFERENCES and AI_EFLAGS mappings in SEZAINST(EZACOBOL) for the COBOL example of the OPTNAME and flag definitions.</p>

Table 18. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IPV6_JOIN_GROUP</p> <p>Use this option to control the reception of multicast packets and specify that the socket join a multicast group.</p> <p>This is an IPv6-only socket option.</p>	<p>Contains the IPV6_MREQ structure as defined in SYS1.MACLIB(BPXYSOCK). The IPV6_MREQ structure contains a 16-byte IPv6 multicast address followed by a 4-byte IPv6 interface index number.</p> <p>If the interface index number is 0, then the stack chooses the local interface.</p> <p>See the SEZAINST(CBLOCK) for the PL/I example of IPV6_MREQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IPV6-MREQ.</p>	<p>N/A</p>
<p>IPV6_LEAVE_GROUP</p> <p>Use this option to control the reception of multicast packets and specify that the socket leave a multicast group.</p> <p>This is an IPv6-only socket option.</p>	<p>Contains the IPV6_MREQ structure as defined in SYS1.MACLIB(BPXYSOCK). The IPV6_MREQ structure contains a 16-byte IPv6 multicast address followed by a 4-byte IPv6 interface index number.</p> <p>If the interface index number is 0, then the stack chooses the local interface.</p> <p>See the SEZAINST(CBLOCK) for the PL/I example of IPV6_MREQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IPV6-MREQ.</p>	<p>N/A</p>
<p>IPV6_MULTICAST_HOPS</p> <p>Use to set or obtain the hop limit used for outgoing multicast packets.</p> <p>This is an IPv6-only socket option.</p>	<p>Contains a 4-byte binary value specifying the multicast hops. If not specified, then the default is 1 hop.</p> <p>-1 indicates use stack default.</p> <p>0 – 255 is the valid hop limit range.</p> <p>Note: An application must be APF authorized to enable it to set the hop limit value above the system defined hop limit value. CICS applications cannot execute as APF authorized.</p>	<p>Contains a 4-byte binary value in the range 0 – 255 indicating the number of multicast hops.</p>

Table 18. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IPv6_MULTICAST_IF</p> <p>Use this option to set or obtain the index of the IPv6 interface used for sending outbound multicast datagrams from the socket application.</p> <p>This is an IPv6-only socket option.</p>	<p>Contains a 4-byte binary field containing an IPv6 interface index number.</p>	<p>Contains a 4-byte binary field containing an IPv6 interface index number.</p>
<p>IPv6_MULTICAST_LOOP</p> <p>Use this option to control or determine whether a multicast datagram is looped back on the outgoing interface by the IP layer for local delivery when datagrams are sent to a group to which the sending host itself belongs. The default is to loop multicast datagrams back.</p> <p>This is an IPv6-only socket option.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to 1.</p> <p>To disable, set to 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>
<p>IPv6_UNICAST_HOPS</p> <p>Use this option to set or obtain the hop limit used for outgoing unicast IPv6 packets.</p> <p>This is an IPv6-only socket option.</p>	<p>Contains a 4-byte binary value specifying the unicast hops. If not specified, then the default is 1 hop.</p> <p>-1 indicates use stack default.</p> <p>0 – 255 is the valid hop limit range.</p> <p>Note: APF authorized applications are permitted to set a hop limit that exceeds the system configured default. CICS applications cannot execute as APF authorized.</p>	<p>Contains a 4-byte binary value in the range 0 – 255 indicating the number of unicast hops.</p>
<p>IPv6_V6ONLY</p> <p>Use this option to set or determine whether the socket is restricted to send and receive only IPv6 packets. The default is to not restrict the sending and receiving of only IPv6 packets.</p> <p>This is an IPv6-only socket option.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to 1.</p> <p>To disable, set to 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>

Table 18. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>MCAST_BLOCK_SOURCE</p> <p>Use this option to enable an application to block multicast packets that have a source address that matches the given source address. You must specify an interface index and a source address with this option. The specified multicast group must have been joined previously.</p>	<p>Contains the GROUP_SOURCE_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_SOURCE_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address and a socket address structure of the source address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_SOURCE_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-SOURCE-REQ.</p>	N/A
<p>MCAST_JOIN_GROUP</p> <p>Use this option to enable an application to join a multicast group on a specific interface. You must specify an interface index. Applications that want to receive multicast datagrams must join multicast groups.</p>	<p>Contains the GROUP_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-REQ.</p>	N/A
<p>MCAST_JOIN_SOURCE_GROUP</p> <p>Use this option to enable an application to join a source multicast group on a specific interface and a source address. You must specify an interface index and the source address. Applications that want to receive multicast datagrams only from specific source addresses need to join source multicast groups.</p>	<p>Contains the GROUP_SOURCE_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_SOURCE_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address and a socket address structure of the source address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_SOURCE_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-SOURCE-REQ.</p>	N/A

Table 18. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>MCAST_LEAVE_GROUP</p> <p>Use this option to enable an application to exit a multicast group or exit all sources for a given multicast groups.</p>	<p>Contains the GROUP_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-REQ.</p>	<p>N/A</p>
<p>MCAST_LEAVE_SOURCE_GROUP</p> <p>Use this option to enable an application to exit a source multicast group.</p>	<p>Contains the GROUP_SOURCE_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_SOURCE_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address and a socket address structure of the source address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_SOURCE_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-SOURCE-REQ.</p>	<p>N/A</p>
<p>MCAST_UNBLOCK_SOURCE</p> <p>Use this option to enable an application to unblock a previously blocked source for a given multicast group. You must specify an interface index and a source address with this option.</p>	<p>Contains the GROUP_SOURCE_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_SOURCE_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address and a socket address structure of the source address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_SOURCE_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-SOURCE-REQ.</p>	<p>N/A</p>

Table 18. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_ASCII</p> <p>Use this option to set or determine the translation to ASCII data option. When SO_ASCII is set, data is translated to ASCII. When SO_ASCII is not set, data is not translated to or from ASCII.</p> <p>Note: This is a REXX-only socket option.</p>	<p>To enable, set to ON.</p> <p>To disable, set to OFF.</p> <p>Note: The <i>optvalue</i> is returned and is optionally followed by the name of the translation table that is used if translation is applied to the data.</p>	<p>If enabled, contains ON.</p> <p>If disabled, contains OFF.</p> <p>Note: The <i>optvalue</i> is returned and is optionally followed by the name of the translation table that is used if translation is applied to the data.</p>
<p>SO_BROADCAST</p> <p>Use this option to set or determine whether a program can send broadcast messages over the socket to destinations that can receive datagram messages. The default is disabled.</p> <p>Note: This option has no meaning for stream sockets.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to 1 or a positive value.</p> <p>To disable, set to 0.</p>	<p>A 4-byte field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>
<p>SO_DEBUG</p> <p>Use SO_DEBUG to set or determine the status of the debug option. The default is <i>disabled</i>. The debug option controls the recording of debug information.</p> <p>Notes:</p> <ol style="list-style-type: none"> 1. This is a REXX-only socket option. 2. This option has meaning only for stream sockets. 	<p>To enable, set to ON.</p> <p>To disable, set to OFF.</p>	<p>If enabled, contains ON.</p> <p>If disabled, contains OFF.</p>
<p>SO_EBCDIC</p> <p>Use this option to set or determine the translation to EBCDIC data option. When SO_EBCDIC is set, data is translated to EBCDIC. When SO_EBCDIC is not set, data is not translated to or from EBCDIC. This option is ignored by EBCDIC hosts.</p> <p>Note: This is a REXX-only socket option.</p>	<p>To enable, set to ON.</p> <p>To disable, set to OFF.</p> <p>Note: The <i>optvalue</i> is returned and is optionally followed by the name of the translation table that is used if translation is applied to the data.</p>	<p>If enabled, contains ON.</p> <p>If disabled, contains OFF.</p> <p>Note: The <i>optvalue</i> is returned and is optionally followed by the name of the translation table that is used if translation is applied to the data.</p>
<p>SO_ERROR</p> <p>Use this option to request pending errors on the socket or to check for asynchronous errors on connected datagram sockets or for other errors that are not explicitly returned by one of the socket calls. The error status is clear afterwards.</p>	<p>N/A</p>	<p>A 4-byte binary field containing the most recent ERRNO for the socket.</p>

Table 18. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_KEEPAIVE</p> <p>Use this option to set or determine whether the keep alive mechanism periodically sends a packet on an otherwise idle connection for a stream socket.</p> <p>The default is disabled.</p> <p>When activated, the keep alive mechanism periodically sends a packet on an otherwise idle connection. If the remote TCP does not respond to the packet or to retransmissions of the packet, the connection is terminated with the error ETIMEDOUT.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to 1 or a positive value.</p> <p>To disable, set to 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>
<p>SO_LINGER</p> <p>Use this option to control or determine how TCP/IP processes data that has not been transmitted when a CLOSE is issued for the socket. The default is disabled.</p> <p>Notes:</p> <ol style="list-style-type: none"> 1. This option has meaning only for stream sockets. 2. If you set a zero linger time, the connection cannot close in an orderly manner, but stops, resulting in a RESET segment being sent to the connection partner. Also, if the aborting socket is in nonblocking mode, the close call is treated as though no linger option had been set. <p>When SO_LINGER is set and CLOSE is called, the calling program is blocked until the data is successfully transmitted or the connection has timed out.</p> <p>When SO_LINGER is not set, the CLOSE returns without blocking the caller, and TCP/IP continues to attempt to send data for a specified time. This usually allows sufficient time to complete the data transfer.</p> <p>Use of the SO_LINGER option does not guarantee successful completion because TCP/IP waits only the amount of time specified in OPTVAL for SO_LINGER.</p>	<p>Contains an 8-byte field containing two 4-byte binary fields.</p> <p>Assembler coding:</p> <pre>ONOFF DS F LINGER DS F</pre> <p>COBOL coding:</p> <pre>ONOFF PIC 9(8) BINARY. LINGER PIC 9(8) BINARY.</pre> <p>Set ONOFF to a nonzero value to enable and set to 0 to disable this option. Set LINGER to the number of seconds that TCP/IP lingers after the CLOSE is issued.</p>	<p>Contains an 8-byte field containing two 4-byte binary fields.</p> <p>Assembler coding:</p> <pre>ONOFF DS F LINGER DS F</pre> <p>COBOL coding:</p> <pre>ONOFF PIC 9(8) BINARY. LINGER PIC 9(8) BINARY.</pre> <p>A nonzero value returned in ONOFF indicates enabled, a 0 indicates disabled. LINGER indicates the number of seconds that TCP/IP will try to send data after the CLOSE is issued.</p>

Table 18. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_OOBLINE</p> <p>Use this option to control or determine whether out-of-band data is received.</p> <p>Note: This option has meaning only for stream sockets.</p> <p>When this option is set, out-of-band data is placed in the normal data input queue as it is received and is available to a RECV or a RECVFROM even if the OOB flag is not set in the RECV or the RECVFROM.</p> <p>When this option is disabled, out-of-band data is placed in the priority data input queue as it is received and is available to a RECV or a RECVFROM only when the OOB flag is set in the RECV or the RECVFROM.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to 1 or a positive value.</p> <p>To disable, set to 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>
<p>SO_RCVBUF</p> <p>Use this option to control or determine the size of the data portion of the TCP/IP receive buffer.</p> <p>The size of the data portion of the receive buffer is protocol-specific, based on the following values prior to any SETSOCKOPT call:</p> <ul style="list-style-type: none"> • TCPRCVBufsize keyword on the TCPCONFIG statement in the PROFILE.TCPIP data set for a TCP Socket • UDPRCVBufsize keyword on the UDPCONFIG statement in the PROFILE.TCPIP data set for a UDP Socket • The default of 65535 for a raw socket 	<p>A 4-byte binary field.</p> <p>To enable, set to a positive value specifying the size of the data portion of the TCP/IP receive buffer.</p> <p>To disable, set to a 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a positive value indicating the size of the data portion of the TCP/IP receive buffer.</p> <p>If disabled, contains a 0.</p>

Table 18. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_RCVTIMEO</p> <p>Use this option to control or determine the maximum length of time that a receive-type function can wait before it completes.</p> <p>If a receive-type function has blocked for the maximum length of time that was specified without receiving data, control is returned with an errno set to EWOULDBLOCK. The default value for this option is 0, which indicates that a receive-type function does not time out.</p> <p>When the MSG_WAITALL flag (stream sockets only) is specified, the timeout takes precedence. The receive-type function can return the partial count. See the explanation of that operation's MSG_WAITALL flag parameter.</p> <p>The following receive-type functions are supported:</p> <ul style="list-style-type: none"> • READ • READV • RECV • RECVFROM • RECVMMSG 	<p>This option requires a TIMEVAL structure, which is defined in SYS1.MACLIB(BPXRLIM) macro. The TIMEVAL structure contains the number of seconds and microseconds specified as fullword binary numbers. The seconds can be a value in the range 0 - 2 678 400 (equal to 31 days), and the microseconds can be a value in the range 0 - 1 000 000 (equal to 1 second). Although TIMEVAL value can be specified using microsecond granularity, the internal TCP/IP timers that are used to implement this function have a granularity of approximately 100 milliseconds.</p>	<p>This option stores a TIMEVAL structure that is defined in the SYS1.MACLIB(BPXRLIM) macro. The TIMEVAL structure contains the number of seconds and microseconds, which are specified as fullword binary numbers. The number of seconds value that is returned is in the range 0 - 2 678 400 (equal to 31 days). The number of microseconds value that is returned is in the range 0 - 1 000 000.</p>

Table 18. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_REUSEADDR</p> <p>Use this option to control or determine whether local addresses are reused. The default is disabled. This alters the normal algorithm used with BIND. The normal BIND algorithm allows each Internet address and port combination to be bound only once. If the address and port have been already bound, then a subsequent BIND will fail and result error will be EADDRINUSE.</p> <p>When this option is enabled, the following situations are supported:</p> <ul style="list-style-type: none"> • A server can BIND the same port multiple times as long as every invocation uses a different local IP address and the wildcard address INADDR_ANY is used only one time per port. • A server with active client connections can be restarted and can bind to its port without having to close all of the client connections. • For datagram sockets, multicasting is supported so multiple bind() calls can be made to the same class D address and port number. • If you require multiple servers to BIND to the same port and listen on INADDR_ANY, see the SHAREPORT option on the PORT statement in TCPIP.PROFILE. 	<p>A 4-byte binary field.</p> <p>To enable, set to 1 or a positive value.</p> <p>To disable, set to 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>
<p>SO_SNDBUF</p> <p>Use this option to control or determine the size of the data portion of the TCP/IP send buffer. The size of the TCP/IP send buffer is protocol specific and is based on the following conditions:</p> <ul style="list-style-type: none"> • The TCPSENDBufsize keyword on the TCPCONFIG statement in the PROFILE.TCPIP data set for a TCP socket • The UDPSENDBufsize keyword on the UDPCONFIG statement in the PROFILE.TCPIP data set for a UDP socket • The default of 65535 for a raw socket 	<p>A 4-byte binary field.</p> <p>To enable, set to a positive value specifying the size of the data portion of the TCP/IP send buffer.</p> <p>To disable, set to a 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a positive value indicating the size of the data portion of the TCP/IP send buffer.</p> <p>If disabled, contains a 0.</p>

Table 18. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_SNDTIMEO</p> <p>Use this option to control or determine the maximum length of time that a send-type function can remain blocked before it completes.</p> <p>If a send-type function has blocked for this length of time, it returns with a partial count or, if no data is sent, with an errno set to EWOULDBLOCK. The default value for this is 0, which indicates that a send-type function does not time out.</p> <p>For a SETSOCKOPT, the following send-type functions are supported:</p> <ul style="list-style-type: none"> • SEND • SENDMSG • SENDTO • WRITE • WRITEV 	<p>This option requires a TIMEVAL structure, which is defined in the SYS1.MACLIB(BPXYRLIM) macro. The TIMEVAL structure contains the number of seconds and microseconds specified as fullword binary numbers. The seconds value is in the range 0 - 2 678 400 (equal to 31 days), and the microseconds value is in the range 0 - 1 000 000 (equal to 1 second). Although the TIMEVAL value can be specified using microsecond granularity, the internal TCP/IP timers that are used to implement this function have a granularity of approximately 100 milliseconds.</p>	<p>This option stores a TIMEVAL structure that is defined in SYS1.MACLIB(BPXYRLIM). The TIMEVAL structure contains the number of seconds and microseconds, which are specified as fullword binary numbers. The number of seconds value that is returned is in the range 0 - 2 678 400 (equal to 31 days). The microseconds value that is returned is in the range 0 - 1 000 000.</p>
<p>SO_TYPE</p> <p>Use this option to return the socket type.</p>	<p>N/A</p>	<p>A 4-byte binary field indicating the socket type:</p> <p>X'1' indicates SOCK_STREAM.</p> <p>X'2' indicates SOCK_DGRAM.</p> <p>X'3' indicates SOCK_RAW.</p>
<p>TCP_KEEPAIVE</p> <p>Use this option to set or determine whether a socket-specific timeout value (in seconds) is to be used in place of a configuration-specific value whenever keep alive timing is active for that socket.</p> <p>When activated, the socket-specified timer value remains in effect until respecified by SETSOCKOPT or until the socket is closed. For more information about the socket option parameters, see TCP_KeepAlive socket option inz/OS Communications Server: IP Programmer's Guide and Reference.</p> <p>Tip: The site administrator can enable the global keep-alive mechanism by specifying the INTERVAL parameter on the TCPCONFIG statement in the TCP/IP stack profile data set, TCPIP.PROFILE.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to a value in the range of 1 – 2 147 460.</p> <p>To disable, set to a value of 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains the specific timer value (in seconds) that is in effect for the given socket.</p> <p>If disabled, contains a 0 indicating keep alive timing is not active.</p>

Table 18. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>TCP_NODELAY</p> <p>Use this option to set or determine whether data sent over the socket is subject to the Nagle algorithm (RFC 896).</p> <p>Under most circumstances, TCP sends data when it is presented. When this option is enabled, TCP will wait to send small amounts of data until the acknowledgment for the previous data sent is received. When this option is disabled, TCP will send small amounts of data even before the acknowledgment for the previous data sent is received.</p> <p>Note: Use the following format to set TCP_NODELAY OPTNAME value for COBOL programs:</p> <pre> 01 TCP-NODELAY-VAL PIC 9(10) COMP VALUE 2147483649. 01 TCP-NODELAY-REDEF REDEFINES TCP-NODELAY-VAL. 05 FILLER PIC 9(6) BINARY. 05 TCP-NODELAY PIC 9(8) BINARY. </pre>	<p>A 4-byte binary field.</p> <p>To enable, set to a 0.</p> <p>To disable, set to a 1 or nonzero.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 0.</p> <p>If disabled, contains a 1.</p>

GIVESOCKET

The GIVESOCKET call is used to pass a socket from one process to another.

UNIX-based platforms use a command called FORK to create a new child process that has the same descriptors as the parent process. You can use this new child process in the same way that you used the parent process.

TCP/IP normally uses GETCLIENTID, GIVESOCKET, and TAKESOCKET calls in the following sequence:

1. A process issues a GETCLIENTID call to get the job name of its region and its MVS subtask identifier. This information is used in a GIVESOCKET call.
2. The process issues a GIVESOCKET call to prepare a socket for use by a child process.
3. The child process issues a TAKESOCKET call to get the socket. The socket now belongs to the child process, and can be used by TCP/IP to communicate with another process.

Note: The TAKESOCKET call returns a new socket descriptor in RETCODE. The child process must use this new socket descriptor for all calls that use this socket. The socket descriptor that was passed to the TAKESOCKET call must not be used.

4. After issuing the GIVESOCKET command, the parent process issues a SELECT command that waits for the child to get the socket.
5. When the child gets the socket, the parent receives an exception condition that releases the SELECT command.
6. The parent process closes the socket.

The original socket descriptor can now be reused by the parent.

Sockets that have been given, but not taken for a period of four days, will be closed and will no longer be available for taking. If a select for the socket is outstanding, it will be posted.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 93 on page 449 shows an example of GIVESOCKET call instructions.

```
WORKING-STORAGE SECTION.  
  01 SOC-FUNCTION PIC X(16) VALUE IS 'GIVESOCKET'.  
  01 S PIC 9(4) BINARY.  
  01 CLIENT.  
    03 DOMAIN PIC 9(8) BINARY.  
    03 NAME PIC X(8).  
    03 TASK PIC X(8).  
    03 RESERVED PIC X(20).  
  01 ERRNO PIC 9(8) BINARY.  
  01 RETCODE PIC S9(8) BINARY.  
  
PROCEDURE DIVISION.  
  CALL 'EZASOKET' USING SOC-FUNCTION S CLIENT ERRNO RETCODE.
```

Figure 93. GIVESOCKET call instruction example

For equivalent PL/I and assembler language declarations, see ["Converting parameter descriptions"](#) on page 392.

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing GIVESOCKET. The field is left-justified and padded on the right with blanks.

S

A halfword binary number set to the socket descriptor of the socket to be given.

CLIENT

A structure containing the identifier of the application to which the socket should be given.

DOMAIN

A fullword binary number that must be set to decimal 2, indicating AF_INET, or decimal 19 indicating AF_INET6.

Note: A socket given by GIVESOCKET can only be taken by a TAKESOCKET with the same DOMAIN (AF_INET or AF_INET6).

NAME

Specifies an eight-character field, left-justified, padded to the right with blanks, that can be set to the name of the MVS address space that will contain the application that is going to take the socket.

- If the socket-taking application is in the *same* address space as the socket-giving application (as in CICS), NAME can be specified. The socket-giving application can determine its own address space name by issuing the GETCLIENTID call.
- If the socket-taking application is in a *different* MVS address space (as in IMS), this field should be set to blanks. When this is done, any MVS address space that requests the socket can have it.

TASK

Specifies an 8-byte field that can be set to blanks, or to the identifier of the socket-taking MVS subtask. If this field is set to blanks, any subtask in the address space specified in the NAME field can take the socket.

- As used by IMS and CICS, the field should be set to blanks.
- If TASK identifier is non-blank, the socket-receiving task should already be in execution when the GIVESOCKET is issued.

RESERVED

A 20-byte reserved field. This field is required, but not used.

Parameter values returned to the application

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See [Appendix B, “Socket call error return codes,”](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check **ERRNO** for an error code.

INET6_IS_SRCADDR

The INET6_IS_SRCADDR call verifies whether the input IP address matches an IP address in the node that conforms to all IPV6_ADDR_PREFERENCES flags specified in the call. You can use this call with IPv6 addresses or with IPv4-mapped IPv6 addresses.

You can use this call to test local IP addresses to verify whether these addresses have the characteristics that are required by your application.

See RFC 5014 *IPv6 Socket API for Source Address Selection* for more information about the INET6_IS_SRCADDR call. See [Appendix G, “Related protocol specifications,”](#) on page 779 for information about accessing RFCs.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389.
ASC mode:	Primary address space control (ASC) mode.

Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 94 on page 451 shows an example of INET6_IS_SRCADDR call instructions.

```

WORKING-STORAGE SECTION.
01 SOC-FUNCTION    PIC X(16)  VALUE IS 'INET6_IS_SRCADDR'.
   * IPv6 socket address structure.
01 NAME.
   03 FAMILY       PIC 9(4)  BINARY.
   03 PORT         PIC 9(4)  BINARY.
   03 FLOWINFO     PIC 9(8)  BINARY.
   03 IP-ADDRESS.
       10 FILLER    PIC 9(16) BINARY.
       10 FILLER    PIC 9(16) BINARY.
   03 SCOPE-ID     PIC 9(8)  BINARY.
01 FLAGS          PIC 9(8)  BINARY
   88 IPV6-PREFER-SRC-HOME      PIC 9(8) BINARY VALUE 1.
   88 IPV6-PREFER-SRC-COA      PIC 9(8) BINARY VALUE 2.
   88 IPV6-PREFER-SRC-TMP      PIC 9(8) BINARY VALUE 4.
   88 IPV6-PREFER-SRC-PUBLIC    PIC 9(8) BINARY VALUE 8.
   88 IPV6-PREFER-SRC-CGA      PIC 9(8) BINARY VALUE 16.
   88 IPV6-PREFER-SRC-NONCGA   PIC 9(8) BINARY VALUE 32.
01 ERRNO          PIC 9(8)  BINARY.
01 RETCODE        PIC S9(8)  BINARY.

PROCEDURE DIVISION.
   CALL 'EZASOKET' USING SOC-FUNCTION NAME FLAGS ERRNO RETCODE.

```

Figure 94. INET6_IS_SRCADDR call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions” on page 392](#).

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing INET6_IS_SRCADDR.

NAME

Specifies the AF_INET6 socket address structure for the address that is to be tested.

Requirement: You must specify an AF_INET6 address. You can specify an IPv6 address, or an IPv4-mapped IPv6 address.

The IPv6 socket address structure specifies the following fields:

FAMILY

A halfword binary field that specifies the IPv6 addressing family. For TCP/IP the value is the decimal value 19, indicating AF_INET6.

PORT

A halfword binary field. This field is ignored by INET6_IS_SRCADDR processing.

FLOWINFO

A fullword binary field that specifies the traffic class and flow label. This field is ignored by INET6_IS_SRCADDR processing.

IP-ADDRESS

A 16-byte binary field that is set to the 128-bit IPv6 IP address (network byte order) of the IP address to be tested.

Rule: Specify an IPv4 address by using its IPv4-mapped IPv6 address format.

SCOPE-ID

A fullword binary field that identifies a set of appropriate interfaces for the scope of the address that is specified in the IP-ADDRESS field. The value 0 indicates that the SCOPE-ID field does not identify the set of interfaces to be used.

Requirements:

- If the IP address is a link-local address, this field must be set to a nonzero value.
- If the IP address is not a link-local address, this field must be set to 0.

FLAGS

A fullword binary field that contains one or more valid IPV6_ADDR_PREFERENCES flags.

Flag name	Binary value	Decimal value	Description
IPV6_PREFER_SRC_HOME	X'00000001'	1	Test whether the input IP address is a home address. ¹
IPV6_PREFER_SRC_COA	X'00000002'	2	Test whether the input IP address is a care-of address. ²
IPV6_PREFER_SRC_TMP	X'00000004'	4	Test whether the input IP address is a temporary address.
IPV6_PREFER_SRC_PUBLIC	X'00000008'	8	Test whether the input IP address is a public address.
IPV6_PREFER_SRC_CGA	X'00000010'	16	Test whether the input IP address is cryptographically generated. ²
IPV6_PREFER_SRC_NONCGA	X'00000020'	32	Test whether the input IP address is not cryptographically generated. ¹
Notes: 1. Any valid IP address that is known to the stack satisfies this flag. 2. z/OS Communications Server does not support this type of address. The call always returns FALSE if this flag is specified with a valid IP address that is known to the stack.			

Tips:

- The SEZAINST(EZACOBOL) and SEZAINST(CBLOCK) samples contain mappings for these flags. For assembler programs, the flags are defined in the system maclib member BPXYSOCK.
- Some of these flags are contradictory, for example:
 - The flag IPV6_PREFER_SRC_HOME contradicts the flag IPV6_PREFER_SRC_COA.
 - The flag IPV6_PREFER_SRC_CGA contradicts the flag IPV6_PREFER_SRC_NONCGA.
 - The flag IPV6_PREFER_SRC_TMP contradicts the flags IPV6_PREFER_SRC_PUBLIC.

Result: If you specify contradictory flags in the call, the result is FALSE.

Parameter values returned to the application

ERRNO

A fullword binary field. If the RETCODE value is negative, the field contains an error number. See [Appendix B, “Socket call error return codes,” on page 743](#) for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value	Description
-------	-------------

0

FALSE

The call was successful and the result is FALSE. The input AF_INET6 address corresponds to an IP address on the node, but does not conform to one or more IPV6_ADDR_PREFERENCES flags that are specified in the call.

1

TRUE

The call was successful and the result is TRUE. The input AF_INET6 address corresponds to an IP address on the node and conforms to all IPV6_ADDR_PREFERENCES flags that are specified in the call.

-1

Check **ERRNO** for an error code.

INITAPI

The INITAPI call connects an application to the TCP/IP interface. Almost all sockets programs that are written in COBOL, PL/I, or assembler language must issue the INITAPI socket command before they issue other socket commands.

The exceptions to this rule are the following calls, which, when issued first, will generate a default INITAPI call.

- GETCLIENTID
- GETHOSTID
- GETHOSTNAME
- GETIBMOPT
- SELECT
- SELECTEX
- SOCKET
- TAKESOCKET

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

[Figure 95 on page 454](#) shows an example of INITAPI call instructions.

```

WORKING-STORAGE SECTION.
01  SOC-FUNCTION      PIC X(16)  VALUE IS 'INITAPI'.
01  MAXSOC            PIC 9(4)  BINARY.
01  IDENT.
    02  TCPNAME        PIC X(8).
    02  ADSNAME        PIC X(8).
01  SUBTASK           PIC X(8).
01  MAXSNO            PIC 9(8)  BINARY.
01  ERRNO             PIC 9(8)  BINARY.
01  RETCODE           PIC S9(8)  BINARY.

PROCEDURE DIVISION.
    CALL 'EZASOKET' USING SOC-FUNCTION MAXSOC IDENT SUBTASK
    MAXSNO ERRNO RETCODE.

```

Figure 95. INITAPI call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions” on page 392](#).

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing INITAPI. The field is left-justified and padded on the right with blanks.

MAXSOC

A halfword binary field set to the maximum number of sockets this application will ever have open at one time. The maximum number is 65535 and the minimum number is 50. This value is used to determine the amount of memory that is allocated for socket control blocks and buffers. If less than 50 are requested, MAXSOC defaults to 50.

IDENT

A structure containing the identities of the TCP/IP address space and the calling program's address space. Specify IDENT on the INITAPI call from an address space.

TCPNAME

An 8-byte character field that should be set to the MVS job name of the TCP/IP address space with which you are connecting.

ADSNAME

An 8-byte character field set to the identity of the calling program's address space. It is the name of the CICS startup job. For explicit-mode IMS server programs, use the TIMSrvAddrSpc field passed in the TIM. If ADSNAME is not specified, the system derives a value from the MVS control block structure.

SUBTASK

Indicates an 8-byte field that contains a unique subtask identifier, which is used to distinguish between multiple subtasks within a single address space. Use your own job name as part of your subtask name. This ensures that, if you issue more than one INITAPI command from the same address space, each SUBTASK parameter is unique.

Restriction: EZASOKET calls outside of the CICS environment are not reentrant. If EZASOKET is to be used by a multithread or multitask application, a separate copy needs to be loaded for each thread or task. See [z/OS Communications Server: IP CICS Sockets Guide](#) for information about use in the CICS environment.

Parameter values returned to the application

MAXSNO

A fullword binary field that contains the highest socket number assigned to this application. The lowest socket number is 0. If you have 50 sockets, they are numbered from 0 to 49. If MAXSNO is not specified, the value for MAXSNO is 49.

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See [Appendix B, “Socket call error return codes,”](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check **ERRNO** for an error code.

IOCTL

The IOCTL call is used to control certain operating characteristics for a socket.

Before you issue an IOCTL socket command, you must load a value that represents the characteristic that you want to control into the COMMAND field.

The variable length parameters REQARG and RETARG are arguments that are passed to and returned from IOCTL. The length of REQARG and RETARG is determined by the value that you specify in COMMAND. See [Table 19 on page 462](#) for information about REQARG and RETARG.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

[Figure 96 on page 456](#) shows an example of IOCTL call instructions.

```

WORKING-STORAGE SECTION.
01 SOC-FUNCTION          PIC X(16) VALUE 'IOCTL'.
01 S                     PIC 9(4)  BINARY.
01 COMMAND               PIC 9(8)  BINARY.

01 IFREQ.
03 NAME                  PIC X(16).
03 FAMILY                PIC 9(4)  BINARY.
03 PORT                  PIC 9(4)  BINARY.
03 ADDRESS               PIC 9(8)  BINARY.
03 RESERVED              PIC X(8).

01 IFREQOUT.
03 NAME                  PIC X(16).
03 FAMILY                PIC 9(4)  BINARY.
03 PORT                  PIC 9(4)  BINARY.
03 ADDRESS               PIC 9(8)  BINARY.
03 RESERVED              PIC X(8).

01 GRP-IOCTL-TABLE.
02 IOCTL-ENTRY OCCURS 100 TIMES.
03 NAME                  PIC X(16).
03 FAMILY                PIC 9(4)  BINARY.
03 PORT                  PIC 9(4)  BINARY.
03 ADDRESS               PIC 9(8)  BINARY.
03 NULLS                 PIC X(8).

01 IOCTL-REQARG          USAGE IS POINTER.
01 IOCTL-RETARG          USAGE IS POINTER.
01 ERRNO                 PIC 9(8)  BINARY.
01 RETCODE               PIC 9(8)  BINARY.

PROCEDURE DIVISION.
    CALL 'EZASOCKET' USING SOC-FUNCTION S COMMAND REQARG
    RETARG ERRNO RETCODE.

```

Figure 96. IOCTL call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions” on page 392](#).

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing IOCTL. The field is left-justified and padded to the right with blanks.

S

A halfword binary number set to the descriptor of the socket to be controlled.

COMMAND

To control an operating characteristic, set this field to one of the following symbolic names. A value in a bit mask is associated with each symbolic name. By specifying one of these names, you are turning on a bit in a mask which communicates the requested operating characteristic to TCP/IP.

FIONBIO

Sets or clears blocking status.

FIONREAD

Returns the number of immediately readable bytes for the socket.

SIOCATMARK

Determines whether the current location in the data input is pointing to out-of-band data.

SIOCGHOMEIF6

Requests all IPv6 home interfaces.

Note: To request OSM interfaces, the application must have READ authorization to the EZB.OSM.sysname.tcpname resource.

When the SIOCGHOMEIF6 IOCTL is issued, the REGARQ must contain a Network Configuration Header. The NETCONFHDR is defined in the SYS1.MACLIB(BPXVIOC6) for assembler language. The following fields are input fields and must be filled out:

NchEyeCatcher

Contains eye catcher '6NCH'

NchIoctl

Contains the command code

NchBufferLength

Buffer length large enough to contain all the IPv6 interface records. Each interface record is length of HOME-IF-ADDRESS. If buffer is not large enough, then errno will be set to ERANGE and the NchNumEntryRet will be set to number of interfaces. Based on NchNumEntryRet and size of HOME-IF-ADDRESS, calculate the necessary storage to contain the entire list.

NchBufferPtr

This is a pointer to an array of HOME-IF structures returned on a successful call. The size will depend on the number of qualifying interfaces returned.

NchNumEntryRet

If return code is 0 this will be set to number of HOME-IF-ADDRESS returned. If errno is ERANGE, then will be set to number of qualifying interfaces. No interfaces are returned. Recalculate The NchBufferLength based on this value times the size of HOME-IF-ADDRESS.

REQARG and RETARG

Point to the arguments that are passed between the calling program and IOCTL. The length of the argument is determined by the COMMAND request. REQARG is an input parameter and is used to pass arguments to IOCTL. RETARG is an output parameter and is used for arguments returned by IOCTL. For the lengths and meanings of REQARG and RETARG for each COMMAND type, see [Table 19 on page 462](#).

```
Working-Storage Section.
01 SIOCGHOMEIF6-VAL          pic s9(10) binary value 3222599176.
01 SIOCGHOMEIF6-REDEF REDEFINES SIOCGHOMEIF6-VAL.
   05 FILLER                  PIC 9(6) COMP.
   05 SIOCGHOMEIF6            PIC 9(8) COMP.
01 IOCTL-RETARG              USAGE IS POINTER.
01 NET-CONF-HDR.
   05 NCH-EYE-CATCHER         PIC X(4) VALUE '6NCH'.
   05 NCH-IOCTL               PIC 9(8) BINARY.
   05 NCH-BUFFER-LENGTH       PIC 9(8) BINARY.
   05 NCH-BUFFER-PTR          USAGE IS POINTER.
   05 NCH-NUM-ENTRY-RET       PIC 9(8) BINARY.
01 HOME-IF.
   03 HOME-IF-ADDRESS.
   05 FILLER                  PIC 9(16) BINARY.

Linkage Section.

01 L1.
   03 NetConfHdr.
   05 NchEyeCatcher           pic x(4).
   05 NchIoctl                pic 9(8) binary.
   05 NchBufferLength         pic 9(8) binary.
   05 NchBufferPtr            usage is pointer.
   05 NchNumEntryRet          pic 9(8) binary.
* Allocate storage based on your need.
   03 Allocated-Storage       pic x(nn).

Procedure Division using L1.
   move '6NCH' to NchEyeCatcher.
   set NchBufferPtr to address of Allocated-Storage.
* Set NchBufferLength to the length of your allocated storage.
   move nn to NchBufferLength.
   move SIOCGHOMEIF6 to NchIoctl.
   Call 'EZASOKET' using socket-ioctl socket-descriptor
                           SIOCGHOMEIF6
                           NETCONFHDR NETCONFHDR
                           errno retcode.
```

Figure 97. COBOL language example for SIOCGHOMEIF6

SIOCGIFADDR

Requests the IPv4 network interface address for a given interface name. For assembler, see the IOCN_IFNAME field in the SYS1.MACLIB(BPX1IOCC) API. For COBOL, see the IFR-NAME field in the SEZAINST(EZACOBOL) API. For PL/I, see the IFR_NAME field in the SEZAINST(CBLOCK) API.

SIOCGIFBRDADDR

Requests the IPv4 network interface broadcast address for a given interface name. For assembler, see the IOCN_IFNAME field in the SYS1.MACLIB(BPX1IOCC) API. For COBOL, see the IFR-NAME field in the SEZAINST(EZACOBOL) API. For PL/I, see the IFR_NAME field in the SEZAINST(CBLOCK) API.

SIOCGIFCONF

Requests the IPv4 network interface configuration. The configuration is a variable number of 32-byte structures. For assembler, see the IOCN_IFREQ field in the SYS1.MACLIB(BPX1IOCC) API for the structure format. For COBOL, see the IFREQ field in the SEZAINST(EZACOBOL) API for the structure format. For PL/I, see the IFREQ field in the SEZAINST(CBLOCK) API for the structure format.

- When IOCTL is issued, REQARG must contain the length of the array to be returned. To determine the length of REQARG, multiply the structure length (array element) by the number of interfaces requested. The maximum number of array elements that TCP/IP can return is 100.
- When IOCTL is issued, RETARG must be set to the beginning of the storage area that you have defined in your program for the array to be returned.

SIOCGIFDSTADDR

Requests the network interface destination address for a given interface name. For assembler, see the IOCN_IFNAME field in the SYS1.MACLIB(BPX1IOCC) API. For COBOL, see the IFR-NAME field in the SEZAINST(EZACOBOL) API. For PL/I, see the IFR_NAME field in the SEZAINST(CBLOCK) API.

SIOCGIFMTU

Requests the IPv4 network interface MTU (maximum transmission unit) for a given interface name. For assembler, see the IOCN_IFNAME field in the SYS1.MACLIB(BPX1IOCC) API. For COBOL, see the IFR-NAME field in the SEZAINST(EZACOBOL) API. For PL/I, see the IFR_NAME field in the SEZAINST(CBLOCK) API.

SIOCGIFNAMEINDEX

Requests all interface names and interface indexes including local loopback but excluding VIPAs. Information is returned for both IPv4 and IPv6 interfaces whether they are active or inactive. For IPv6 interfaces, information is only returned for an interface if it has at least one available IP address.

Note: To request OSM interfaces, the application must have READ authorization to the EZB.OSM.sysname.tcpname resource.

The configuration consists of IF_NAMEINDEX structure, which is defined in SYS1.MACLIB(BPX1IOCC) for the assembler language.

- When the SIOCGIFNAMEINDEX IOCTL is issued, the first word in REQARG must contain the length (in bytes) to contain an IF-NAME-INDEX structure to return the interfaces. The formula to compute this length is as follows:
 1. Determine the number of interfaces expected to be returned upon successful completion of this command.
 2. Multiply the number of interfaces by the array element (size of IF-NIINDEX, IF-NINAME, and IF-NIEXT) to get the size of the array element.
 3. Add the size of the IF-NITOTALIF and IF-NIENTRIES to the size of the array to get the total number of bytes needed to accommodate the name and index information returned.
- When IOCTL is issued, RETARG must be set to the address of the beginning of the area in your program's storage that is reserved for the IF-NAMEINDEX structure that is to be returned by IOCTL.

- The command 'SIOCGIFNAMEINDEX' returns a variable number of all the qualifying network interfaces.

```

WORKING-STORAGE SECTION.
01 SIOCGIFNAMEINDEX-VAL pic 9(10) binary value 1073804803.
01 SIOCGIFNAMEINDEX-REDEF REDEFINES SIOCGIFNAMEINDEX-VAL.
05 FILLER PIC 9(6) COMP.
05 SIOCGIFNAMEINDEX PIC 9(8) COMP.
01 reqarg pic 9(8) binary.
01 reqarg-header-only pic 9(8) binary.
01 IF-NIHEADER.
05 IF-NITOTALIF PIC 9(8) BINARY.
05 IF-NIENTRIES PIC 9(8) BINARY.
01 IF-NAME-INDEX-ENTRY.
05 IF-NIINDEX PIC 9(8) BINARY.
05 IF-NINAME PIC X(16).
05 IF-NINAMETERM PIC X(1).
05 IF-NIRESV1 PIC X(3).
01 OUTPUT-STORAGE PIC X(500).
Procedure Division.
move 8 to reqarg-header-only.
Call 'EZASOKET' using socket-ioctl socket-descriptor
                        SIOCGIFNAMEINDEX
                        REQARG-HEADER-ONLY IF-NIHEADER
                        errno retcode.

move 500 to reqarg.
Call 'EZASOKET' using socket-ioctl socket-descriptor
                        SIOCGIFNAMEINDEX
                        REQARG OUTPUT-STORAGE
                        errno retcode.

```

Figure 98. COBOL language example for SIOCGIFNAMEINDEX

SIOCGIPMSFILTER

Requests a list of the IPv4 source addresses that comprise the source filter, with the current mode on a given interface and a multicast group for a socket. The source filter can include or exclude the set of source address, depending on the filter mode (MCAST_INCLUDE or MCAST_EXCLUDE). When the SIOCGIPMSFILTER IOCTL is issued, the REQARG parameter must contain a IP_MSFILTER structure, which is defined in SYS1.MACLIB(BPXVIOCC) for assembler language, in SEZAINST(CBLOCK) for PL/I, and in SEZAINST(EZACOBOL) for COBOL. The IP_MSFILTER must include an interface address (input), a multicast address (input), filter mode (output), the number of source addresses in the following array (input and output), and an array of source addresses (output). On input, the number of source addresses is the number of source addresses that will fit in the input array. On output, the number of source addresses contains the total number of source filters in the output array. If the application does not know the size of the source list prior to processing, it can make a reasonable guess (for example, 0), and if when the call completes the number of source addresses is a greater value, the IOCTL can be repeated with a buffer that is large enough. That is, on output, the number of source addresses is always updated to be the total number of sources in the filter, but the array holds as many source addresses as will fit, up to the minimum of the array size passed in as the input number.

Calculate the size of IF_MSFILTER value as follows:

1. Determine the number of expected source addresses.
2. Multiply the number of source addresses by the array element (size of the IMSF_SrcEntry value) to determine the size of all array elements.
3. Add the size of all array elements to the size of the IMSF_Header value to determine the total number of bytes needed to accommodate the source addresses information that is returned.

SIOCGMSFILTER

Requests a list of the IPv4 or IPv6 source addresses that comprise the source filter, with the current mode on a given interface index and a multicast group for a socket. The source filter can include or exclude the set of source address, depending on the filter mode (MCAST_INCLUDE or MCAST_EXCLUDE). When the SIOCGMSFILTER IOCTL is issued, the REQARG parameter must contain a GROUP_FILTER structure, which is defined in SYS1.MACLIB(BPXVIOCC) for assembler , in SEZAINST(CBLOCK) for PL/I, and in SEZAINST(EZACOBOL) for COBOL. The GROUP_FILTER option must include an interface index (input), a socket address structure of the multicast address

(input), filter mode (output), the number of source addresses in the following array (output), and an array of the socket address structure of source addresses (input and output). On input, the number of source addresses is the number of source addresses that will fit in the input array. On output, the number of source addresses contains the total number of source filters in the output array. If the application does not know the size of the source list prior to processing, it can make a reasonable guess (for example, 0), and if when the call completes the number of source addresses is a greater value, the IOCTL can be repeated with a buffer that is large enough. That is, on output, the number of source addresses is always updated to be the total number of sources in the filter, but the array holds as many source addresses as will fit, up to the minimum of the array size passed in as the input number.

Calculate the size of the GROUP_FILTER value as follows:

1. Determine the number of source addresses expected.
2. Multiply the number of source addresses by the array element (size of the GF_SrcEntry value) to determine the size of all array elements.
3. Add the size of all array elements to the size of the GF_Header value to determine the total number of bytes needed to accommodate the source addresses information returned.

SIOCGPARTNERINFO

Provides an interface for an application to retrieve security information about its partner. When you issue the SIOCGPARTNERINFO IOCTL, the REQARG parameter must contain a PartnerInfo structure. The PartnerInfo structure is defined in members within SEZANMAC; EZBPINF1 defines the PL/I layout, EZBPINF2 defines the assembler layout, and EZBPINF3 defines the COBOL layout. For more information about using the SIOCGPARTNERINFO IOCTL, see [z/OS Communications Server: IP Programmer's Guide and Reference](#).

SIOCSAPPLDATA

The SIOCSAPPLDATA IOCTL enables an application to set 40 bytes of user-specified application data against a socket endpoint. You can also use this application data to identify socket endpoints in interfaces such as Netstat, SMF, or network management applications. When the SIOCSAPPLDATA IOCTL is issued, the REQARG parameter must contain a SetApplData structure as defined by the EZBYAPPL macro. See the CBLOCK and the EZACOBOL samples for the equivalent SetApplData and SetADcontainer structure definitions for PL/I and COBOL programming environments. See [z/OS Communications Server: IP Programmer's Guide and Reference](#) for more information about programming the SIOCSAPPLDATA IOCTL.

SetAD_buffer: The user-defined application data is 40 bytes of data that identifies the endpoint with the application. You can obtain this application data from the following sources:

- Netstat reports. The information is displayed in the ALL/-A report. If you use the APPLDATA modifier, then the information also is displayed on the ALLConn/-a and Conn/-c reports.
- The SMF 119 TCP connection termination record. See [TCP connection termination record \(subtype 2\)](#) in [z/OS Communications Server: IP Programmer's Guide and Reference](#) for more information.
- Network management interfaces. See [IPSec NMI](#) in [z/OS Communications Server: IP Programmer's Guide and Reference](#) for more information.

Consider the following guidelines:

- The application must document the content, format and meaning of the ApplData strings that it associates with the sockets that it owns.
- The application should uniquely identify itself with printable EBCDIC characters at the beginning of the string. Strings beginning with 3-character IBM product identifiers, such as TCP/IP's EZA or EZB, are reserved for IBM use. IBM product identifiers begin with a letter in the range A-I.
- Use printable EBCDIC characters for the entire string to enable searching with Netstat filters.

Tip: Separate application data elements with a blank for easier reading.

SIOCSIPMSFILTER

Sets a list of the IPv4 source addresses that comprise the source filter, with the current mode on a given interface and a multicast group for a socket. The source filter can include or exclude the set of source address, depending on the filter mode (MCAST_INCLUDE or MCAST_EXCLUDE). When the SIOCSIPMSFILTER IOCTL is issued, the REQARG parameter must contain a IP_MSFILTER structure, which is defined in SYS1.MACLIB(BPXYIOCC) for assembler, in SEZAINST(CBLOCK) for PL/I and in SEZAINST(EZACOBOL) for COBOL. The IP_MSFILTER option must include an interface address, a multicast address, filter mode, the number of source addresses in the following array, and an array of source addresses.

Calculate the size of the IF_MSFILTER value as follows:

1. Determine the number of expected source addresses.
2. Multiply the number of source addresses by the array element (size of the IMSF_SrcEntry value) to determine the size of all array elements.
3. Add the size of all array elements to the size of the IMSF_Header value to determine the total number of bytes needed to accommodate the source addresses information that is returned.

SIOCSMSFILTER

Sets a list of the IPv4 or IPv6 source addresses that comprise the source filter, along with the current mode on a given interface index and a multicast group for a socket. The source filter can include or exclude the set of source address, depending on the filter mode (INCLUDE or EXCLUDE). When the SIOCSMSFILTER IOCTL is issued, the REQARG parameter must contain a GROUP_FILTER structure which is defined in SYS1.MACLIB(BPXYIOCC) for assembler, in SEZAINST(CBLOCK) for PL/I, and in SEZAINST(EZACOBOL) for COBOL. The GROUP_FILTER option must include an interface index, a socket address structure of the multicast address, filter mode, the number of source addresses in the following array, and an array of the socket address structure of source addresses.

Calculate the size of GROUP_FILTER as follows:

1. Determine the number of source addresses expected.
2. Multiply the number of source addresses by the array element (size of the GF_SrcEntry value) to get the size of all array elements.
3. Add the size of all array elements to the size of the GF_Header value to get the total number of bytes needed to accommodate the source addresses information returned.

SIOCSPARTNERINFO

The SIOCSPARTNERINFO IOCTL sets an indicator to retrieve the partner security credentials during connection setup and saves the information, enabling an application to issue a SIOCGPARTNERINFO IOCTL without suspending the application, or at least minimizing the time it takes to retrieve the information. The SIOCSPARTNERINFO IOCTL must be issued prior to the SIOCGPARTNERINFO IOCTL. When you issue the SIOCSPARTNERINFO IOCTL, the REQARG parameter must contain a constant value, PI_REQTYPE_SET_PARTNERDATA. This constant is defined in members within SEZANMAC; EZBPINF1 defines the PL/I layout, EZBPINF A defines the assembler layout, and EZBPINF B defines the COBOL layout. For more information about using the [SIOCSPARTNERINFO](#) IOCTL, see [z/OS Communications Server: IP Programmer's Guide and Reference](#).

SIOCTLSSL

Controls Application Transparent Transport Layer Security (AT-TLS) for the connection. REQARG and RETARG must contain a TTLS_IOCTL structure. If a partner certificate is requested, the TTLS_IOCTL must include a pointer to additional buffer space and the length of that buffer. Information is returned in the TTLS_IOCTL structure. If a partner certificate is requested and one is available, it is returned in the additional buffer space. The TTLS_IOCTL structure is defined in members within SEZANMAC. EZBZTLS1 defines the PL/I layout, EZBZTLSP defines the assembler layout, and EZBZTLSB defines the COBOL layout. For more usage details, see the [Application Transparent Transport Layer Security \(AT-TLS\)](#) information in [z/OS Communications Server: IP Programmer's Guide and Reference](#).

Restriction: Use of this ioctl for functions other than query requires that the AT-TLS policy mapped to the connection be defined with the ApplicationControlled parameter set to On.

REQARG and RETARG

Points to arguments that are passed between the calling program and IOCTL. The length of the argument is determined by the COMMAND request. REQARG is an input parameter or an output parameter and is used to pass and receive arguments to and from IOCTL. RETARG is an output parameter and receives arguments from IOCTL. The REQARG and RETARG parameters are described in Table 19 on page 462.

Table 19. IOCTL call arguments

COMMAND/CODE	SIZE	REQARG	SIZE	RETARG
FIONBIO X'8004A77E'	4	Set socket mode to: X'00'=blocking, X'01'=nonblocking.	0	Not used.
FIONREAD X'4004A77F'	0	Not used.	4	Number of characters available for read.
SIOCATMARK X'4004A707'	0	Not used.	4	X'00'= not at OOB data X'01'= at OOB data.
SIOCGHOMEIF6 X'C014F608'	20	NetConfHdr		See Figure 97 on page 457 NetConfHdr.
SIOCGIFADDR X'C020A70D'	32	First 16 bytes - interface name. Last 16 bytes - not used.	32	Network interface address. For assembler, see the IOCN_SADDRIF field in the SYS1.MACLIB(BPXIOCC) API. For COBOL, see the IFR-ADDR field in the SEZAINST(EZACOBOL) API. For PL/I, see the IFR_ADDR field in the SEZAINST(CBLOCK) API.
SIOCGIFBRDADDR X'C020A712'	32	First 16 bytes - interface name. Last 16 bytes - not used.	32	Network interface address. For assembler, see the IOCN_SADDRIFBROADCAST field in the SYS1.MACLIB(BPXIOCC) API. For COBOL, see the IFR-BROADADDR field in the SEZAINST(EZACOBOL) API. For PL/I, see the IFR_BROADADDR field in the SEZAINST(CBLOCK) API.
SIOCGIFCONF X'C008A714'	8	Size of RETARG.	See note ¹ .	
SIOCGIFDSTADDR X'C020A70F'	32	First 16 bytes - interface name. Last 16 bytes - not used.	32	Destination interface address. For assembler, see the IOCN_SADDRIFDEST field in the SYS1.MACLIB(BPXIOCC) API. For COBOL, see the IFR-DSTADDR field in the SEZAINST(EZACOBOL) API. For PL/I, see the IFR_DSTADDR field in the SEZAINST(CBLOCK) API.
SIOCGIFMTU X'C020A726'	32	First 16 bytes - interface name. Last 16 bytes - not used.	32	IPv4 interface MTU (maximum transmission unit). For assembler, see the IOCN_MTUSIZE field in the SYS1.MACLIB(BPXIOCC) API. For COBOL, see the IFR-MTU field in the SEZAINST(EZACOBOL) API. For PL/I, see the IFR_MTU field in the SEZAINST(CBLOCK) API.
SIOCGIFNAMEINDEX X'4000F603'	4	First 4 bytes size of return buffer.		See Figure 98 on page 459 IF- NAMEINDEX .
SIOCGIPMSFILTER X'C000A724'	–	See IP_MSFILTER structure in macro BPXYIOCC. See note 2.	0	Not used

Table 19. IOCTL call arguments (continued)

COMMAND/CODE	SIZE	REQARG	SIZE	RETARG
SIOCGMSFILTER X'C000F610'	–	See GROUP_FILTER structure in macro BPXYIOCC. See note 3	0	Not used
SIOCGPARTNERINFO X'C000F612'	–	For the PartnerInfo structure layout, see SEZANMAC(EZBPINF1) for assembler, SEZANMAC(EZBPINF1) for PL/I, and SEZANMAC(EZBPINF1) for COBOL. See note 4.	0	Not used
SIOCSAPPLDATA X'8018D90C'	–	See SETAPPLDATA structure in macro EZBYAPPL	0	Not used
SIOCSIPMSFILTER X'8000A725'	–	See IP_MSFILTER structure in macro BPXYIOCC. See note 2.	0	Not used
SIOCSMSFILTER X'8000F611'	–	See GROUP_FILTER structure in macro BPXYIOCC. See note 3	0	Not used
SIOCSPARTNERINFO X'8004F613'	4	See PI_REQTYPE_SET_PARTNERDATA in SEZANMAC(EZBPINF1) for assembler, SEZANMAC(EZBPINF1) for PL/I, and SEZANMAC(EZBPINF1) for COBOL.	0	Not used
SIOCTLCTL X'C038D90B'	56	For IOCTL structure layout, see SEZANMAC(EZBZTLS1) for PL/I, SEZANMAC(EZBZTLSP) for assembler, and SEZANMAC(EZBZTLSP) for COBOL.	56	For IOCTL structure layout, see SEZANMAC(EZBZTLS1) for PL/I, SEZANMAC(EZBZTLSP) for assembler, and SEZANMAC(EZBZTLSP) for COBOL.

Notes:

1. When you call IOCTL with the SIOCGIFCONF command set, REQARG should contain the length in bytes of RETARG. Each interface is assigned a 32-byte array element and REQARG should be set to the number of interfaces times 32. TCP/IP Services can return up to 100 array elements.
2. The size of the IP_MSFILTER structure must be equal to or greater than the size of the IMSF_Header value.
3. The size of the GROUP_FILTER structure must be equal to or greater than the size of GF_Header value.
4. The size of the PartnerInfo structure must be equal to or greater than the PI_FIXED_SIZE value.

Parameter values returned to the application

RETARG

Returns an array whose size is based on the value in COMMAND. See [Table 19 on page 462](#) for information about REQARG and RETARG.

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See [Appendix B, “Socket call error return codes,” on page 743](#) for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value**Description****0**

Successful call.

-1Check **ERRNO** for an error code.

The COMMAND SIOGIFCONF returns a variable number of network interface configurations. [Figure 99 on page 464](#) contains an example of a COBOL II routine that can be used to work with such a structure.

Note: This call can be programmed only in languages that support address pointers. [Figure 99 on page 464](#) shows a COBOL II example for SIOCGIFCONF.

```

WORKING-STORAGE SECTION.
77  REQARG      PIC 9(8) COMP.
77  COUNT       PIC 9(8) COMP VALUE max number of interfaces.
LINKAGE SECTION.
01  RETARG.
    05  IOCTL-TABLE OCCURS 1 TO max TIMES DEPENDING ON COUNT.
        10  NAME      PIC X(16).
        10  FAMILY    PIC 9(4) BINARY.
        10  PORT      PIC 9(4) BINARY.
        10  ADDR      PIC 9(8) BINARY.
        10  NULLS     PIC X(8).
PROCEDURE DIVISION.
MULTIPLY COUNT BY 32 GIVING REQARG.
CALL 'EZASOKET' USING SOC-FUNCTION S COMMAND
REQARG RETARG ERRNO RETCODE.

```

Figure 99. COBOL II example for SIOCGIFCONF

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.

Note: The LISTEN call is not supported for datagram sockets or raw sockets.

The LISTEN call is typically used by a server to receive connection requests from clients. When a connection request is received, a new socket is created by a subsequent ACCEPT call, and the original socket continues to listen for additional connection requests. The LISTEN call converts an active socket to a passive socket and conditions it to accept connection requests from clients. If a socket becomes passive, it cannot initiate connection requests.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 100 on page 464 shows an example of LISTEN call instructions.

```

WORKING-STORAGE SECTION.
01  SOC-FUNCTION PIC X(16) VALUE IS 'LISTEN'.
01  S            PIC 9(4) BINARY.
01  BACKLOG      PIC 9(8) BINARY.
01  ERRNO        PIC 9(8) BINARY.
01  RETCODE      PIC S9(8) BINARY.

PROCEDURE DIVISION.
CALL 'EZASOKET' USING SOC-FUNCTION S BACKLOG ERRNO RETCODE.

```

Figure 100. LISTEN call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions”](#) on page 392.

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing LISTEN. The field is left-justified and padded to the right with blanks.

S

A halfword binary number set to the socket descriptor.

BACKLOG

A fullword binary number set to the number of communication requests to be queued.

Rule: The BACKLOG value specified on the LISTEN call is limited to the value configured by the SOMAXCONN statement in the stack's TCPIP PROFILE (default=10); no error is returned if a larger backlog is requested. SOMAXCONN might need to be updated if a larger backlog is desired. see [z/OS Communications Server: IP Configuration Reference](#) for details.

Parameter values returned to the application

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See [Appendix B, “Socket call error return codes,”](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check **ERRNO** for an error code.

NTOP

The NTOP call converts an IP address from its numeric binary form into a standard text presentation form. On successful completion, NTOP returns the converted IP address in the buffer provided.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 101 on page 466 shows an example of NTOP call instructions.

```

WORKING-STORAGE SECTION.
  01 SOC-ACCEPT-FUNCTION      PIC X(16)  VALUE IS 'ACCEPT'.
  01 SOC-NTOP-FUNCTION        PIC X(16)  VALUE IS 'NTOP'.
  01 S                        PIC 9(4)  BINARY.

* IPv4 socket structure.
  01 NAME.
    03 FAMILY      PIC 9(4)  BINARY.
    03 PORT        PIC 9(4)  BINARY.
    03 IP-ADDRESS  PIC 9(8)  BINARY.
    03 RESERVED    PIC X(8).

* IPv6 socket structure.
  01 NAME.
    03 FAMILY      PIC 9(4)  BINARY.
    03 PORT        PIC 9(4)  BINARY.
    03 FLOWINFO    PIC 9(8)  BINARY.
    03 IP-ADDRESS.
      10 FILLER    PIC 9(16) BINARY.
      10 FILLER    PIC 9(16) BINARY.
    03 SCOPE-ID    PIC 9(8)  BINARY.
  01 NTOP-FAMILY    PIC 9(8)  BINARY.
  01 ERRNO         PIC 9(8)  BINARY.
  01 RETCODE       PIC S9(8) BINARY.

  01 PRESENTABLE-ADDRESS      PIC X(45).
  01 PRESENTABLE-ADDRESS-LEN PIC 9(4)  BINARY.

PROCEDURE DIVISION.

  CALL 'EZASOKET' USING SOC-ACCEPT-FUNCTION S NAME
  ERRNO RETCODE.
  CALL 'EZASOKET' USING SOC-NTOP-FUNCTION NTOP-FAMILY IP-ADDRESS
  PRESENTABLE-ADDRESS
  PRESENTABLE-ADDRESS-LEN ERRNO RETURN-CODE.

```

Figure 101. NTOP call instruction example

Parameter values set by the application

Keyword

Description

FAMILY

The addressing family for the IP address being converted. The value of decimal 2 must be specified for AF_INET and 19 for AF_INET6.

IP-ADDRESS

A field containing the numeric binary form of the IPv4 or IPv6 address being converted. For an IPv4 address this field must be a fullword and for an IPv6 address this field must be 16 bytes. The address must be in network byte order.

Parameter values returned to the application

Keyword

Description

PRESENTABLE-ADDRESS

A field used to receive the standard text presentation form of the IPv4 or IPv6 address being converted. For IPv4 the address will be in dotted-decimal format and for IPv6 the address will be in colon-hex format. The size of the IPv4 address will be a maximum of 15 bytes and the size of the converted IPv6 address will be a maximum of 45 bytes. Consult the value returned in PRESENTABLE-ADDRESS-LEN for the actual length of the value in PRESENTABLE-ADDRESS.

PRESENTABLE-ADDRESS-LEN

Initially, an input parameter. The address of a binary halfword field that is used to specify the length of DSTADDR field on input and upon a successful return will contain the length of converted IP address.

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, ERRNO contains a valid error number. Otherwise, ignore the ERRNO field.

See Appendix B, “Socket call error return codes,” on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check ERRNO for an error code.

PTON

The PTON call converts an IP address in its standard text presentation form to its numeric binary form. On successful completion, PTON returns the converted IP address in the buffer provided.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

[Figure 102 on page 468](#) shows an example of PTON call instructions.

```

WORKING-STORAGE SECTION.
  01 SOC-BIND-FUNCTION      PIC X(16)  VALUE IS 'BIND'.
  01 SOC-PTON-FUNCTION      PIC X(16)  VALUE IS 'PTON'.
  01 S                      PIC 9(4)  BINARY.

* IPv4 socket structure.
  01 NAME.
    03 FAMILY              PIC 9(4)  BINARY.
    03 PORT                PIC 9(4)  BINARY.
    03 IP-ADDRESS          PIC 9(8)  BINARY.
    03 RESERVED            PIC X(8).

* IPv6 socket structure.
  01 NAME.
    03 FAMILY              PIC 9(4)  BINARY.
    03 PORT                PIC 9(4)  BINARY.
    03 FLOWINFO            PIC 9(8)  BINARY.
    03 IP-ADDRESS.
      10 FILLER            PIC 9(16) BINARY.
      10 FILLER            PIC 9(16) BINARY.
    03 SCOPE-ID            PIC 9(8)  BINARY.

  01 AF-INET               PIC 9(8)  BINARY VALUE 2.
  01 AF_INET6              PIC 9(8)  BINARY VALUE 19.

* IPv4 address.
  01 PRESENTABLE-ADDRESS    PIC X(45).
  01 PRESENTABLE-ADDRESS-IPV4 REDEFINES PRESENTABLE-ADDRESS.
    05 PRESENTABLE-IPV4-ADDRESS PIC X(15) VALUE '192.26.5.19'.
    05 FILLER               PIC X(30).
  01 PRESENTABLE-ADDRESS-LEN PIC 9(4)  BINARY VALUE 11.

* IPv6 address.
  01 PRESENTABLE-ADDRESS    PIC X(45)
    VALUE '12f9:0:0:c30:123:457:9cb:1112'.
  01 PRESENTABLE-ADDRESS-LEN PIC 9(4)  BINARY VALUE 29.

* IPv4-mapped IPv6 address.
  01 PRESENTABLE-ADDRESS    PIC X(45)
    VALUE '12f9:0:0:c30:123:457:192.26.5.19'.
  01 PRESENTABLE-ADDRESS-LEN PIC 9(4)  BINARY VALUE 32.

  01 ERRNO                  PIC 9(8)  BINARY.
  01 RETCODE                PIC S9(8) BINARY.

PROCEDURE DIVISION.

* IPv4 address.
  CALL 'EZASOKET' USING SOC-PTON-FUNCTION AF-INET PRESENTABLE-ADDRESS
    PRESENTABLE-ADDRESS-LEN IP-ADDRESS ERRNO RETURN-CODE.

* IPv6 address.
  CALL 'EZASOKET' USING SOC-PTON-FUNCTION AF_INET6 PRESENTABLE-ADDRESS
    PRESENTABLE-ADDRESS-LEN IP-ADDRESS ERRNO RETURN-CODE.
  CALL 'EZASOKET' USING SOC-BIND-FUNCTION S NAME ERRNO RETURN-CODE.

```

Figure 102. PTON call instruction example

Parameter values set by the application

Keyword

Description

FAMILY

The addressing family for the IP address being converted. The value of decimal 2 must be specified for AF_INET and 19 for AF_INET6.

PRESENTABLE-ADDRESS

A field containing the standard text presentation form of the IPv4 or IPv6 address being converted. For IPv4 the address will be in dotted-decimal format and for IPv6 the address will be in colon-hex format.

PRESENTABLE-ADDRESS-LEN

Input parameter. The address of a binary halfword field that must contain the length of the IP address to be converted.

Parameter values returned to the application

Keyword

Description

IP-ADDRESS

A field containing the numeric binary form of the IPv4 or IPv6 address being converted. For an IPv4 address this field must be a fullword and for an IPv6 address this field must be 16 bytes. The address must be in network byte order.

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, ERRNO contains a valid error number. Otherwise, ignore the ERRNO field.

See [Appendix B, “Socket call error return codes,”](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check ERRNO for an error code.

READ

The READ call reads the data on socket *s*. This is the conventional TCP/IP read data operation. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard extra bytes.

For stream sockets, data is processed as streams of information with no boundaries separating the data. For example, if programs A and B are connected with a stream socket and program A sends 1000 bytes, each call to this function can return any number of bytes, up to the entire 1000 bytes. The number of bytes returned will be contained in RETCODE. Therefore, programs using stream sockets should place this call in a loop that repeats until all data has been received.

Note: See [“EZACIC05 ”](#) on page 524 for a subroutine that will translate ASCII input data to EBCDIC.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

[Figure 103 on page 470](#) shows an example of READ call instructions.

```

WORKING-STORAGE SECTION.
01  SOC-FUNCTION      PIC X(16)  VALUE IS 'READ'.
01  S                  PIC 9(4)  BINARY.
01  NBYTE              PIC 9(8)  BINARY.
01  BUF                PIC X(length of buffer).
01  ERRNO              PIC 9(8)  BINARY.
01  RETCODE            PIC S9(8) BINARY.

PROCEDURE DIVISION.
    CALL 'EZASOCKET' USING SOC-FUNCTION S NBYTE BUF
                        ERRNO RETCODE.

```

Figure 103. READ call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions”](#) on page 392.

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing READ. The field is left-justified and padded to the right with blanks.

S

A halfword binary number set to the socket descriptor of the socket that is going to read the data.

NBYTE

A fullword binary number set to the size of BUF. READ does not return more than the number of bytes of data in NBYTE even if more data is available.

Parameter values returned to the application

BUF

On input, a buffer to be filled by completion of the call. The length of BUF must be at least as long as the value of NBYTE.

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See [Appendix B, “Socket call error return codes,”](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

0

A 0 return code indicates that the connection is closed and no data is available.

>0

A positive value indicates the number of bytes copied into the buffer.

-1

Check **ERRNO** for an error code.

READV

The READV function reads data on a socket and stores it in a set of buffers. If a datagram packet is too long to fit in the supplied buffers, datagram sockets discard extra bytes.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.

Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 104 on page 471 shows an example of READV call instructions.

```

WORKING-STORAGE SECTION.
01 SOC-FUNCTION          PIC X(16) VALUE 'READV'.
01 S                     PIC 9(4)  BINARY.
01 IOVCNT                PIC 9(8)  BINARY.

01 IOV.
    03 BUFFER-ENTRY OCCURS N TIMES.
        05 BUFFER-POINTER  USAGE IS POINTER.
        05 RESERVED        PIC X(4).
        05 BUFFER_LENGTH   PIC 9(8) BINARY.

01 ERRNO                 PIC 9(8) BINARY.
01 RETCODE               PIC 9(8) BINARY.

PROCEDURE DIVISION.
SET BUFFER-POINTER(1) TO ADDRESS OF BUFFER1.
SET BUFFER-LENGTH(1) TO LENGTH OF BUFFER1.
SET BUFFER-POINTER(2) TO ADDRESS OF BUFFER2.
SET BUFFER-LENGTH(2) TO LENGTH OF BUFFER2.
" " " " " "
" " " " " "
SET BUFFER-POINTER(n) TO ADDRESS OF BUFFERn.
SET BUFFER-LENGTH(n) TO LENGTH OF BUFFERn.
Call 'EZASOCKET' USING SOC-FUNCTION S IOV IOVCNT ERRNO RETCODE.

```

Figure 104. READV call instruction example

For equivalent PL/I and assembler language declarations, see ["Converting parameter descriptions"](#) on page 392.

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing READV. The field is left-justified and padded to the right with blanks.

S

A value or the address of a halfword binary number specifying the descriptor of the socket into which the data is to be read.

IOV

An array of tripleword structures with the number of structures equal to the value in IOVCNT and the format of the structures as follows:

Fullword 1

Pointer to the address of a data buffer, which is completed on completion of the call

Fullword 2

Reserved

Fullword 3

The length of the data buffer referenced in fullword one

IOVCNT

A fullword binary field specifying the number of data buffers provided for this call.

Parameter values returned to the application

ERRNO

A fullword binary field. If RETCODE is negative, this contains an error number. See [Appendix B, “Socket call error return codes,”](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

0

A 0 return code indicates that the connection is closed and no data is available.

>0

A positive value indicates the number of bytes copied into the buffer.

-1

Check **ERRNO** for an error code.

RECV

The RECV call, like READ, receives data on a socket with descriptor S. RECV applies only to connected sockets. If a datagram packet is too long to fit in the supplied buffers, datagram sockets discard extra bytes.

For additional control of the incoming data, RECV can:

- Peek at the incoming message without having it removed from the buffer
- Read out-of-band data

For stream sockets, data is processed as streams of information with no boundaries separating the data. For example, if programs A and B are connected with a stream socket and program A sends 1000 bytes, each call to this function can return any number of bytes, up to the entire 1000 bytes. The number of bytes returned will be contained in RETCODE. Therefore, programs using stream sockets should place RECV in a loop that repeats until all data has been received.

If data is not available for the socket, and the socket is in blocking mode, RECV blocks the caller until data arrives. If data is not available and the socket is in nonblocking mode, RECV returns a -1 and sets ERRNO to 35 (EWOULDBLOCK). See [“FCNTL” on page 404](#) or [“IOCTL” on page 455](#) for a description of how to set nonblocking mode.

For raw sockets, RECV adds a 20-byte header.

Note: See [“EZACIC05 ” on page 524](#) for a subroutine that will translate ASCII input data to EBCDIC.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389 .
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.

Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 105 on page 473 shows an example of RECV call instructions.

```

WORKING-STORAGE SECTION.
01 SOC-FUNCTION    PIC X(16)  VALUE IS 'RECV'.
01 S              PIC 9(4)  BINARY.
01 FLAGS          PIC 9(8)  BINARY.
   88 NO-FLAG      VALUE IS 0.
   88 OOB          VALUE IS 1.
   88 PEEK         VALUE IS 2.
01 NBYTE          PIC 9(8)  BINARY.
01 BUF            PIC X(length of buffer).
01 ERRNO          PIC 9(8)  BINARY.
01 RETCODE        PIC S9(8)  BINARY.

PROCEDURE DIVISION.
    CALL 'EZASOKET' USING SOC-FUNCTION S FLAGS NBYTE BUF
                        ERRNO RETCODE.

```

Figure 105. RECV call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions” on page 392](#).

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing RECV. The field is left-justified and padded to the right with blanks.

S

A halfword binary number set to the socket descriptor of the socket to receive the data.

FLAGS

A fullword binary field with values as follows:

Literal Value	Binary Value	Description
NO-FLAG	X'00000000'	Read data.
MSG-OOB	X'00000001'	Receive out-of-band data (stream sockets only). Even if the OOB flag is not set, out-of-band data can be read if the SO_OOBLIN option is set for the socket.
MSG-PEEK	X'00000002'	Peek at the data, but do not destroy data. If the peek flag is set, the next receive operation reads the same data.
MSG-WAITALL	X'00000040'	Requests that the function block until the full amount of data that was requested can be returned (stream sockets only). The function might return a smaller amount of data if the connection is closed, if an error is pending, or if the SO_RCVTIMEO field is set and the timer has expired for the socket.

NBYTE

A value or the address of a fullword binary number set to the size of BUF. RECV does not receive more than the number of bytes of data in NBYTE even if more data is available.

Parameter values returned to the application

BUF

The input buffer to receive the data.

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See [Appendix B, “Socket call error return codes,”](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

0

The socket is closed.

>0

A positive return code indicates the number of bytes copied into the buffer.

-1

Check **ERRNO** for an error code.

RECVFROM

The RECVFROM call receives data on a socket with descriptor S and stores it in a buffer. The RECVFROM call applies to both connected and unconnected sockets. The socket address is returned in the NAME structure. If a datagram packet is too long to fit in the supplied buffers, datagram sockets discard extra bytes.

For datagram protocols, RECVFROM returns the source address associated with each incoming datagram. For connection-oriented protocols like TCP, GETPEERNAME returns the address associated with the other end of the connection.

If NAME is nonzero, the call returns the address of the sender. The NBYTE parameter should be set to the size of the buffer.

On return, NBYTE contains the number of data bytes received.

For stream sockets, data is processed as streams of information with no boundaries separating the data. For example, if programs A and B are connected with a stream socket and program A sends 1000 bytes, each call to this function can return any number of bytes, up to the entire 1000 bytes. The number of bytes returned will be contained in RETCODE. Therefore, programs using stream sockets should place RECVFROM in a loop that repeats until all data has been received.

For raw sockets, RECVFROM adds a 20-byte header.

If data is not available for the socket, and the socket is in blocking mode, RECVFROM blocks the caller until data arrives. If data is not available and the socket is in nonblocking mode, RECVFROM returns a -1 and sets ERRNO to 35 (EWOULDBLOCK). See [“FCNTL”](#) on page 404 or [“IOCTL”](#) on page 455 for a description of how to set nonblocking mode.

Note: See [“EZACIC05 ”](#) on page 524 for a subroutine that will translate ASCII input data to EBCDIC.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.

Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 106 on page 475 shows an example of RECVFROM call instructions.

```

WORKING-STORAGE SECTION.
  01 SOC-FUNCTION    PIC X(16)  VALUE IS 'RECVFROM'.
  01 S               PIC 9(4)  BINARY.
  01 FLAGS           PIC 9(8)  BINARY.
      88 NO-FLAG      VALUE IS 0.
      88 OOB          VALUE IS 1.
      88 PEEK         VALUE IS 2.
  01 NBYTE           PIC 9(8)  BINARY.
  01 BUF             PIC X(length of buffer).

* IPv4 socket address structure.
  01 NAME.
      03 FAMILY       PIC 9(4)  BINARY.
      03 PORT         PIC 9(4)  BINARY.
      03 IP-ADDRESS   PIC 9(8)  BINARY.
      03 RESERVED     PIC X(8).

* IPv6 socket address structure.
  01 NAME.
      03 FAMILY       PIC 9(4)  BINARY.
      03 PORT         PIC 9(4)  BINARY.
      03 FLOWINFO     PIC 9(8)  BINARY.
      03 IP-ADDRESS.
          10 FILLER    PIC 9(16) BINARY.
          10 FILLER    PIC 9(16) BINARY.
      03 SCOPE-ID     PIC 9(8)  BINARY.

  01 ERRNO           PIC 9(8)  BINARY.
  01 RETCODE         PIC S9(8)  BINARY.

PROCEDURE DIVISION.
  CALL 'EZASOKET' USING SOC-FUNCTION S FLAGS
                      NBYTE BUF NAME ERRNO RETCODE.

```

Figure 106. RECVFROM call instruction example

For equivalent PL/I and assembler language declarations, see ["Converting parameter descriptions"](#) on page 392.

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing RECVFROM. The field is left-justified and padded to the right with blanks.

S

A halfword binary number set to the socket descriptor of the socket to receive the data.

FLAGS

A fullword binary field containing flag values as follows:

Literal Value	Binary Value	Description
NO-FLAG	X'00000000'	Read data.

Literal Value	Binary Value	Description
MSG-OOB	X'00000001'	Receive out-of-band data (stream sockets only). Even if the OOB flag is not set, out-of-band data can be read if the SO_OOBLINE option is set for the socket.
MSG-PEEK	X'00000002'	Peek at the data, but do not destroy data. If the peek flag is set, the next receive operation reads the same data.
MSG-WAITALL	X'00000040'	Requests that the function block until the requested amount of data can be returned (stream sockets only). The function might return a smaller amount of data if the connection is closed, if an error is pending, or if the SO_RCVTIMEO field is set and the timer has expired for the socket.

NBYTE

A fullword binary number specifying the length of the input buffer.

Parameter values returned to the application

BUF

Defines an input buffer to receive the input data.

NAME

An IPv4 socket address structure containing the address of the socket that sent the data. The structure is as follows:

FAMILY

A halfword binary number specifying the IPv4 addressing family. The value is always decimal 2, indicating AF_INET.

PORT

A halfword binary number specifying the port number of the sending socket.

IP-ADDRESS

A fullword binary number specifying the 32-bit IPv4 IP address of the sending socket.

RESERVED

An 8-byte reserved field. This field is required, but is not used.

An IPv6 socket address structure containing the address of the socket that sent the data. The structure is as follows:

Field

Description

FAMILY

A halfword binary number specifying the IPv6 addressing family. The value is decimal 19, indicating AF_INET6.

PORT

A halfword binary number specifying the port number of the sending socket.

FLOWINFO

A fullword binary field specifying the traffic class and flow label. This value of this field is undefined.

IP-ADDRESS

A 16-byte binary field set to the 128-bit IPv6 IP address of the sending socket.

SCOPE-ID

A fullword binary field which identifies a set of interfaces as appropriate for the scope of the address carried in the IPv6-ADDRESS field. For a link scope IPv6-ADDRESS, SCOPE-ID contains the link index for the IPv6-ADDRESS. For all other address scopes, SCOPE-ID is undefined.

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See [Appendix B, “Socket call error return codes,”](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

0

The socket is closed.

>0

A positive return code indicates the number of bytes of data transferred by the read call.

-1

Check **ERRNO** for an error code.

RECVMSG

The RECVMSG call receives messages on a socket with descriptor S and stores them in an array of message headers. If a datagram packet is too long to fit in the supplied buffers, datagram sockets discard extra bytes.

For datagram protocols, RECVMSG returns the source address associated with each incoming datagram. For connection-oriented protocols like TCP, GETPEERNAME returns the address associated with the other end of the connection.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

[Figure 107 on page 478](#) shows an example of RECVMSG call instructions.

```
WORKING-STORAGE SECTION.
01 SOC-FUNCTION      PIC X(16)  VALUE IS 'RECVMSG'.
01 S                 PIC 9(4)   BINARY.
01 MSG-HDR.
03 MSG-NAME          USAGE IS POINTER.
03 MSG-NAME-LEN      PIC 9(8)  COMP.
03 IOV               USAGE IS POINTER.
03 IOVCNT            USAGE IS POINTER.
03 MSG-ACCRIGHTS     USAGE IS POINTER.
03 MSG-ACCRIGHTS-LEN USAGE IS POINTER.
```

```

01  FLAGS          PIC 9(8)  BINARY.
    88 NO-FLAG          VALUE IS 0.
    88 OOB             VALUE IS 1.
    88 PEEK            VALUE IS 2.
01  ERRNO          PIC 9(8)  BINARY.
01  RETCODE        PIC S9(8) BINARY.

LINKAGE SECTION.
01  L1.
    03 RECVMSG-IOVECTOR.
        05 IOV1A          USAGE IS POINTER.
        05 IOV1AL         PIC 9(8) COMP.
        05 IOV1L         PIC 9(8) COMP.
        05 IOV2A          USAGE IS POINTER.
        05 IOV2AL         PIC 9(8) COMP.
        05 IOV2L         PIC 9(8) COMP.
        05 IOV3A          USAGE IS POINTER.
        05 IOV3AL         PIC 9(8) COMP.
        05 IOV3L         PIC 9(8) COMP.

    03 RECVMSG-BUFFER1    PIC X(16).
    03 RECVMSG-BUFFER2    PIC X(16).
    03 RECVMSG-BUFFER3    PIC X(16).
    03 RECVMSG-BUFNO      PIC 9(8) COMP.

* IPv4 socket address structure.
03  NAME.
    05 FAMILY            PIC 9(4) BINARY.
    05 PORT              PIC 9(4) BINARY.
    05 IP-ADDRESS        PIC 9(8) BINARY.
    05 RESERVED          PIC X(8).

* IPv6 socket address structure.
03  NAME.
    05 FAMILY            PIC 9(4) BINARY.
    05 PORT              PIC 9(4) BINARY.
    53 FLOWINFO          PIC 9(8) BINARY.
    05 IP-ADDRESS.
        10 FILLER        PIC 9(16) BINARY.
        10 FILLER        PIC 9(16) BINARY.
    05 SCOPE-ID          PIC 9(8) BINARY.

```

PROCEDURE DIVISION USING L1.

```

SET MSG-NAME TO ADDRESS OF NAME.
MOVE LENGTH OF NAME TO MSG-NAME-LEN.
SET IOV TO ADDRESS OF RECVMSG-IOVECTOR.
MOVE 3 TO RECVMSG-BUFNO.
SET IOVCNT TO ADDRESS OF RECVMSG-BUFNO.
SET IOV1A TO ADDRESS OF RECVMSG-BUFFER1.
MOVE 0 TO IOV1AL.
MOVE LENGTH OF RECVMSG-BUFFER1 TO IOV1L.
SET IOV2A TO ADDRESS OF RECVMSG-BUFFER2.
MOVE 0 TO IOV2AL.
MOVE LENGTH OF RECVMSG-BUFFER2 TO IOV2L.
SET IOV3A TO ADDRESS OF RECVMSG-BUFFER3.
MOVE 0 TO IOV3AL.
MOVE LENGTH OF RECVMSG-BUFFER3 TO IOV3L.
SET MSG-ACCRIGHTS TO NULLS.
SET MSG-ACCRIGHTS-LEN TO NULLS.
MOVE 0 TO FLAGS.
MOVE SPACES TO RECVMSG-BUFFER1.
MOVE SPACES TO RECVMSG-BUFFER2.
MOVE SPACES TO RECVMSG-BUFFER3.

```

```
CALL 'EZASOKET' USING SOC-FUNCTION S MSG-HDR FLAGS ERRNO RETCODE.
```

Figure 107. RECVMSG call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions”](#) on [page 392](#).

Parameter values set by the application

S

A value or the address of a halfword binary number specifying the socket descriptor.

MSG

On input, a pointer to a message header into which the message is received upon completion of the call.

Field

Description

NAME

On input, a pointer to a buffer where the sender address is stored upon completion of the call. The storage being pointed to should be for an IPv4 socket address or an IPv6 socket address. The IPv4 socket address structure contains the following fields:

Field

Description

FAMILY

Output parameter. A halfword binary number specifying the IPv4 addressing family. The value for IPv4 socket descriptor (S parameter) is decimal 2, indicating AF_INET.

PORT

Output parameter. A halfword binary number specifying the port number of the sending socket.

IP-ADDRESS

Output parameter. A fullword binary number specifying the 32-bit IPv4 IP address of the sending socket.

RESERVED

Output parameter. An 8-byte reserved field. This field is required, but is not used.

The IPv6 socket address structure contains the following fields:

Field

Description

FAMILY

Output parameter. A halfword binary number specifying the IPv6 addressing family. The value for IPv6 socket descriptor (S parameter) is decimal 19, indicating AF_INET6.

PORT

Output parameter. A halfword binary number specifying the port number of the sending socket.

FLOWINFO

A fullword binary field specifying the traffic class and flow label. This value of this field is undefined.

IP-ADDRESS

Output parameter. A 16 byte binary field specifying the 128-bit IPv6 IP address, in network byte order, of the sending socket.

SCOPE-ID

A fullword binary field which identifies a set of interfaces as appropriate for the scope of the address carried in the IPv6-ADDRESS field. For a link scope IPv6-ADDRESS, SCOPE-ID contains the link index for the IPv6-ADDRESS. For all other address scopes, SCOPE-ID is undefined.

NAME-LEN

On input, a pointer to the size of the NAME.

IOV

On input, a pointer to an array of tripleword structures with the number of structures equal to the value in IOVCNT and the format of the structures as follows:

Fullword 1

A pointer to the address of a data buffer. This data buffer must be in the home address space.

Fullword 2

Reserved. This storage will be cleared.

Fullword 3

A pointer to the length of the data buffer referenced in fullword 1.

In COBOL, the IOV structure must be defined separately in the Linkage section, as shown in the example.

IOVCNT

On input, a pointer to a fullword binary field specifying the number of data buffers provided for this call.

ACCRIGHTS

On input, a pointer to the access rights received. This field is ignored.

ACCRLEN

On input, a pointer to the length of the access rights received. This field is ignored.

FLAGS

A fullword binary field with values as follows:

Literal Value	Binary Value	Description
NO-FLAG	X'00000000'	Read data.
MSG-OOB	X'00000001'	Receive out-of-band data (stream sockets only). Even if the OOB flag is not set, out-of-band data can be read if the SO-OOBINLINE option is set for the socket.
MSG-PEEK	X'00000002'	Peek at the data, but do not destroy data. If the peek flag is set, the next receive operation reads the same data.
MSG-WAITALL	X'00000040'	Requests that the function block until the requested amount of data can be returned (stream sockets only). The function might return a smaller amount of data if the connection is closed, if an error is pending, or if the SO_RCVTIMEO field is set and the timer has expired for the socket.

Parameter values returned to the application**ERRNO**

A fullword binary field. If RETCODE is negative, this contains an error number. See [Appendix B, "Socket call error return codes," on page 743](#) for information about ERRNO return codes.

RETCODE

A fullword binary field with the following values:

Value**Description**

<0

Call returned error. See ERRNO field.

0

Connection partner has closed connection.

>0

Number of bytes read.

SELECT

In a process where multiple I/O operations can occur it is necessary for the program to be able to wait on one or several of the operations to complete. For example, consider a program that issues a READ to multiple sockets whose blocking mode is set. Because the socket would block on a READ call, only one socket could be read at a time. Setting the sockets nonblocking would solve this problem, but would require polling each socket repeatedly until data became available. The SELECT call allows you to test several sockets and to execute a subsequent I/O call only when one of the tested sockets is ready, thereby ensuring that the I/O call will not block.

To use the SELECT call as a timer in your program, take one of the following actions:

- Set the read, write, and exception arrays to zeros.
- Specify MAXSOC <= 0.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Defining which sockets to test

The SELECT call monitors for read operations, write operations, and exception operations:

- When a socket is ready to read, one of the following situations has occurred:
 - A buffer for the specified sockets contains input data. If input data is available for a given socket, a read operation on that socket will not block.
 - A connection has been requested on that socket.
- When a socket is ready to write, TCP/IP can accommodate additional output data. If TCP/IP can accept additional output for a given socket, a write operation on that socket will not block.
- When an exception condition has occurred on a specified socket it is an indication that a TAKESOCKET has occurred for that socket.
- A timeout occurs on the SELECT call. The timeout period can be specified when the SELECT call is issued.

Each socket descriptor is represented by a bit in a bit string. The length of this bit-mask array is dependent on the value of the MAXSOC parameter and must be a multiple of 4 bytes.

See ["Selecting requests in a concurrent server program"](#) on page 37 for more information.

Note: To simplify string processing in COBOL, you can use the program EZACIC06 to convert each bit in the string to a character. For more information, see ["EZACIC06 "](#) on page 525.

Read operations

Read operations include ACCEPT, READ, READV, RECV, RECVFROM, or RECVMSG calls. A socket is ready to be read when data has been received for it or when a connection request has occurred.

To test whether any of several sockets is ready for reading, set the appropriate bits in RSNDSK to one before issuing the SELECT call. When the SELECT call returns, the corresponding bits in the RRETMSK indicate sockets are ready for reading.

Write operations

A socket is selected for writing (ready to be written) when:

- TCP/IP can accept additional outgoing data.
- The socket is marked nonblocking and a previous CONNECT did not complete immediately. In this case, CONNECT returned an ERRNO with a value of 36 (EINPROGRESS). This socket will be selected for write when the CONNECT completes.

A call to WRITE, SEND, or SENDTO blocks when the amount of data to be sent exceeds the amount of data TCP/IP can accept. To avoid this, you can precede the write operation with a SELECT call to ensure that the socket is ready for writing. When a socket is selected for WRITE, the program can determine the amount of TCP/IP buffer space available by issuing the GETSOCKOPT call with the SO-SNDBUF option.

To test whether any of several sockets is ready for writing, set the WSNDSK bits representing those sockets to 1 before issuing the SELECT call. When the SELECT call returns, the corresponding bits in the WRETMSK indicate sockets are ready for writing.

Exception operations

For each socket to be tested, the SELECT call can check for an existing exception condition. Two exception conditions are supported:

- The calling program (concurrent server) has issued a GIVESOCKET command and the target child server has successfully issued the TAKESOCKET call. When this condition is selected, the calling program (concurrent server) should issue CLOSE to dissociate itself from the socket.
- A socket has received out-of-band data. On this condition, a READ will return the out-of-band data ahead of program data.

To test whether any of several sockets have an exception condition, set the ESNDSK bits representing those sockets to 1. When the SELECT call returns, the corresponding bits in the ERETMSK indicate sockets with exception conditions.

MAXSOC parameter

The SELECT call must test each bit in each string before returning results. For efficiency, the MAXSOC parameter can be used to specify the largest socket descriptor number that needs to be tested for any event type. The SELECT call tests only bits that are in the range 0 through the MAXSOC value minus 1.

Example: If MAXSOC is set to 50, the range would be 0 through 49.

TIMEOUT parameter

If the time specified in the TIMEOUT parameter elapses before any event is detected, the SELECT call returns, and the RETCODE is set to 0.

[Figure 108 on page 483](#) shows an example of SELECT call instructions.

```

WORKING-STORAGE SECTION.
01 SOC-FUNCTION      PIC X(16) VALUE IS 'SELECT'.
01 MAXSOC            PIC 9(8) BINARY.
01 TIMEOUT.
03 TIMEOUT-SECONDS  PIC 9(8) BINARY.
03 TIMEOUT-MICROSEC PIC 9(8) BINARY.
01 RSNDMSK          PIC X(*).
01 WSNDMSK          PIC X(*).
01 ESNDMSK          PIC X(*).
01 RRETMSK          PIC X(*).
01 WRETMSK          PIC X(*).
01 ERETMSK          PIC X(*).
01 ERRNO            PIC 9(8) BINARY.
01 RETCODE          PIC S9(8) BINARY.

PROCEDURE DIVISION.
    CALL 'EZASOKET' USING SOC-FUNCTION MAXSOC TIMEOUT
                        RSNDMSK WSNDMSK ESNDMSK
                        RRETMSK WRETMSK ERETMSK
                        ERRNO RETCODE.

```

* The bit mask lengths can be determined from the expression:

```
((maximum socket number +32)/32 (drop the remainder))*4
```

Figure 108. SELECT call instruction example

Bit masks are 32-bit fullwords with one bit for each socket. Up to 32 sockets fit into one 32-bit mask [PIC X(4)]. If you have 33 sockets, you must allocate two 32-bit masks [PIC X(8)].

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions” on page 392](#).

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing SELECT. The field is left-justified and padded on the right with blanks.

MAXSOC

A fullword binary field that specifies the largest socket descriptor value that is being checked. The SELECT call tests only bits that are in the range 0 through the MAXSOC value minus 1. For example, if you set the MAXSOC value to 50, the range is 0 – 49.

TIMEOUT

If TIMEOUT is a positive value, it specifies the maximum interval to wait for the selection to complete. If TIMEOUT-SECONDS is a negative value, the SELECT call blocks until a socket becomes ready. To poll the sockets and return immediately, specify the TIMEOUT value to be 0.

TIMEOUT is specified in the two-word TIMEOUT as follows:

- TIMEOUT-SECONDS, word one of the TIMEOUT field, is the seconds component of the timeout value.
- TIMEOUT-MICROSEC, word two of the TIMEOUT field, is the microseconds component of the timeout value (0–999999).

For example, if you want SELECT to time out after 3.5 seconds, set TIMEOUT-SECONDS to 3 and TIMEOUT-MICROSEC to 500000.

RSNDMSK

A bit string sent to request read event status.

- For each socket to be checked for pending read events, the corresponding bit in the string should be set to 1.
- For sockets to be ignored, the value of the corresponding bit should be set to 0.

If this parameter is set to all zeros, the SELECT will not check for read events.

WSNDMSK

A bit string sent to request write event status.

- For each socket to be checked for pending write events, the corresponding bit in the string should be set to 1.
- For sockets to be ignored, the value of the corresponding bit should be set to 0.

If this parameter is set to all zeros, the SELECT will not check for write events.

ESNDMSK

A bit string sent to request exception event status.

- For each socket to be checked for pending exception events, the corresponding bit in the string should be set to 1.
- For each socket to be ignored, the corresponding bit should be set to 0.

If this parameter is set to all zeros, the SELECT will not check for exception events.

Parameter values returned to the application

RRETMSK

A bit string returned with the status of read events. The length of the string should be equal to the maximum number of sockets to be checked. For each socket that is ready to read, the corresponding bit in the string will be set to 1; bits that represent sockets that are not ready to read will be set to 0.

WRETMSK

A bit string returned with the status of write events. The length of the string should be equal to the maximum number of sockets to be checked. For each socket that is ready to write, the corresponding bit in the string will be set to 1; bits that represent sockets that are not ready to be written will be set to 0.

ERETMSK

A bit string returned with the status of exception events. The length of the string should be equal to the maximum number of sockets to be checked. For each socket that has an exception status, the corresponding bit will be set to 1; bits that represent sockets that do not have exception status will be set to 0.

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See [Appendix B, “Socket call error return codes,”](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value	Description
-------	-------------

>0	Indicates the sum of all ready sockets in the three masks.
0	Indicates that the SELECT time limit has expired.
-1	Check ERRNO for an error code.

SELECTEX

The SELECTEX call monitors a set of sockets, a time value, and an ECB. It completes when either one of the sockets has activity, the time value expires, or one of the ECBs is posted.

To use the SELECTEX call as a timer in your program, take either of the following actions:

- Set the read, write, and exception arrays to zeros.
- Specify MAXSOC ≤ 0.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

[Figure 109 on page 486](#) shows an example of SELECTEX call instructions.

If an application intends to pass a single ECB on the SELECTEX call, then the corresponding working storage definitions and CALL instruction should be coded as below:

```
WORKING-STORAGE SECTION.
01 SOC-FUNCTION PIC X(16) VALUE IS 'SELECTEX'.
01 MAXSOC PIC 9(8) BINARY.
01 TIMEOUT.
03 TIMEOUT-SECONDS PIC 9(8) BINARY.
03 TIMEOUT-MINUTES PIC 9(8) BINARY.
01 RSNDMSK PIC X(*).
01 WSNDMSK PIC X(*).
01 ESNDMSK PIC X(*).
01 RRETMASK PIC X(*).
01 WRETMASK PIC X(*).
01 ERETMASK PIC X(*).
01 SELECB PIC X(4).
01 ERRNO PIC 9(8) BINARY.
01 RETCODE PIC S9(8) BINARY.
```

Where * is the size of the select mask

```
PROCEDURE DIVISION.
CALL 'EZASOKET' USING SOC-FUNCTION MAXSOC TIMEOUT
RSNDMSK WSNDMSK ESNDMSK
RRETMASK WRETMASK ERETMASK
SELECB ERRNO RETCODE.
```

However, if the application intends to pass the address of an ECB list on the SELECTEX call, then the application must set the high order bit in the ECB list address and pass that address using the BY VALUE option as documented in the following example. The remaining parameters must be set back to the default by specifying BY REFERENCE before ERRNO:

```
WORKING-STORAGE SECTION.
01 SOC-FUNCTION PIC X(16) VALUE IS 'SELECTEX'.
01 MAXSOC PIC 9(8) BINARY.
01 TIMEOUT.
03 TIMEOUT-SECONDS PIC 9(8) BINARY.
03 TIMEOUT-MINUTES PIC 9(8) BINARY.
01 RSNDMSK PIC X(*).
01 WSNDMSK PIC X(*).
01 ESNDMSK PIC X(*).
01 RRETMASK PIC X(*).
01 WRETMASK PIC X(*).
01 ERETMASK PIC X(*).
01 ECBLIST-PTR USAGE IS POINTER.
01 ERRNO PIC 9(8) BINARY.
01 RETCODE PIC S9(8) BINARY.
```

Where * is the size of the select mask

```
PROCEDURE DIVISION.
CALL 'EZASOKET' USING SOC-FUNCTION MAXSOC TIMEOUT
RSNDMSK WSNDMSK ESNDMSK
RRETMASK WRETMASK ERETMASK
BY VALUE ECBLIST-PTR
BY REFERENCE ERRNO RETCODE.
```

* The bit mask lengths can be determined from the expression:

```
((maximum socket number +32)/32 (drop the remainder))*4
```

Figure 109. SELECTEX call instruction example

Defining which sockets to test

The SELECTEX call monitors for read operations, write operations, and exception operations:

- When a socket is ready to read, one of the following situations has occurred:
 - A buffer for the specified sockets contains input data. If input data is available for a given socket, a read operation on that socket will not block.

- A connection has been requested on that socket.
- When a socket is ready to write, TCP/IP can accommodate additional output data. If TCP/IP can accept additional output for a given socket, a write operation on that socket will not block.
- When an exception condition has occurred on a specified socket it is an indication that a TAKESOCKET has occurred for that socket.
- A timeout occurs on the SELECTEX call. The timeout period can be specified when the SELECTEX call is issued.
- The ECB (or one of the ECBs in the ECB list) passed on the SELECTEX call has been posted.

Each socket descriptor is represented by a bit in a bit string. The length of this bit-mask array is dependent on the value of the MAXSOC parameter and must be a multiple of 4 bytes.

See [“Selecting requests in a concurrent server program” on page 37](#) for more information.

Note: To simplify string processing in COBOL, you can use the program EZACIC06 to convert each bit in the string to a character. For more information, see [“EZACIC06 ” on page 525](#).

Read operations

Read operations include ACCEPT, READ, READV, RECV, RECVFROM, or RECVMSG calls. A socket is ready to be read when data has been received for it or when a connection request has occurred.

To test whether any of several sockets is ready for reading, set the appropriate bits in RSNDSK to one before issuing the SELECTEX call. When the SELECTEX call returns, the corresponding bits in the RRETMSK indicate sockets are ready for reading.

Write operations

A socket is selected for writing (ready to be written) when:

- TCP/IP can accept additional outgoing data.
- The socket is marked nonblocking and a previous CONNECT did not complete immediately. In this case, CONNECT returned an ERRNO with a value of 36 (EINPROGRESS). This socket will be selected for write when the CONNECT completes.

A call to WRITE, SEND, or SENDTO blocks when the amount of data to be sent exceeds the amount of data TCP/IP can accept. To avoid this, you can precede the write operation with a SELECTEX call to ensure that the socket is ready for writing. When a socket is selected for WRITE, the program can determine the amount of TCP/IP buffer space available by issuing the GETSOCKOPT call with the SO-SNDBUF option.

To test whether any of several sockets is ready for writing, set the WSNDSK bits representing those sockets to 1 before issuing the SELECTEX call. When the SELECTEX call returns, the corresponding bits in the WRETMSK indicate sockets are ready for writing.

Exception operations

For each socket to be tested, the SELECTEX call can check for an existing exception condition. Two exception conditions are supported:

- The calling program (concurrent server) has issued a GIVESOCKET command and the target child server has successfully issued the TAKESOCKET call. When this condition is selected, the calling program (concurrent server) should issue CLOSE to dissociate itself from the socket.
- A socket has received out-of-band data. On this condition, a READ will return the out-of-band data ahead of program data.

To test whether any of several sockets have an exception condition, set the ESNDSK bits representing those sockets to 1. When the SELECTEX call returns, the corresponding bits in the ERETMSK indicate sockets with exception conditions.

MAXSOC parameter

The SELECTEX call must test each bit in each string before returning results. For efficiency, the MAXSOC parameter can be used to specify the largest socket descriptor number that needs to be tested for any event type. The SELECTEX call tests only bits that are in the range 0 through the MAXSOC value minus 1.

Example: If MAXSOC is set to 50, the range would be 0 through 49.

TIMEOUT parameter

If the time specified in the TIMEOUT parameter elapses before any event is detected, the SELECTEX call returns, and the RETCODE is set to 0.

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing SELECT. The field is left-justified and padded on the right with blanks.

MAXSOC

A fullword binary field that specifies the largest socket descriptor value that is being checked. The SELECTEX call tests only bits that are in the range 0 through the MAXSOC value minus 1. For example, if you set the MAXSOC value to 50, the range is 0 – 49.

TIMEOUT

If TIMEOUT is a positive value, it specifies a maximum interval to wait for the selection to complete. If TIMEOUT-SECONDS is a negative value, the SELECTEX call blocks until a socket becomes ready or an ECB or ECB in a list is posted. To poll the sockets and return immediately, set TIMEOUT to be zeros.

TIMEOUT is specified in the two-word TIMEOUT as follows:

- TIMEOUT-SECONDS, word one of the TIMEOUT field, is the seconds component of the timeout value.
- TIMEOUT-MICROSEC, word two of the TIMEOUT field, is the microseconds component of the timeout value (0–999999).

For example, if you want SELECTEX to time out after 3.5 seconds, set TIMEOUT-SECONDS to 3 and TIMEOUT-MICROSEC to 500000.

RSNDMSK

The bit-mask array to control checking for read interrupts. If this parameter is not specified or the specified bit-mask is zeros, the SELECT will not check for read interrupts. The length of this bit-mask array is dependent on the value in MAXSOC.

WSNDMSK

The bit-mask array to control checking for write interrupts. If this parameter is not specified or the specified bit-mask is zeros, the SELECT will not check for write interrupts. The length of this bit-mask array is dependent on the value in MAXSOC.

ESNDMSK

The bit-mask array to control checking for exception interrupts. If this parameter is not specified or the specified bit-mask is zeros, the SELECT will not check for exception interrupts. The length of this bit-mask array is dependent on the value in MAXSOC.

SELECB

An ECB which, if posted, causes completion of the SELECTEX.

ECBLIST-PTR

A pointer to an ECB list. The application must set the high order bit in the ECB list address and pass that address using the BY VALUE option. The remaining parameters must be set back to the default by specifying BY REFERENCE before ERRNO.

Parameter values returned to the application

ERRNO

A fullword binary field; if RETCODE is negative, this contains an error number. See [Appendix B, “Socket call error return codes,”](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field

Value

Meaning

>0

The number of ready sockets.

0

Either the SELECTEX time limit has expired (ECB value is 0) or one of the caller's ECBs has been posted (ECB value is nonzero and the caller's descriptor sets is set to 0). The caller must initialize the ECB values to 0 before issuing the SELECTEX socket command.

-1

Check **ERRNO** for an error code.

RRETMSK

The bit-mask array returned by the SELECT if RSNDSK is specified. The length of this bit-mask array is dependent on the value in MAXSOC.

WRETMSK

The bit-mask array returned by the SELECT if WSNDSK is specified. The length of this bit-mask array is dependent on the value in MAXSOC.

ERETMSK

The bit-mask array returned by the SELECT if ESNDSK is specified. The length of this bit-mask array is dependent on the value in MAXSOC.

SEND

The SEND call sends data on a specified connected socket.

The FLAGS field allows you to:

- Send out-of-band data, such as interrupts, aborts, and data marked urgent. Only stream sockets created in the AF_INET address family support out-of-band data.
- Suppress use of local routing tables. This implies that the caller takes control of routing and writing network software.

For datagram sockets, SEND transmits the entire datagram if it fits into the receiving buffer. Extra data is discarded.

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 call to this function can send any number of bytes, up to the entire 1000 bytes, with the number of bytes sent returned in RETCODE. Therefore, programs using stream sockets should place this call in a loop, reissuing the call until all data has been sent.

Note: See [“EZACIC04 ”](#) on page 523 for a subroutine that will translate EBCDIC input data to ASCII.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.

Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 110 on page 490 shows an example of SEND call instructions.

```

WORKING-STORAGE SECTION.
01 SOC-FUNCTION    PIC X(16)  VALUE IS 'SEND'.
01 S              PIC 9(4)  BINARY.
01 FLAGS          PIC 9(8)  BINARY.
   88 NO-FLAG      VALUE IS 0.
   88 OOB          VALUE IS 1.
   88 DONT-ROUTE   VALUE IS 4.
01 NBYTE          PIC 9(8)  BINARY.
01 BUF            PIC X(length of buffer).
01 ERRNO          PIC 9(8)  BINARY.
01 RETCODE        PIC S9(8)  BINARY.

PROCEDURE DIVISION.
    CALL 'EZASOKET' USING SOC-FUNCTION S FLAGS NBYTE
                        BUF ERRNO RETCODE.

```

Figure 110. SEND call instruction example

For equivalent PL/I and assembler language declarations, see ["Converting parameter descriptions"](#) on page 392.

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing SEND. The field is left-justified and padded on the right with blanks.

S

A halfword binary number specifying the socket descriptor of the socket that is sending data.

FLAGS

A fullword binary field with values as follows:

Literal Value	Binary Value	Description
NO-FLAG	X'00000000'	No flag is set. The command behaves like a WRITE call.
MSG-OOB	X'00000001'	Send out-of-band data. (Stream sockets only.) Even if the OOB flag is not set, out-of-band data can be read if the SO-OOBINLINE option is set for the socket.
MSG-DONTROUTE	X'00000004'	Do not route. Routing is provided by the calling program.

NBYTE

A fullword binary number set to the number of bytes of data to be transferred.

BUF

The buffer containing the data to be transmitted. BUF should be the size specified in NBYTE.

Parameter values returned to the application

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See [Appendix B, “Socket call error return codes,”](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

≥0

A successful call. The value is set to the number of bytes transmitted.

-1

Check **ERRNO** for an error code.

SENDMSG

The SENDMSG call sends messages on a socket with descriptor S passed in an array of messages.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

[Figure 111 on page 493](#) shows an example of SENDMSG call instructions.

```
WORKING-STORAGE SECTION.
01 SOC-FUNCTION      PIC X(16)  VALUE IS 'SENDMSG'.
01 S                 PIC 9(4)   BINARY.
01 MSG-HDR.
03 MSG-NAME          USAGE IS POINTER.
03 MSG-NAME-LEN      PIC 9(8)  BINARY.
03 IOV               USAGE IS POINTER.
03 IOVCNT            USAGE IS POINTER.
03 MSG-ACCRIGHTS     USAGE IS POINTER.
03 MSG-ACCRIGHTS-LEN USAGE IS POINTER.

01 FLAGS             PIC 9(8)   BINARY.
88 NO-FLAG           VALUE IS 0.
88 OOB               VALUE IS 1.
88 DONTRROUTE        VALUE IS 4.
01 ERRNO             PIC 9(8)   BINARY.
01 RETCODE           PIC S9(8)  BINARY.

01 SENDMSG-IPV4ADDR  PIC 9(8)  BINARY.
01 SENDMSG-IPV6ADDR.
```

```

05 FILLER          PIC9(16) BINARY.
05 FILLER          PIC9(16) BINARY.

LINKAGE SECTION.
01 L1.
03 SENDMSG-IOVECTOR.
05 IOV1A           USAGE IS POINTER.
05 IOV1AL          PIC 9(8) COMP.
05 IOV1L           PIC 9(8) COMP.
05 IOV2A           USAGE IS POINTER.
05 IOV2AL          PIC 9(8) COMP.
05 IOV2L           PIC 9(8) COMP.
05 IOV3A           USAGE IS POINTER.
05 IOV3AL          PIC 9(8) COMP.
05 IOV3L           PIC 9(8) COMP.

03 SENDMSG-BUFFER1 PIC X(16).
03 SENDMSG-BUFFER2 PIC X(16).
03 SENDMSG-BUFFER3 PIC X(16).
03 SENDMSG-BUFNO   PIC 9(8) COMP.

```

* IPv4 socket address structure.

```

03 NAME.
05 FAMILY      PIC 9(4) BINARY.
05 PORT        PIC 9(4) BINARY.
05 IP-ADDRESS  PIC 9(8) BINARY.
05 RESERVED    PIC X(8) BINARY.

```

* IPv6 socket address structure.

```

03 NAME.
05 FAMILY      PIC 9(4) BINARY.
05 PORT        PIC 9(4) BINARY.
05 FLOWINFO    PIC 9(8) BINARY.
05 IP-ADDRESS.
10 FILLER      PIC 9(16) BINARY.
10 FILLER      PIC 9(16) BINARY.
05 SCOPE-ID    PIC 9(8) BINARY.

```

PROCEDURE DIVISION USING L1.

* For IPv6.

```

MOVE 19 TO FAMILY.
MOVE 1234 TO PORT.
MOVE 0 TO FLOWINFO.
MOVE SENDMSG-IPV6ADDR TO IP-ADDRESS.
MOVE 0 TO SCOPE-ID.

```

* For IPv4.

```

MOVE 2 TO FAMILY.
MOVE 1234 TO PORT.
MOVE SENDMSG-IPV4ADDR TO IP-ADDRESS.

```

```

SET MSG-NAME TO ADDRESS OF NAME.
MOVE LENGTH OF NAME TO MSG-NAME-LEN.
SET IOV TO ADDRESS OF SENDMSG-IOVECTOR.
MOVE 3 TO SENDMSG-BUFNO.
SET MSG-IOVCNT TO ADDRESS OF SENDMSG-BUFNO.
SET IOV1A TO ADDRESS OF SENDMSG-BUFFER1.
MOVE 0 TO IOV1AL.
MOVE LENGTH OF SENDMSG-BUFFER1 TO IOV1L.
SET IOV2A TO ADDRESS OF SENDMSG-BUFFER2.
MOVE 0 TO IOV2AL.
MOVE LENGTH OF SENDMSG-BUFFER2 TO IOV2L.
SET IOV3A TO ADDRESS OF SENDMSG-BUFFER3.
MOVE 0 TO IOV3AL.
MOVE LENGTH OF SENDMSG-BUFFER3 TO IOV3L.
SET MSG-ACCRIGHTS TO NULLS.
SET MSG-ACCRIGHTS-LEN TO NULLS.
MOVE 0 TO FLAGS.
MOVE 'MESSAGE TEXT 1 ' TO SENDMSG-BUFFER1.
MOVE 'MESSAGE TEXT 2 ' TO SENDMSG-BUFFER2.
MOVE 'MESSAGE TEXT 3 ' TO SENDMSG-BUFFER3.

```

```

CALL 'EZASOKET' USING SOC-FUNCTION S MSG-HDR FLAGS ERRNO RETCODE.

```


Figure 111. SENDMSG call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions” on page 392](#).

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing SENDMSG. The field is left-justified and padded on the right with blanks.

S

A value or the address of a halfword binary number specifying the socket descriptor.

MSG

A pointer to an array of message headers from which messages are sent.

Field

Description

NAME

On input, a pointer to a buffer where the sender's address is stored upon completion of the call. The storage being pointed to should be for an IPv4 socket address or an IPv6 socket address. The IPv4 socket address structure contains the following fields:

Field

Description

FAMILY

Output parameter. A halfword binary number specifying the IPv4 addressing family. The value for IPv4 socket descriptor (S parameter) is decimal 2, indicating AF_INET.

PORT

Output parameter. A halfword binary number specifying the port number of the sending socket.

IP-ADDRESS

Output parameter. A fullword binary number specifying the 32-bit IPv4 IP address of the sending socket.

RESERVED

Output parameter. An 8-byte reserved field. This field is required, but is not used.

The IPv6 socket address structure contains the following fields:

Field

Description

FAMILY

Output parameter. A halfword binary number specifying the IPv6 addressing family. The value for IPv6 socket descriptor (S parameter) is decimal 19, indicating AF_INET6.

PORT

Output parameter. A halfword binary number specifying the port number of the sending socket.

FLOWINFO

A fullword binary field specifying the traffic class and flow label. This field must be set to 0.

IP-ADDRESS

Output parameter. A 16-byte binary field set to the 128-bit IPv6 IP address of the sending socket.

SCOPE-ID

A fullword binary field which identifies a set of interfaces as appropriate for the scope of the address carried in the IPv6-ADDRESS field. A value of 0 indicates the SCOPE-ID field does not identify the set of interfaces to be used, and may be specified for any address types and

scopes. For a link scope IPv6-ADDRESS, SCOPE-ID may specify a link index which identifies a set of interfaces. For all other address scopes, SCOPE-ID must be set to 0.

NAME-LEN

On input, a pointer to the size of the address buffer.

IOV

On input, a pointer to an array of three fullword structures with the number of structures equal to the value in IOVCNT and the format of the structures as follows:

Fullword 1

A pointer to the address of a data buffer.

Fullword 2

Reserved.

Fullword 3

A pointer to the length of the data buffer referenced in Fullword 1.

In COBOL, the IOV structure must be defined separately in the Linkage section, as shown in the example.

IOVCNT

On input, a pointer to a fullword binary field specifying the number of data buffers provided for this call.

ACCRIGHTS

On input, a pointer to the access rights received. This field is ignored.

ACCRIGHTS-LEN

On input, a pointer to the length of the access rights received. This field is ignored.

FLAGS

A fullword field containing the following values:

Literal Value	Binary Value	Description
NO-FLAG	X'00000000'	No flag is set. The command behaves like a WRITE call.
MSG-OOB	X'00000001'	Send out-of-band data. (Stream sockets only.)
MSG-DONTRROUTE	X'00000004'	Do not route. Routing is provided by the calling program.

Parameter values returned to the application

ERRNO

A fullword binary field. If RETCODE is negative, this contains an error number. See [Appendix B, "Socket call error return codes," on page 743](#) for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value**Description****≥0**

A successful call. The value is set to the number of bytes transmitted.

-1

Check ERRNO for an error code.

SENDTO

SENDTO is similar to SEND, except that it includes the destination address parameter. The destination address allows you to use the SENDTO call to send datagrams on a UDP socket, regardless of whether the socket is connected.

The FLAGS parameter allows you to:

- Send out-of-band data, such as interrupts, aborts, and data marked as urgent.
- Suppress use of local routing tables. This implies that the caller takes control of routing, which requires writing network software.

For datagram sockets, SENDTO transmits the entire datagram if it fits into the receiving buffer. Extra data is discarded.

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 call to this function can send any number of bytes, up to the entire 1000 bytes, with the number of bytes sent returned in RETCODE. Therefore, programs using stream sockets should place SENDTO in a loop that repeats the call until all data has been sent.

Note: See [“EZACIC04 ” on page 523](#) for a subroutine that will translate EBCDIC input data to ASCII.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389 .
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

[Figure 112 on page 496](#) shows an example of SENDTO call instructions.

```

WORKING-STORAGE SECTION.
01 SOC-FUNCTION PIC X(16) VALUE IS 'SENDTO'.
01 S PIC 9(4) BINARY.
01 FLAGS PIC 9(8) BINARY.
    88 NO-FLAG VALUE IS 0.
    88 OOB VALUE IS 1.
    88 DONT-ROUTE VALUE IS 4.
01 NBYTE PIC 9(8) BINARY.
01 BUF PIC X(length of buffer).

* IPv4 socket address structure.
01 NAME
    03 FAMILY PIC 9(4) BINARY.
    03 PORT PIC 9(4) BINARY.
    03 IP-ADDRESS PIC 9(8) BINARY.
    03 RESERVED PIC X(8).

* IPv6 socket address structure.
01 NAME
    03 FAMILY PIC 9(4) BINARY.
    03 PORT PIC 9(4) BINARY.
    03 FLOWINFO PIC 9(8) BINARY.
    03 IP-ADDRESS.
        10 FILLER PIC 9(16) BINARY.
        10 FILLER PIC 9(16) BINARY.
    03 SCOPE-ID PIC 9(8) BINARY.

01 ERRNO PIC 9(8) BINARY.
01 RETCODE PIC S9(8) BINARY.

PROCEDURE DIVISION.
    CALL 'EZASOKET' USING SOC-FUNCTION S FLAGS NBYTE
        BUF NAME ERRNO RETCODE.

```

Figure 112. SENDTO call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions” on page 392](#).

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing SENDTO. The field is left-justified and padded on the right with blanks.

S

A halfword binary number set to the socket descriptor of the socket sending the data.

FLAGS

A fullword field that returns one of the following values:

Literal Value	Binary Value	Description
NO-FLAG	X'00000000'	No flag is set. The command behaves like a WRITE call.
MSG-OOB	X'00000001'	Send out-of-band data. (Stream sockets only.)
MSG-DONTROUTE	X'00000004'	Do not route. Routing is provided by the calling program.

NBYTE

A fullword binary number set to the number of bytes to transmit.

BUF

Specifies the buffer containing the data to be transmitted. BUF should be the size specified in NBYTE.

NAME

Specifies the IPv4 socket address structure as follows:

FAMILY

A halfword binary field containing the IPv4 addressing family. For TCP/IP the value must be decimal 2, indicating AF_INET.

PORT

A halfword binary field containing the port number bound to the socket.

IP-ADDRESS

A fullword binary field containing the socket's 32-bit IPv4 IP address.

RESERVED

Specifies eight-byte reserved field. This field is required, but not used.

Specifies the IPv6 socket address structure as follows:

FAMILY

A halfword binary field containing the IPv6 addressing family. For TCP/IP the value is decimal 19, indicating AF_INET6.

PORT

A halfword binary field containing the port number bound to the socket.

FLOWINFO

A fullword binary field specifying the traffic class and flow label. This field must be set to 0.

IP-ADDRESS

A 16-byte binary field set to the 128-bit IPv6 IP address, in network byte order.

SCOPE-ID

A fullword binary field which identifies a set of interfaces as appropriate for the scope of the address carried in the IPv6-ADDRESS field. A value of 0 indicates the SCOPE-ID field does not identify the set of interfaces to be used, and may be specified for any address types and scopes. For a link scope IPv6-ADDRESS, SCOPE-ID may specify a link index which identifies a set of interfaces. For all other address scopes, SCOPE-ID must be set to 0.

Parameter values returned to the application

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See [Appendix B, "Socket call error return codes,"](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value**Description**

≥0

A successful call. The value is set to the number of bytes transmitted.

-1

Check **ERRNO** for an error code.

SETSOCKOPT

The SETSOCKOPT call sets the options associated with a socket. SETSOCKOPT can be called only for sockets in the AF_INET or AF_INET6 domains.

The OPTVAL and OPTLEN parameters are used to pass data used by the particular set command. The OPTVAL parameter points to a buffer containing the data needed by the set command. The OPTLEN parameter must be set to the size of the data pointed to by OPTVAL.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.

Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 113 on page 498 shows an example of SETSOCKOPT call instructions.

```

WORKING-STORAGE SECTION.
01 SOC-FUNCTION      PIC X(16)  VALUE IS 'SETSOCKOPT'.
01 S                 PIC 9(4)   BINARY.
01 OPTNAME           PIC 9(8)   BINARY.
01 OPTVAL            PIC 9(16)  BINARY.
01 OPTLEN            PIC 9(8)   BINARY.
01 ERRNO             PIC 9(8)   BINARY.
01 RETCODE           PIC S9(8)  BINARY.
01 OPTVAL            PIC 9(16)  BINARY.
01 OPTLEN            PIC 9(8)   BINARY.
01 ERRNO             PIC 9(8)   BINARY.
01 RETCODE           PIC S9(8)  BINARY.

PROCEDURE DIVISION
    CALL 'EZASOKET' USING SOC-FUNCTION S OPTNAME
                        OPTVAL OPTLEN ERRNO RETCODE.

```

Figure 113. SETSOCKOPT call instruction example

For equivalent PL/I and assembler language declarations, see ["Converting parameter descriptions"](#) on page 392.

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing SETSOCKOPT. The field is left-justified and padded to the right with blanks.

S

A halfword binary number set to the socket whose options are to be set.

OPTNAME

Input parameter. See the table below for a list of the options and their unique requirements.

See Appendix D, ["GETSOCKOPT/SETSOCKOPT command values,"](#) on page 767 for the numeric values of OPTNAME.

Note: COBOL programs cannot contain field names with the underbar character. Fields representing the option name should contain dashes instead.

OPTVAL

Contains data which further defines the option specified in OPTNAME. For the SETSOCKOPT API, OPTVAL will be an input parameter. See the table below for a list of the options and their unique requirements.

OPTLEN

Input parameter. A fullword binary field containing the length of the data returned in OPTVAL. See the table below for determining on what to base the value of OPTLEN.

Parameter values returned to the application

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See [Appendix B, “Socket call error return codes,”](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check ERRNO for an error code.

Table 20. OPTNAME options for GETSOCKOPT and SETSOCKOPT

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
IP_ADD_MEMBERSHIP Use this option to enable an application to join a multicast group on a specific interface. An interface has to be specified with this option. Only applications that want to receive multicast datagrams need to join multicast groups. This is an IPv4-only socket option.	Contains the IP_MREQ structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 interface address. See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ. See SEZAINST(EZACOBOL) for the COBOL example of IP-MREQ.	N/A
IP_ADD_SOURCE_MEMBERSHIP Use this option to enable an application to join a source multicast group on a specific interface and a specific source address. You must specify an interface and a source address with this option. Applications that want to receive multicast datagrams need to join source multicast groups. This is an IPv4-only socket option.	Contains the IP_MREQ_SOURCE structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ_SOURCE structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 source address and a 4-byte IPv4 interface address. See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ_SOURCE. See SEZAINST(EZACOBOL) for the COBOL example of IP-MREQ-SOURCE.	N/A

Table 20. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IP_BLOCK_SOURCE</p> <p>Use this option to enable an application to block multicast packets that have a source address that matches the given IPv4 source address. You must specify an interface and a source address with this option. The specified multicast group must have been joined previously.</p> <p>This is an IPv4-only socket option.</p>	<p>Contains the IP_MREQ_SOURCE structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ_SOURCE structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 source address and a 4-byte IPv4 interface address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ_SOURCE.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IP-MREQ-SOURCE.</p>	N/A
<p>IP_DROP_MEMBERSHIP</p> <p>Use this option to enable an application to exit a multicast group or to exit all sources for a multicast group.</p> <p>This is an IPv4-only socket option.</p>	<p>Contains the IP_MREQ structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 interface address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IP-MREQ.</p>	N/A
<p>IP_DROP_SOURCE_MEMBERSHIP</p> <p>Use this option to enable an application to exit a source multicast group.</p> <p>This is an IPv4-only socket option.</p>	<p>Contains the IP_MREQ_SOURCE structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ_SOURCE structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 source address and a 4-byte IPv4 interface address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ_SOURCE.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IP-MREQ-SOURCE.</p>	N/A

Table 20. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IP_MULTICAST_IF</p> <p>Use this option to set or obtain the IPv4 interface address used for sending outbound multicast datagrams from the socket application.</p> <p>This is an IPv4-only socket option.</p> <p>Note: Multicast datagrams can be transmitted only on one interface at a time.</p>	<p>A 4-byte binary field containing an IPv4 interface address.</p>	<p>A 4-byte binary field containing an IPv4 interface address.</p>
<p>IP_MULTICAST_LOOP</p> <p>Use this option to control or determine whether a copy of multicast datagrams is looped back for multicast datagrams sent to a group to which the sending host itself belongs. The default is to loop the datagrams back.</p> <p>This is an IPv4-only socket option.</p>	<p>A 1-byte binary field.</p> <p>To enable, set to 1.</p> <p>To disable, set to 0.</p>	<p>A 1-byte binary field.</p> <p>If enabled, will contain a 1.</p> <p>If disabled, will contain a 0.</p>
<p>IP_MULTICAST_TTL</p> <p>Use this option to set or obtain the IP time-to-live of outgoing multicast datagrams. The default value is '01'x meaning that multicast is available only to the local subnet.</p> <p>This is an IPv4-only socket option.</p>	<p>A 1-byte binary field containing the value of '00'x to 'FF'x.</p>	<p>A 1-byte binary field containing the value of '00'x to 'FF'x.</p>
<p>IP_UNBLOCK_SOURCE</p> <p>Use this option to enable an application to unblock a previously blocked source for a given IPv4 multicast group. You must specify an interface and a source address with this option.</p> <p>This is an IPv4-only socket option.</p>	<p>Contains the IP_MREQ_SOURCE structure as defined in SYS1.MACLIB(BPXYSOCK). The IP_MREQ_SOURCE structure contains a 4-byte IPv4 multicast address followed by a 4-byte IPv4 source address and a 4-byte IPv4 interface address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of IP_MREQ_SOURCE.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IP-MREQ-SOURCE.</p>	

Table 20. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IPV6_ADDR_PREFERENCES</p> <p>Use this option to query or set IPv6 address preferences of a socket. The default source address selection algorithm considers these preferences when it selects an IP address that is appropriate to communicate with a given destination address.</p> <p>This is an AF_INET6-only socket option.</p> <p>Result: These flags are only preferences. The stack could assign a source IP address that does not conform to the IPV6_ADDR_PREFERENCES flags that you specify.</p> <p>Guideline: Use the INET6_IS_SRCADDR function to test whether the source IP address matches one or more IPV6_ADDR_PREFERENCES flags.</p>	<p>Contains the 4-byte flags field IPV6_ADDR_PREFERENCES_FLAGS that is defined in SYS1.MACLIB(BPXYSOCK) with the following flags:</p> <p>IPV6_PREFER_SRC_HOME (X'00000001') Prefer home address</p> <p>IPV6_PREFER_SRC_COA (X'00000002') Prefer care-of address</p> <p>IPV6_PREFER_SRC_TMP (X'00000004') Prefer temporary address</p> <p>IPV6_PREFER_SRC_PUBLIC (X'00000008') Prefer public address</p> <p>IPV6_PREFER_SRC_CGA (X'00000010') Prefer cryptographically generated address</p> <p>IPV6_PREFER_SRC_NONCGA (X'00000020') Prefer non-cryptographically generated address</p> <p>Some of these flags are contradictory. Combining contradictory flags, such as IPV6_PREFER_SRC_CGA and IPV6_PREFER_SRC_NONCGA, results in error code EINVAL.</p> <p>See IPV6_ADDR_PREFERENCES and Mapping of GAI_HINTS/GAI_ADDRINFO EFLAGS in SEZAINST(CBLOCK) for the PL/I example of the OPTNAME and flag definitions.</p> <p>See IPV6_ADDR_PREFERENCES and AI_EFLAGS mappings in SEZAINST(EZACOBOL) for the COBOL example of the OPTNAME and flag definitions.</p>	<p>Contains the 4-byte flags field IPV6_ADDR_PREFERENCES_FLAGS that is defined in SYS1.MACLIB(BPXYSOCK) with the following flags:</p> <p>IPV6_PREFER_SRC_HOME (X'00000001') Prefer home address</p> <p>IPV6_PREFER_SRC_COA (X'00000002') Prefer care-of address</p> <p>IPV6_PREFER_SRC_TMP (X'00000004') Prefer temporary address</p> <p>IPV6_PREFER_SRC_PUBLIC (X'00000008') Prefer public address</p> <p>IPV6_PREFER_SRC_CGA (X'00000010') Prefer cryptographically generated address</p> <p>IPV6_PREFER_SRC_NONCGA (X'00000020') Prefer non-cryptographically generated address</p> <p>See IPV6_ADDR_PREFERENCES and Mapping of GAI_HINTS/GAI_ADDRINFO EFLAGS in SEZAINST(CBLOCK) for the PL/I example of the OPTNAME and flag definitions.</p> <p>See IPV6_ADDR_PREFERENCES and AI_EFLAGS mappings in SEZAINST(EZACOBOL) for the COBOL example of the OPTNAME and flag definitions.</p>

Table 20. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IPV6_JOIN_GROUP</p> <p>Use this option to control the reception of multicast packets and specify that the socket join a multicast group.</p> <p>This is an IPv6-only socket option.</p>	<p>Contains the IPV6_MREQ structure as defined in SYS1.MACLIB(BPXYSOCK). The IPV6_MREQ structure contains a 16-byte IPv6 multicast address followed by a 4-byte IPv6 interface index number.</p> <p>If the interface index number is 0, then the stack chooses the local interface.</p> <p>See the SEZAINST(CBLOCK) for the PL/I example of IPV6_MREQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IPV6-MREQ.</p>	<p>N/A</p>
<p>IPV6_LEAVE_GROUP</p> <p>Use this option to control the reception of multicast packets and specify that the socket leave a multicast group.</p> <p>This is an IPv6-only socket option.</p>	<p>Contains the IPV6_MREQ structure as defined in SYS1.MACLIB(BPXYSOCK). The IPV6_MREQ structure contains a 16-byte IPv6 multicast address followed by a 4-byte IPv6 interface index number.</p> <p>If the interface index number is 0, then the stack chooses the local interface.</p> <p>See the SEZAINST(CBLOCK) for the PL/I example of IPV6_MREQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of IPV6-MREQ.</p>	<p>N/A</p>
<p>IPV6_MULTICAST_HOPS</p> <p>Use to set or obtain the hop limit used for outgoing multicast packets.</p> <p>This is an IPv6-only socket option.</p>	<p>Contains a 4-byte binary value specifying the multicast hops. If not specified, then the default is 1 hop.</p> <p>-1 indicates use stack default.</p> <p>0 – 255 is the valid hop limit range.</p> <p>Note: An application must be APF authorized to enable it to set the hop limit value above the system defined hop limit value. CICS applications cannot execute as APF authorized.</p>	<p>Contains a 4-byte binary value in the range 0 – 255 indicating the number of multicast hops.</p>

Table 20. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>IPv6_MULTICAST_IF</p> <p>Use this option to set or obtain the index of the IPv6 interface used for sending outbound multicast datagrams from the socket application.</p> <p>This is an IPv6-only socket option.</p>	<p>Contains a 4-byte binary field containing an IPv6 interface index number.</p>	<p>Contains a 4-byte binary field containing an IPv6 interface index number.</p>
<p>IPv6_MULTICAST_LOOP</p> <p>Use this option to control or determine whether a multicast datagram is looped back on the outgoing interface by the IP layer for local delivery when datagrams are sent to a group to which the sending host itself belongs. The default is to loop multicast datagrams back.</p> <p>This is an IPv6-only socket option.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to 1.</p> <p>To disable, set to 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>
<p>IPv6_UNICAST_HOPS</p> <p>Use this option to set or obtain the hop limit used for outgoing unicast IPv6 packets.</p> <p>This is an IPv6-only socket option.</p>	<p>Contains a 4-byte binary value specifying the unicast hops. If not specified, then the default is 1 hop.</p> <p>-1 indicates use stack default.</p> <p>0 – 255 is the valid hop limit range.</p> <p>Note: APF authorized applications are permitted to set a hop limit that exceeds the system configured default. CICS applications cannot execute as APF authorized.</p>	<p>Contains a 4-byte binary value in the range 0 – 255 indicating the number of unicast hops.</p>
<p>IPv6_V6ONLY</p> <p>Use this option to set or determine whether the socket is restricted to send and receive only IPv6 packets. The default is to not restrict the sending and receiving of only IPv6 packets.</p> <p>This is an IPv6-only socket option.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to 1.</p> <p>To disable, set to 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>

Table 20. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>MCAST_BLOCK_SOURCE</p> <p>Use this option to enable an application to block multicast packets that have a source address that matches the given source address. You must specify an interface index and a source address with this option. The specified multicast group must have been joined previously.</p>	<p>Contains the GROUP_SOURCE_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_SOURCE_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address and a socket address structure of the source address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_SOURCE_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-SOURCE-REQ.</p>	N/A
<p>MCAST_JOIN_GROUP</p> <p>Use this option to enable an application to join a multicast group on a specific interface. You must specify an interface index. Applications that want to receive multicast datagrams must join multicast groups.</p>	<p>Contains the GROUP_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-REQ.</p>	N/A
<p>MCAST_JOIN_SOURCE_GROUP</p> <p>Use this option to enable an application to join a source multicast group on a specific interface and a source address. You must specify an interface index and the source address. Applications that want to receive multicast datagrams only from specific source addresses need to join source multicast groups.</p>	<p>Contains the GROUP_SOURCE_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_SOURCE_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address and a socket address structure of the source address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_SOURCE_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-SOURCE-REQ.</p>	N/A

Table 20. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>MCAST_LEAVE_GROUP</p> <p>Use this option to enable an application to exit a multicast group or exit all sources for a given multicast groups.</p>	<p>Contains the GROUP_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-REQ.</p>	<p>N/A</p>
<p>MCAST_LEAVE_SOURCE_GROUP</p> <p>Use this option to enable an application to exit a source multicast group.</p>	<p>Contains the GROUP_SOURCE_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_SOURCE_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address and a socket address structure of the source address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_SOURCE_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-SOURCE-REQ.</p>	<p>N/A</p>
<p>MCAST_UNBLOCK_SOURCE</p> <p>Use this option to enable an application to unblock a previously blocked source for a given multicast group. You must specify an interface index and a source address with this option.</p>	<p>Contains the GROUP_SOURCE_REQ structure as defined in SYS1.MACLIB(BPXYSOCK). The GROUP_SOURCE_REQ structure contains a 4-byte interface index number followed by a socket address structure of the multicast address and a socket address structure of the source address.</p> <p>See SEZAINST(CBLOCK) for the PL/I example of GROUP_SOURCE_REQ.</p> <p>See SEZAINST(EZACOBOL) for the COBOL example of GROUP-SOURCE-REQ.</p>	<p>N/A</p>

Table 20. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_ASCII</p> <p>Use this option to set or determine the translation to ASCII data option. When SO_ASCII is set, data is translated to ASCII. When SO_ASCII is not set, data is not translated to or from ASCII.</p> <p>Note: This is a REXX-only socket option.</p>	<p>To enable, set to ON.</p> <p>To disable, set to OFF.</p> <p>Note: The <i>optvalue</i> is returned and is optionally followed by the name of the translation table that is used if translation is applied to the data.</p>	<p>If enabled, contains ON.</p> <p>If disabled, contains OFF.</p> <p>Note: The <i>optvalue</i> is returned and is optionally followed by the name of the translation table that is used if translation is applied to the data.</p>
<p>SO_BROADCAST</p> <p>Use this option to set or determine whether a program can send broadcast messages over the socket to destinations that can receive datagram messages. The default is disabled.</p> <p>Note: This option has no meaning for stream sockets.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to 1 or a positive value.</p> <p>To disable, set to 0.</p>	<p>A 4-byte field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>
<p>SO_DEBUG</p> <p>Use SO_DEBUG to set or determine the status of the debug option. The default is <i>disabled</i>. The debug option controls the recording of debug information.</p> <p>Notes:</p> <ol style="list-style-type: none"> 1. This is a REXX-only socket option. 2. This option has meaning only for stream sockets. 	<p>To enable, set to ON.</p> <p>To disable, set to OFF.</p>	<p>If enabled, contains ON.</p> <p>If disabled, contains OFF.</p>
<p>SO_EBCDIC</p> <p>Use this option to set or determine the translation to EBCDIC data option. When SO_EBCDIC is set, data is translated to EBCDIC. When SO_EBCDIC is not set, data is not translated to or from EBCDIC. This option is ignored by EBCDIC hosts.</p> <p>Note: This is a REXX-only socket option.</p>	<p>To enable, set to ON.</p> <p>To disable, set to OFF.</p> <p>Note: The <i>optvalue</i> is returned and is optionally followed by the name of the translation table that is used if translation is applied to the data.</p>	<p>If enabled, contains ON.</p> <p>If disabled, contains OFF.</p> <p>Note: The <i>optvalue</i> is returned and is optionally followed by the name of the translation table that is used if translation is applied to the data.</p>
<p>SO_ERROR</p> <p>Use this option to request pending errors on the socket or to check for asynchronous errors on connected datagram sockets or for other errors that are not explicitly returned by one of the socket calls. The error status is clear afterwards.</p>	<p>N/A</p>	<p>A 4-byte binary field containing the most recent ERRNO for the socket.</p>

Table 20. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_KEEPAIVE</p> <p>Use this option to set or determine whether the keep alive mechanism periodically sends a packet on an otherwise idle connection for a stream socket.</p> <p>The default is disabled.</p> <p>When activated, the keep alive mechanism periodically sends a packet on an otherwise idle connection. If the remote TCP does not respond to the packet or to retransmissions of the packet, the connection is terminated with the error ETIMEDOUT.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to 1 or a positive value.</p> <p>To disable, set to 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>
<p>SO_LINGER</p> <p>Use this option to control or determine how TCP/IP processes data that has not been transmitted when a CLOSE is issued for the socket. The default is disabled.</p> <p>Notes:</p> <ol style="list-style-type: none"> 1. This option has meaning only for stream sockets. 2. If you set a zero linger time, the connection cannot close in an orderly manner, but stops, resulting in a RESET segment being sent to the connection partner. Also, if the aborting socket is in nonblocking mode, the close call is treated as though no linger option had been set. <p>When SO_LINGER is set and CLOSE is called, the calling program is blocked until the data is successfully transmitted or the connection has timed out.</p> <p>When SO_LINGER is not set, the CLOSE returns without blocking the caller, and TCP/IP continues to attempt to send data for a specified time. This usually allows sufficient time to complete the data transfer.</p> <p>Use of the SO_LINGER option does not guarantee successful completion because TCP/IP waits only the amount of time specified in OPTVAL for SO_LINGER.</p>	<p>Contains an 8-byte field containing two 4-byte binary fields.</p> <p>Assembler coding:</p> <pre>ONOFF DS F LINGER DS F</pre> <p>COBOL coding:</p> <pre>ONOFF PIC 9(8) BINARY. LINGER PIC 9(8) BINARY.</pre> <p>Set ONOFF to a nonzero value to enable and set to 0 to disable this option. Set LINGER to the number of seconds that TCP/IP lingers after the CLOSE is issued.</p>	<p>Contains an 8-byte field containing two 4-byte binary fields.</p> <p>Assembler coding:</p> <pre>ONOFF DS F LINGER DS F</pre> <p>COBOL coding:</p> <pre>ONOFF PIC 9(8) BINARY. LINGER PIC 9(8) BINARY.</pre> <p>A nonzero value returned in ONOFF indicates enabled, a 0 indicates disabled. LINGER indicates the number of seconds that TCP/IP will try to send data after the CLOSE is issued.</p>

Table 20. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_OOBLINE</p> <p>Use this option to control or determine whether out-of-band data is received.</p> <p>Note: This option has meaning only for stream sockets.</p> <p>When this option is set, out-of-band data is placed in the normal data input queue as it is received and is available to a RECV or a RECVFROM even if the OOB flag is not set in the RECV or the RECVFROM.</p> <p>When this option is disabled, out-of-band data is placed in the priority data input queue as it is received and is available to a RECV or a RECVFROM only when the OOB flag is set in the RECV or the RECVFROM.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to 1 or a positive value.</p> <p>To disable, set to 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>
<p>SO_RCVBUF</p> <p>Use this option to control or determine the size of the data portion of the TCP/IP receive buffer.</p> <p>The size of the data portion of the receive buffer is protocol-specific, based on the following values prior to any SETSOCKOPT call:</p> <ul style="list-style-type: none"> • TCPRCVBufsize keyword on the TCPCONFIG statement in the PROFILE.TCPIP data set for a TCP Socket • UDPRCVBufsize keyword on the UDPCONFIG statement in the PROFILE.TCPIP data set for a UDP Socket • The default of 65535 for a raw socket 	<p>A 4-byte binary field.</p> <p>To enable, set to a positive value specifying the size of the data portion of the TCP/IP receive buffer.</p> <p>To disable, set to a 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a positive value indicating the size of the data portion of the TCP/IP receive buffer.</p> <p>If disabled, contains a 0.</p>

Table 20. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_RCVTIMEO</p> <p>Use this option to control or determine the maximum length of time that a receive-type function can wait before it completes.</p> <p>If a receive-type function has blocked for the maximum length of time that was specified without receiving data, control is returned with an errno set to EWOULDBLOCK. The default value for this option is 0, which indicates that a receive-type function does not time out.</p> <p>When the MSG_WAITALL flag (stream sockets only) is specified, the timeout takes precedence. The receive-type function can return the partial count. See the explanation of that operation's MSG_WAITALL flag parameter.</p> <p>The following receive-type functions are supported:</p> <ul style="list-style-type: none"> • READ • READV • RECV • RECVFROM • RECVMMSG 	<p>This option requires a TIMEVAL structure, which is defined in SYS1.MACLIB(BPXRLIM) macro. The TIMEVAL structure contains the number of seconds and microseconds specified as fullword binary numbers. The seconds can be a value in the range 0 - 2 678 400 (equal to 31 days), and the microseconds can be a value in the range 0 - 1 000 000 (equal to 1 second). Although TIMEVAL value can be specified using microsecond granularity, the internal TCP/IP timers that are used to implement this function have a granularity of approximately 100 milliseconds.</p>	<p>This option stores a TIMEVAL structure that is defined in the SYS1.MACLIB(BPXRLIM) macro. The TIMEVAL structure contains the number of seconds and microseconds, which are specified as fullword binary numbers. The number of seconds value that is returned is in the range 0 - 2 678 400 (equal to 31 days). The number of microseconds value that is returned is in the range 0 - 1 000 000.</p>

Table 20. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_REUSEADDR</p> <p>Use this option to control or determine whether local addresses are reused. The default is disabled. This alters the normal algorithm used with BIND. The normal BIND algorithm allows each Internet address and port combination to be bound only once. If the address and port have been already bound, then a subsequent BIND will fail and result error will be EADDRINUSE.</p> <p>When this option is enabled, the following situations are supported:</p> <ul style="list-style-type: none"> • A server can BIND the same port multiple times as long as every invocation uses a different local IP address and the wildcard address INADDR_ANY is used only one time per port. • A server with active client connections can be restarted and can bind to its port without having to close all of the client connections. • For datagram sockets, multicasting is supported so multiple bind() calls can be made to the same class D address and port number. • If you require multiple servers to BIND to the same port and listen on INADDR_ANY, see the SHAREPORT option on the PORT statement in TCPIP.PROFILE. 	<p>A 4-byte binary field.</p> <p>To enable, set to 1 or a positive value.</p> <p>To disable, set to 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>
<p>SO_SNDBUF</p> <p>Use this option to control or determine the size of the data portion of the TCP/IP send buffer. The size of the TCP/IP send buffer is protocol specific and is based on the following conditions:</p> <ul style="list-style-type: none"> • The TCPSENDBufsize keyword on the TCPCONFIG statement in the PROFILE.TCPIP data set for a TCP socket • The UDPSENDBufsize keyword on the UDPCONFIG statement in the PROFILE.TCPIP data set for a UDP socket • The default of 65535 for a raw socket 	<p>A 4-byte binary field.</p> <p>To enable, set to a positive value specifying the size of the data portion of the TCP/IP send buffer.</p> <p>To disable, set to a 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a positive value indicating the size of the data portion of the TCP/IP send buffer.</p> <p>If disabled, contains a 0.</p>

Table 20. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_SNDTIMEO</p> <p>Use this option to control or determine the maximum length of time that a send-type function can remain blocked before it completes.</p> <p>If a send-type function has blocked for this length of time, it returns with a partial count or, if no data is sent, with an errno set to EWOULDBLOCK. The default value for this is 0, which indicates that a send-type function does not time out.</p> <p>For a SETSOCKOPT, the following send-type functions are supported:</p> <ul style="list-style-type: none"> • SEND • SENDMSG • SENDTO • WRITE • WRITEV 	<p>This option requires a TIMEVAL structure, which is defined in the SYS1.MACLIB(BPXYRLIM) macro. The TIMEVAL structure contains the number of seconds and microseconds specified as fullword binary numbers. The seconds value is in the range 0 - 2 678 400 (equal to 31 days), and the microseconds value is in the range 0 - 1 000 000 (equal to 1 second). Although the TIMEVAL value can be specified using microsecond granularity, the internal TCP/IP timers that are used to implement this function have a granularity of approximately 100 milliseconds.</p>	<p>This option stores a TIMEVAL structure that is defined in SYS1.MACLIB(BPXYRLIM). The TIMEVAL structure contains the number of seconds and microseconds, which are specified as fullword binary numbers. The number of seconds value that is returned is in the range 0 - 2 678 400 (equal to 31 days). The microseconds value that is returned is in the range 0 - 1 000 000.</p>
<p>SO_TYPE</p> <p>Use this option to return the socket type.</p>	<p>N/A</p>	<p>A 4-byte binary field indicating the socket type:</p> <p>X'1' indicates SOCK_STREAM.</p> <p>X'2' indicates SOCK_DGRAM.</p> <p>X'3' indicates SOCK_RAW.</p>
<p>TCP_KEEPAIVE</p> <p>Use this option to set or determine whether a socket-specific timeout value (in seconds) is to be used in place of a configuration-specific value whenever keep alive timing is active for that socket.</p> <p>When activated, the socket-specified timer value remains in effect until respecified by SETSOCKOPT or until the socket is closed. For more information about the socket option parameters, see TCP_KeepAlive socket option inz/OS Communications Server: IP Programmer's Guide and Reference.</p> <p>Tip: The site administrator can enable the global keep-alive mechanism by specifying the INTERVAL parameter on the TCPCONFIG statement in the TCP/IP stack profile data set, TCPIP.PROFILE.</p>	<p>A 4-byte binary field.</p> <p>To enable, set to a value in the range of 1 – 2 147 460.</p> <p>To disable, set to a value of 0.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains the specific timer value (in seconds) that is in effect for the given socket.</p> <p>If disabled, contains a 0 indicating keep alive timing is not active.</p>

Table 20. OPTNAME options for GETSOCKOPT and SETSOCKOPT (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>TCP_NODELAY</p> <p>Use this option to set or determine whether data sent over the socket is subject to the Nagle algorithm (RFC 896).</p> <p>Under most circumstances, TCP sends data when it is presented. When this option is enabled, TCP will wait to send small amounts of data until the acknowledgment for the previous data sent is received. When this option is disabled, TCP will send small amounts of data even before the acknowledgment for the previous data sent is received.</p> <p>Note: Use the following to set TCP_NODELAY OPTNAME value for COBOL programs:</p> <pre> 01 TCP-NODELAY-VAL PIC 9(10) COMP VALUE 2147483649. 01 TCP-NODELAY-REDEF REDEFINES TCP-NODELAY-VAL. 05 FILLER PIC 9(6) BINARY. 05 TCP-NODELAY PIC 9(8) BINARY. </pre>	<p>A 4-byte binary field.</p> <p>To enable, set to a 0.</p> <p>To disable, set to a 1 or nonzero.</p>	<p>A 4-byte binary field.</p> <p>If enabled, contains a 0.</p> <p>If disabled, contains a 1.</p>

SHUTDOWN

One 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 traffic to shutdown.

When the CLOSE call is used, the SETSOCKOPT OPTVAL LINGER parameter determines the amount of time the system will wait before releasing the connection. For example, with a LINGER value of 30 seconds, system resources (including the IMS or CICS transaction) will remain in the system for up to 30 seconds after the CLOSE call is issued. In high volume, transaction-based systems like CICS and IMS, this can impact performance severely.

If the SHUTDOWN call is issued when the CLOSE call is received, the connection can be closed immediately, rather than waiting for the 30-second delay.

If you issue SHUTDOWN for a socket that currently has outstanding socket calls pending, see [Table 3 on page 32](#) to determine the effects of this operation on the outstanding socket calls.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.

Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 114 on page 514 shows an example of SHUTDOWN call instructions.

```

WORKING-STORAGE SECTION.
01 SOC-FUNCTION    PIC X(16)  VALUE IS 'SHUTDOWN'.
01 S              PIC 9(4)  BINARY.
01 HOW            PIC 9(8)  BINARY.
   88 END-FROM    VALUE 0.
   88 END-TO      VALUE 1.
   88 END-BOTH    VALUE 2.
01 ERRNO          PIC 9(8)  BINARY.
01 RETCODE        PIC S9(8) BINARY.

PROCEDURE DIVISION.
    CALL 'EZASOKET' USING SOC-FUNCTION S HOW ERRNO RETCODE.

```

Figure 114. SHUTDOWN call instruction example

For equivalent PL/I and assembler language declarations, see ["Converting parameter descriptions"](#) on page 392.

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing SHUTDOWN. The field is left-justified and padded on the right with blanks.

S

A halfword binary number set to the socket descriptor of the socket to be shutdown.

HOW

A fullword binary field. Set to specify whether all or part of a connection is to be shut down. The following values can be set:

Value

Description

0 (END-FROM)

Ends further receive operations.

1 (END-TO)

Ends further send operations.

2 (END-BOTH)

Ends further send and receive operations.

Parameter values returned to the application

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See [Appendix B, "Socket call error return codes,"](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

0

Successful call.

-1

Check **ERRNO** for an error code.

SOCKET

The SOCKET call creates an endpoint for communication and returns a socket descriptor representing the endpoint.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 115 on page 515 shows an example of SOCKET call instructions.

```
WORKING-STORAGE SECTION.  
  01 SOC-FUNCTION    PIC X(16)  VALUE IS 'SOCKET'.  
* AF_INET  
  01 AF              PIC 9(8)  COMP VALUE 2.  
  
* AF_INET6  
  01 AF              PIC 9(8)  COMP VALUE 19.  
  01 SOCTYPE         PIC 9(8)  BINARY.  
    88 STREAM        VALUE 1.  
    88 DATAGRAM       VALUE 2.  
    88 RAW            VALUE 3.  
  01 PROTO           PIC 9(8)  BINARY.  
  01 ERRNO           PIC 9(8)  BINARY.  
  01 RETCODE         PIC S9(8) BINARY.  
  
PROCEDURE DIVISION.  
  CALL 'EZASOKET' USING SOC-FUNCTION AF SOCTYPE  
                        PROTO ERRNO RETCODE.
```

Figure 115. SOCKET call instruction example

For equivalent PL/I and assembler language declarations, see ["Converting parameter descriptions"](#) on page 392.

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing SOCKET. The field is left-justified and padded on the right with blanks.

AF

A fullword binary field set to the addressing family. For TCP/IP the value is set to decimal 2 for AF_INET, or decimal 19, indicating AF_INET6.

SOCTYPE

A fullword binary field set to the type of socket required. The types are:

Value

Description

1

Stream sockets provide sequenced, two-way byte streams that are reliable and connection-oriented. They support a mechanism for out-of-band data.

2

Datagram sockets provide datagrams, which are connectionless messages of a fixed maximum length whose reliability is not guaranteed. Datagrams can be corrupted, received out of order, lost, or delivered multiple times.

3

Raw sockets provide the interface to internal protocols (such as IP and ICMP).

PROTO

A fullword binary field set to the protocol to be used for the socket. If this field is set to 0, the default protocol is used. For streams, the default is TCP; for datagrams, the default is UDP.

PROTO numbers are found in the *hlq.etc.proto* data set. For IPv6 raw sockets, PROTO cannot be set to the following values:

Protocol name

Numeric value

IPROTO_HOPOPTS

0

IPPROTO_TCP

6

IPPROTO_UDP

17

IPPROTO_IPV6

41

IPPROTO_ROUTING

43

IPPROTO_FRAGMENT

44

IPPROTO_ESP

50

IPPROTO_AH

51

IPPROTO_NONE

59

IPPROTO_DSTOPTS

60

Parameter values returned to the application

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See [Appendix B, “Socket call error return codes,”](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value

Description

> or = 0

Contains the new socket descriptor.

-1

Check **ERRNO** for an error 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 that it obtained from the concurrent server. See [“GIVESOCKET”](#) on page 448 for a discussion of the use of GETSOCKET and TAKESOCKET calls.

Note: When TAKESOCKET is issued, a new socket descriptor is returned in RETCODE. You should use this new socket descriptor in subsequent calls such as GETSOCKOPT, which require the S (socket descriptor) parameter.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

[Figure 116 on page 518](#) shows an example of TAKESOCKET call instructions.

```

WORKING-STORAGE SECTION.
01 SOC-FUNCTION      PIC X(16)  VALUE IS 'TAKESOCKET'.
01 SOCRECV           PIC 9(4)  BINARY.
01 CLIENT.
03 DOMAIN            PIC 9(8)  BINARY.
03 NAME              PIC X(8).
03 TASK              PIC X(8).
03 RESERVED          PIC X(20).
01 ERRNO             PIC 9(8)  BINARY.
01 RETCODE           PIC S9(8) BINARY.

PROCEDURE DIVISION.
    CALL 'EZASOKET' USING SOC-FUNCTION SOCRECV CLIENT
                        ERRNO RETCODE.

```

Figure 116. TAKESOCKET call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions” on page 392](#).

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing TAKESOCKET. The field is left-justified and padded to the right with blanks.

SOCRECV

A halfword binary field set to the descriptor of the socket to be taken. The socket to be taken is passed by the concurrent server.

CLIENT

Specifies the client ID of the program that is giving the socket. In CICS and IMS, these parameters are passed by the Listener program to the program that issues the TAKESOCKET call.

- In CICS, the information is obtained using EXEC CICS RETRIEVE.
- In IMS, the information is obtained by issuing GU TIM.

DOMAIN

A fullword binary field set to the domain of the program giving the socket. It is decimal 2, indicating AF_INET, or decimal 19, indicating AF_INET6.

Note: The TAKESOCKET can only acquire a socket of the same address family from a GIVESOCKET.

NAME

Specifies an 8-byte character field set to the MVS address space identifier of the program that gave the socket.

TASK

Specifies an 8-byte field set to the task identifier of the task that gave the socket.

RESERVED

A 20-byte reserved field. This field is required, but not used.

Parameter values returned to the application

ERRNO

A fullword binary field. If the value of RETCODE is negative, the field contains an error number. See [Appendix B, “Socket call error return codes,” on page 743](#) for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value	Description
-------	-------------

- ≥ 0**
Contains the new socket descriptor.
- 1**
Check **ERRNO** for an error code.

TERMAPI

This call terminates the session created by INITAPI.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under “CALL instruction API environmental restrictions and programming requirements” on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

[Figure 117 on page 519](#) shows an example of TERMAPI call instructions.

```
WORKING-STORAGE SECTION.
  01  SOC-FUNCTION    PIC X(16)  VALUE IS 'TERMAPI'.

PROCEDURE DIVISION.
  CALL 'EZASOKET' USING SOC-FUNCTION.
```

Figure 117. TERMAPI call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions”](#) on page 392.

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing TERMAPI. The field is left-justified and padded to the right with blanks.

WRITE

The WRITE call writes data on a connected socket. This call is similar to SEND, except that it lacks the control flags available with SEND.

For datagram sockets the WRITE call writes the entire datagram if it fits into the receiving buffer.

Stream sockets act like streams of information with no boundaries separating data. For example, if a program wants to send 1000 bytes, each call to this function can send any number of bytes, up to the entire 1000 bytes. The number of bytes sent will be returned in RETCODE. Therefore, programs using stream sockets should place this call in a loop, calling this function until all data has been sent.

See [“EZACIC04 ”](#) on page 523 for a subroutine that will translate EBCDIC output data to ASCII.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 118 on page 520 shows an example of WRITE call instructions.

```
WORKING-STORAGE SECTION.  
  01  SOC-FUNCTION    PIC X(16)  VALUE IS 'WRITE'.  
  01  S               PIC 9(4)  BINARY.  
  01  NBYTE          PIC 9(8)  BINARY.  
  01  BUF            PIC X(length of buffer).  
  01  ERRNO          PIC 9(8)  BINARY.  
  01  RETCODE        PIC S9(8)  BINARY.  
  
PROCEDURE DIVISION.  
  CALL 'EZASOKET' USING SOC-FUNCTION S NBYTE BUF  
                      ERRNO RETCODE.
```

Figure 118. WRITE call instruction example

For equivalent PL/I and assembler language declarations, see ["Converting parameter descriptions"](#) on page 392.

Parameter values set by the application

SOC-FUNCTION

A 16-byte character field containing WRITE. The field is left-justified and padded on the right with blanks.

S

A halfword binary field set to the socket descriptor.

NBYTE

A fullword binary field set to the number of bytes of data to be transmitted.

BUF

Specifies the buffer containing the data to be transmitted.

Parameter values returned to the application

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See [Appendix B, "Socket call error return codes,"](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following values:

Value	Description
-------	-------------

≥ 0	A successful call. A return code greater than 0 indicates the number of bytes of data written.
-1	Check ERRNO for an error code.

WRITEV

The WRITEV function writes data on a socket from a set of buffers.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. Note: See "Addressability mode (Amode) considerations" under "CALL instruction API environmental restrictions and programming requirements" on page 389.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 119 on page 521 shows an example of WRITEV call instructions.

```

WORKING-STORAGE SECTION.
01 SOC-FUNCTION      PIC X(16) VALUE 'WRITEV'.
01 S                 PIC 9(4)  BINARY.
01 IOVCNT            PIC 9(8)  BINARY.

01 IOV.
03 BUFFER-ENTRY OCCURS N TIMES.
05 BUFFER-POINTER    USAGE IS POINTER.
05 RESERVED          PIC X(4).
05 BUFFER-LENGTH     PIC 9(8) USAGE IS BINARY.

01 ERRNO             PIC 9(8) BINARY.
01 RETCODE           PIC 9(8) BINARY.

PROCEDURE DIVISION.

    SET BUFFER-POINTER(1) TO ADDRESS OF BUFFER1.
    SET BUFFER-LENGTH(1)  TO LENGTH OF BUFFER1.
    SET BUFFER-POINTER(2) TO ADDRESS OF BUFFER2.
    SET BUFFER-LENGTH(2)  TO LENGTH OF BUFFER2.
    " " " " "
    SET BUFFER-POINTER(n) TO ADDRESS OF BUFFERn.
    SET BUFFER-LENGTH(n)  TO LENGTH OF BUFFERn.

    CALL 'EZASOKET' USING SOC-FUNCTION S IOV IOVCNT ERRNO RETCODE.

```

Figure 119. WRITEV call instruction example

For equivalent PL/I and assembler language declarations, see ["Converting parameter descriptions"](#) on page 392.

Parameter values set by the application

S

A value or the address of a halfword binary number specifying the descriptor of the socket from which the data is to be written.

IOV

An array of tripleword structures with the number of structures equal to the value in IOVCNT and the format of the structures as follows:

Fullword 1

The address of a data buffer.

Fullword 2

Reserved.

Fullword 3

The length of the data buffer referenced in Fullword 1.

IOVCNT

A fullword binary field specifying the number of data buffers provided for this call.

Parameters returned by the application

ERRNO

A fullword binary field. If RETCODE is negative, this contains an error number. See [Appendix B, “Socket call error return codes,”](#) on page 743 for information about ERRNO return codes.

RETCODE

A fullword binary field.

Value

Meaning

<0

Check **ERRNO** for an error code.

0

Connection partner has closed connection.

>0

Number of bytes sent.

Using data translation programs for socket call interface

In addition to the socket calls, you can use utility programs to translate data.

Assembler language utility programs call format

The following example shows the assembler language call format for utility programs:

```
>>__CALL EZACIC04,(Inbuf, Inbuf_Length),VL__><
```

Data translation

TCP/IP hosts and networks use ASCII data notation; MVS TCP/IP and its subsystems use EBCDIC data notation. In situations where data must be translated from one notation to the other, you can use the following utility programs:

- EZACIC04 translates EBCDIC data to ASCII data using the translation table documented in the [z/OS Communications Server: IP Configuration Reference](#).
- EZACIC05 translates ASCII data to EBCDIC data using the translation table documented in the [z/OS Communications Server: IP Configuration Reference](#).

- EZACIC14 provides an alternative to EZACIC04 and translates EBCDIC data to ASCII data using the translation table documented in [Figure 127 on page 532](#).
- EZACIC15 provides an alternative to EZACIC05 and translates ASCII data to EBCDIC data using the translation table documented in [Figure 129 on page 533](#).

Bit-string processing

In C-language, bit strings are often used to convey flags, switch settings, and so on; TCP/IP makes frequent uses of bit strings. However, since bit strings are difficult to decode in COBOL, TCP/IP includes the following information:

- EZACIC06 translates bit-masks into character arrays and character arrays into bit-masks.
- EZACIC08 interprets the variable length address list in the HOSTENT structure returned by GETHOSTBYNAME or GETHOSTBYADDR.
- EZACIC09 interprets the ADDRINFO structure returned by GETADDRINFO.

EZACIC04

The EZACIC04 program is used to translate EBCDIC data to ASCII data. [Figure 120 on page 523](#) shows how EZACIC04 translates a byte of EBCDIC data.

first hex digit of byte of EBCDIC data	second hex digit of byte of EBCDIC data															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	01	02	03	1A	09	1A	7F	1A	1A	1A	0B	0C	0D	0E	0F
1	10	11	12	13	1A	0A	08	1A	18	19	1A	1A	1C	1D	1E	1F
2	1A	1A	1C	1A	1A	0A	17	1B	1A	1A	1A	1A	1A	05	06	07
3	1A	1A	16	1A	1A	1E	1A	04	1A	1A	1A	1A	14	15	1A	1A
4	20	A6	E1	80	EB	90	9F	E2	AB	8B	9B	2E	3C	28	2B	7C
5	26	A9	AA	9C	DB	A5	99	E3	A8	9E	21	24	2A	29	3B	5E
6	2D	2F	DF	DC	9A	DD	DE	98	9D	AC	BA	2C	25	5F	3E	3F
7	D7	88	94	B0	B1	B2	FC	D6	FB	60	3A	23	40	27	3D	22
8	F8	61	62	63	64	65	66	67	68	69	96	A4	F3	AF	AE	C5
9	8C	6A	6B	6C	6D	6E	6F	70	71	72	97	87	CE	93	F1	FE
A	C8	7E	73	74	75	76	77	78	79	7A	EF	C0	DA	5B	F2	AE
B	B5	B6	FD	B7	B8	B9	E6	BB	BC	BD	8D	D9	BF	5D	D8	C4
C	7B	41	42	43	44	45	46	47	48	49	CB	CA	BE	E8	EC	ED
D	7D	4A	4B	4C	4D	4E	4F	50	51	52	A1	AD	F5	F4	A3	8F
E	5C	E7	53	54	55	56	57	58	59	5A	A0	85	8E	E9	E4	D1
F	30	31	32	33	34	35	36	37	38	39	B3	F7	F0	FA	A7	FF

Figure 120. EZACIC04 EBCDIC-to-ASCII table

[Figure 121 on page 524](#) shows an example of EZACIC04 call instructions.

```

WORKING-STORAGE SECTION.
  01 OUT-BUFFER    PIC X(length of output).
  01 LENGTH        PIC 9(8) BINARY.

PROCEDURE DIVISION.
  CALL 'EZACIC04' USING OUT-BUFFER LENGTH.

```

Figure 121. EZACIC04 call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions”](#) on page 392.

OUT-BUFFER

A buffer that contains the following data:

- When called, EBCDIC data
- Upon return, ASCII data

LENGTH

Specifies the length of the data to be translated.

EZACIC05

The EZACIC05 program is used to translate ASCII data to EBCDIC data. EBCDIC data is required by COBOL, PL/I, and assembler language programs. [Figure 122 on page 524](#) shows how EZACIC05 translates a byte of ASCII data.

EBCDIC output by EZACIC05		second hex digit of byte of ASCII data															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
first hex digit of byte of ASCII data	0	00	01	02	03	37	2D	2E	2F	16	05	25	0B	0C	0D	0E	0F
	1	10	11	12	13	3C	3D	32	26	18	19	3F	27	22	1D	35	1F
	2	40	5A	7F	7B	5B	6C	50	7D	4D	5D	5C	4E	6B	60	4B	61
	3	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	7A	5E	4C	7E	6E	6F
	4	7C	C1	C2	C3	C4	C5	C6	C7	C8	C9	D1	D2	D3	D4	D5	D6
	5	D7	D8	D9	E2	E3	E4	E5	E6	E7	E8	E9	AD	E0	BD	5F	6D
	6	79	81	82	83	84	85	86	87	88	89	91	92	93	94	95	96
	7	97	98	99	A2	A3	A4	A5	A6	A7	A8	A9	C0	4F	D0	A1	07
	8	00	01	02	03	37	2D	2E	2F	16	05	25	0B	0C	0D	0E	0F
	9	10	11	12	13	3C	3D	32	26	18	19	3F	27	22	1D	35	1F
	A	40	5A	7F	7B	5B	6C	50	7D	4D	5D	5C	4E	6B	60	4B	61
	B	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	7A	5E	4C	7E	6E	6F
	C	7C	C1	C2	C3	C4	C5	C6	C7	C8	C9	D1	D2	D3	D4	D5	D6
	D	D7	D8	D9	E2	E3	E4	E5	E6	E7	E8	E9	AD	E0	BD	5F	6D
	E	79	81	82	83	84	85	86	87	88	89	91	92	93	94	95	96
	F	97	98	99	A2	A3	A4	A5	A6	A7	A8	A9	C0	4F	D0	A1	07

Figure 122. EZACIC05 ASCII-to-EBCDIC table

[Figure 123 on page 525](#) shows an example of EZACIC05 call instructions.


```

WORKING-STORAGE SECTION.
  01 IN-BUFFER      PIC X(length of output)
  01 LENGTH         PIC 9(8) BINARY VALUE

PROCEDURE DIVISION.
  CALL 'EZACIC05' USING IN-BUFFER LENGTH.

```

Figure 123. EZACIC05 call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions”](#) on page 392.

IN-BUFFER

A buffer that contains the following data:

- When called, ASCII data
- Upon return, EBCDIC data

LENGTH

Specifies the length of the data to be translated.

EZACIC06

The SELECT and SELECTEX call uses bit strings to specify the sockets to test and to return the results of the test. Because bit strings are difficult to manage in COBOL, you might want to use EZACIC06 utility program to translate them to character strings to be used with the SELECT or SELECTEX call.

[Figure 124 on page 525](#) shows an example of EZACIC06 call instructions.

```

WORKING-STORAGE SECTION.
  01 CHAR-MASK.
    05 CHAR-STRING          PIC X(nn).

  01 CHAR-ARRAY
    05 CHAR-ENTRY-TABLE    REDEFINES CHAR-MASK.
      10 CHAR-ENTRY        OCCURS nn TIMES.
      PIC X(1).

  01 BIT-MASK.
    05 BIT-ARRAY-FWDS      OCCURS (nn+31)/32 TIMES.
      10 BIT_ARRAY_WORD    PIC 9 (8) COMP.

  01 BIT-FUNCTION-CODES.
    05 CTOB                PIC X(4) VALUE 'CTOB'.
    05 BTOC                PIC X(4) VALUE 'BTOC'.

  01 CHAR-MASK-LENGTH      PIC 9(8) COMP VALUE nn.

PROCEDURE CALL (to convert from character to binary)
  CALL 'EZACIC06' USING CTOB
                        BIT-MASK
                        CHAR-MASK
                        CHAR-MASK-LENGTH
                        RETCODE.

PROCEDURE CALL (to convert from binary to character)
  CALL 'EZACIC06' USING BTOC
                        BIT-MASK
                        CHAR-MASK
                        CHAR-MASK-LENGTH
                        RETCODE.

```

Figure 124. EZACIC06 call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions”](#) on page 392.

Specifies the character array where *nn* is the maximum number of sockets in the array. The first character in the array represents socket 0, the second represents socket 1, and so on. Note that the index is 1 greater than the socket number [for example, CHAR-ENTRY(1) represents socket 0, CHAR-ENTRY (2) represents socket 1, and so on.]

Specifies the bit string to be translated for the SELECT call. Within each fullword of the bit string, the bits are ordered right to left. The right-most bit in the first fullword represents socket 0 and the left-most bit represents socket 31. The right-most bit in the second fullword represents socket 32 and the left-most bit represents socket 63. The number of fullwords in the bit string should be calculated by dividing the sum of 31 and the character array length by 32 (truncate the remainder).

CTOB specifies character array to bit string translation.

Specifies the length of the character array. This field should be no greater than 1 plus the MAXSNO value returned on the INITAPI (which is usually the same as the MAXSOC value specified on the INITAPI).

A binary field that returns one of the following values:

Value	Description
0	Successful call.
-1	Check ERRNO for an error code.

If you want to use the SELECT call to test sockets 0, 5, and 32, and you are using a character array to represent the sockets, you must set the appropriate characters in the character array to 1. In this example, index positions 1, 6 and 33 in the character array are set to 1. Then you can call EZACIC06 with the COMMAND parameter set to CTOB. When EZACIC06 returns, the first fullword of BIT-MASK contains B'0000000000000000000000000100001' to indicate that sockets 0 and 5 will be checked. The second word of BIT-MASK contains B'00000000000000000000000000000001' to indicate that socket 32 will be checked. These instructions process the bit string shown in the following example:

```

MOVE ZEROS TO CHAR-STRING.
MOVE '1' TO CHAR-ENTRY(1), CHAR-ENTRY(6), CHAR-ENTRY(33).
CALL 'EZACIC06' USING TOKEN CTOB BIT-MASK CH-MASK
      CHAR-MASK-LENGTH RETCODE.
MOVE BIT-MASK TO ....

```

When the select call returns and you want to check the bit-mask string for socket activity, enter the following instructions.

```

MOVE ..... TO BIT-MASK.
CALL 'EZACIC06' USING TOKEN BTOC BIT-MASK CH-MASK
      CHAR-MASK-LENGTH RETCODE.
PERFORM TEST-SOCKET THRU TEST-SOCKET-EXIT VARYING IDX
      FROM 1 BY 1 UNTIL IDX EQUAL CHAR-MASK-LENGTH.

TEST-SOCKET.
      IF CHAR-ENTRY(IDX) EQUAL '1'
            THEN PERFORM SOCKET-RESPONSE THRU SOCKET-RESPONSE-EXIT
            ELSE NEXT SENTENCE.
TEST-SOCKET-EXIT.
      EXIT.

```

EZACIC08

The GETHOSTBYNAME and GETHOSTBYADDR calls were derived from C socket calls that return a structure known as HOSTENT. A given TCP/IP host can have multiple alias names and host IP addresses.

TCP/IP uses indirect addressing to connect the variable number of alias names and IP addresses in the HOSTENT structure that are returned by the GETHOSTBYADDR AND GETHOSTBYNAME calls.

If you are coding in PL/I or assembler language, the HOSTENT structure can be processed in a relatively straight-forward manner. However, if you are coding in COBOL, HOSTENT can be more difficult to process and you should use the EZACIC08 subroutine to process it for you.

It works as follows:

1. GETHOSTBYADDR or GETHOSTBYNAME returns a HOSTENT structure that indirectly addresses the lists of alias names and IP addresses.
2. Upon return from GETHOSTBYADDR or GETHOSTBYNAME, your program calls EZACIC08 and passes it the address of the HOSTENT structure. EZACIC08 processes the structure and returns the following information:
 - The length of host name, if present
 - The host name
 - The number of alias names for the host
 - The alias name sequence number
 - The length of the alias name
 - The alias name
 - The host IP address type, always 2 for AF_INET
 - The host IP address length, always 4 for AF_INET
 - The number of host IP addresses for this host
 - The host IP address sequence number
 - The host IP address
3. If the GETHOSTBYADDR or GETHOSTBYNAME call returns more than one alias name or host IP address, the application program should repeat the call to EZACIC08 until all alias names and host IP addresses have been retrieved.

[Figure 125 on page 528](#) shows an example of EZACIC08 call instructions.

WORKING-STORAGE SECTION.

```
01 HOSTENT-ADDR      PIC 9(8) BINARY.
01 HOSTNAME-LENGTH   PIC 9(4) BINARY.
01 HOSTNAME-VALUE     PIC X(255).
01 HOSTALIAS-COUNT    PIC 9(4) BINARY.
01 HOSTALIAS-SEQ      PIC 9(4) BINARY.
01 HOSTALIAS-LENGTH   PIC 9(4) BINARY.
01 HOSTALIAS-VALUE     PIC X(255).
01 HOSTADDR-TYPE      PIC 9(4) BINARY.
01 HOSTADDR-LENGTH    PIC 9(4) BINARY.
01 HOSTADDR-COUNT     PIC 9(4) BINARY.
01 HOSTADDR-SEQ       PIC 9(4) BINARY.
01 HOSTADDR-VALUE     PIC 9(8) BINARY.
01 RETURN-CODE        PIC 9(8) BINARY.
```

PROCEDURE DIVISION.

```
CALL 'EZASOKET' USING 'GETHOSTBYADDR'
                    HOSTADDR HOSTENT-ADDR
                    RETCODE.

CALL 'EZASOKET' USING 'GETHOSTBYNAME'
                    NAMELEN NAME HOSTENT-ADDR
                    RETCODE.

CALL 'EZACIC08' USING HOSTENT-ADDR HOSTNAME-LENGTH
                    HOSTNAME-VALUE HOSTALIAS-COUNT HOSTALIAS-SEQ
                    HOSTALIAS-LENGTH HOSTALIAS-VALUE
                    HOSTADDR-TYPE HOSTADDR-LENGTH HOSTADDR-COUNT
                    HOSTADDR-SEQ HOSTADDR-VALUE RETURN-CODE.
```

Figure 125. EZAZIC08 call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions”](#) on page 392.

Parameter values set by the application

HOSTENT-ADDR

This fullword binary field must contain the address of the HOSTENT structure (as returned by the GETHOSTBYxxxx call). This variable is the same as the variable HOSTENT in the GETHOSTBYADDR and GETHOSTBYNAME socket calls.

HOSTALIAS-SEQ

This halfword field is used by EZACIC08 to index the list of alias names. When EZACIC08 is called, it adds 1 to the current value of HOSTALIAS-SEQ and uses the resulting value to index into the table of alias names. Therefore, for a given instance of GETHOSTBYxxxx, this field should be set to 0 for the initial call to EZACIC08. For all subsequent calls to EZACIC08, this field should contain the HOSTALIAS-SEQ number returned by the previous invocation.

HOSTADDR-SEQ

This halfword field is used by EZACIC08 to index the list of IP addresses. When EZACIC08 is called, it adds 1 to the current value of HOSTADDR-SEQ and uses the resulting value to index into the table of IP addresses. Therefore, for a given instance of GETHOSTBYxxxx, this field should be set to 0 for the initial call to EZACIC08. For all subsequent calls to EZACIC08, this field should contain the HOSTADDR-SEQ number returned by the previous call.

Parameter values returned to the application

HOSTNAME-LENGTH

This halfword binary field contains the length of the host name (if host name was returned).

HOSTNAME-VALUE

This 255-byte character string contains the host name (if host name was returned).

HOSTALIAS-COUNT

This halfword binary field contains the number of alias names returned.

HOSTALIAS-SEQ

This halfword binary field is the sequence number of the alias name currently found in HOSTALIAS-VALUE.

HOSTALIAS-LENGTH

This halfword binary field contains the length of the alias name currently found in HOSTALIAS-VALUE.

HOSTALIAS-VALUE

This 255-byte character string contains the alias name returned by this instance of the call. The length of the alias name is contained in HOSTALIAS-LENGTH.

HOSTADDR-TYPE

This halfword binary field contains the type of host address. For FAMILY type AF_INET, HOSTADDR-TYPE is always 2.

HOSTADDR-LENGTH

This halfword binary field contains the length of the host IP address currently found in HOSTADDR-VALUE. For FAMILY type AF_INET, HOSTADDR-LENGTH is always set to 4.

HOSTADDR-COUNT

This halfword binary field contains the number of host IP addresses returned by this instance of the call.

HOSTADDR-SEQ

This halfword binary field contains the sequence number of the host IP address currently found in HOSTADDR-VALUE.

HOSTADDR-VALUE

This fullword binary field contains a host IP address.

RETURN-CODE

This fullword binary field contains the EZACIC08 return code:

Value**Description****0**

Successful completion.

-1

HOSTENT address is not valid.

-2

A value of HOSTALIAS-SEQ is not valid.

-3

A value of HOSTADDR-SEQ is not valid.

EZACIC09

The GETADDRINFO call was derived from the C socket call that return a structure known as RES. A given TCP/IP host can have multiple sets of NAMES. TCP/IP uses indirect addressing to connect the variable number of NAMES in the RES structure that is returned by the GETADDRINFO call. If you are coding in PL/I or assembler language, the RES structure can be processed in a relatively straight-forward manner. However, if you are coding in COBOL, RES can be more difficult to process and you should use the EZACIC09 subroutine to process it for you. It works as follows:

1. GETADDRINFO returns a RES structure that indirectly addresses the lists of socket address structures.
2. Upon return from GETADDRINFO, your program calls EZACIC09 and passes it the address of the next address information structure as referenced by the NEXT argument. EZACIC09 processes the structure and returns the following information:
 - a. The socket address structure
 - b. The next address information structure
3. If the GETADDRINFO call returns more than one socket address structure the application program should repeat the call to EZACIC09 until all socket address structures have been retrieved.

Figure 126 on page 531 shows an example of EZACIC09 call instructions.

```

WORKING-STORAGE SECTION.
*
* Variables used for the GETADDRINFO call
*
01 getaddrinfo-parms.
    02 node-name pic x(255).
    02 node-name-len pic 9(8) binary.
    02 service-name pic x(32).
    02 service-name-len pic 9(8) binary.
    02 canonical-name-len pic 9(8) binary.
    02 ai-passive pic 9(8) binary value 1.
    02 ai-canonnameok pic 9(8) binary value 2.
    02 ai-numerichost pic 9(8) binary value 4.
    02 ai-numeric serv pic 9(8) binary value 8.
    02 ai-v4mapped pic 9(8) binary value 16.
    02 ai-all pic 9(8) binary value 32.
    02 ai-addrconfig pic 9(8) binary value 64.
*
* Variables used for the EZACIC09 call
*
01 ezacic09-parms.
    02 res usage is pointer.
    02 res-name-len pic 9(8) binary.
    02 res-canonical-name pic x(256).
    02 res-name usage is pointer.
    02 res-next-addrinfo usage is pointer.
*
* Socket address structure
*
01 server-socket-address.
    05 server-family pic 9(4) Binary Value 19.
    05 server-port pic 9(4) Binary Value 9997.
    05 server-flowinfo pic 9(8) Binary Value 0.
    05 server-ipaddr.
        10 filler pic 9(16) binary value 0.
        10 filler pic 9(16) binary value 0.
    05 server-scopeid pic 9(8) Binary Value 0.

LINKAGE SECTION.
01 L1.
    03 HINTS-ADDRINFO.
        05 HINTS-AI-FLAGS PIC 9(8) BINARY.
        05 HINTS-AI-FAMILY PIC 9(8) BINARY.
        05 HINTS-AI-SOCKTYPE PIC 9(8) BINARY.
        05 HINTS-AI-PROTOCOL PIC 9(8) BINARY.
        05 FILLER PIC 9(8) BINARY.
        05 FILLER PIC 9(8) BINARY.
        05 FILLER PIC 9(8) BINARY.
        05 FILLER PIC 9(8) BINARY.
    03 HINTS-ADDRINFO-PTR USAGE IS POINTER.
    03 RES-ADDRINFO-PTR USAGE IS POINTER.
*
* RESULTS ADDRESS INFO
*
01 RESULTS-ADDRINFO.
    05 RESULTS-AI-FLAGS PIC 9(8) BINARY.
    05 RESULTS-AI-FAMILY PIC 9(8) BINARY.
    05 RESULTS-AI-SOCKTYPE PIC 9(8) BINARY.
    05 RESULTS-AI-PROTOCOL PIC 9(8) BINARY.
    05 RESULTS-AI-ADDR-LEN PIC 9(8) BINARY.
    05 RESULTS-AI-CANONICAL-NAME USAGE IS POINTER.
    05 RESULTS-AI-ADDR-PTR USAGE IS POINTER.
    05 RESULTS-AI-NEXT-PTR USAGE IS POINTER.
*
* SOCKET ADDRESS STRUCTURE FROM EZACIC09.
*
01 OUTPUT-NAME-PTR USAGE IS POINTER.
01 OUTPUT-IP-NAME.
    03 OUTPUT-IP-FAMILY PIC 9(4) BINARY.
    03 OUTPUT-IP-PORT PIC 9(4) BINARY.
    03 OUTPUT-IP-SOCK-DATA PIC X(24).
    03 OUTPUT-IPV4-SOCK-DATA REDEFINES OUTPUT-IP-SOCK-DATA.
    05 OUTPUT-IPV4-IPADDR PIC 9(8) BINARY.
    05 FILLER PIC X(20).
    03 OUTPUT-IPV6-SOCK-DATA REDEFINES OUTPUT-IP-SOCK-DATA.
    05 OUTPUT-IPV6-FLOWINFO PIC 9(8) BINARY.
    05 OUTPUT-IPV6-IPADDR.
        10 FILLER PIC 9(16) BINARY.
        10 FILLER PIC 9(16) BINARY.

```

```

05 OUTPUT-IPV6-SCOPEID PIC 9(8) BINARY.
PROCEDURE DIVISION USING L1.
*
* Get and address from the resolver.
*
    move 'yournodename' to node-name.
    move 12 to node-name-len.
    move spaces to service-name.
    move 0 to service-name-len.
    move af_INET6 to hints-ai-family.
    move 49 to hints-ai-flags
    move 0 to hints-ai-socktype.
    move 0 to hints-ai-protocol.
    set address of results-addrinfo to res-addrinfo-ptr.
    set hints-addrinfo-ptr to address of hints-addrinfo.
    call 'EZASOCKET' using soket-getaddrinfo
                                node-name node-name-len
                                service-name service-name-len
                                hints-addrinfo-ptr
                                res-addrinfo-ptr
                                canonical-name-len
                                errno retcode.

*
* Use EZACIC09 to extract the IP address
*
    set address of results-addrinfo to res-addrinfo-ptr.
    set res to address of results-addrinfo.
    move zeros to res-name-len.
    move spaces to res-canonical-name.
    set res-name to nulls.
    set res-next-addrinfo to nulls.
    call 'EZACIC09' using res
                                res-name-len
                                res-canonical-name
                                res-name
                                res-next-addrinfo
                                retcode.
    set address of output-ip-name to res-name.
    move output-ipv6-ipaddr to server-ipaddr.

```

Figure 126. EZACIC09 call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions”](#) on page 392.

Parameter values set by the application:

RES

This fullword binary field must contain the address of the ADDRINFO structure (as returned by the GETADDRINFO call). This variable is the same as the RES variable in the GETADDRINFO socket call.

RES-NAME-LEN

A fullword binary field that will contain the length of the socket address structure as returned by the GETADDRINFO call.

Parameter values returned to the application:

Description

RES-CANONICAL-NAME

A field large enough to hold the canonical name. The maximum field size is 256 bytes. The canonical name length field will indicate the length of the canonical name as returned by the GETADDRINFO call.

RES-NAME

The address of the subsequent socket address structure.

RES-NEXT

The address of the next address information structure.

RETURN-CODE

CODE This fullword binary field contains the EZACIC09 return code:

Value

Description

- 0 Successful call.
- 1 Invalid RES address.

EZACIC14

The EZACIC14 program is an alternative to EZACIC04, which translates EBCDIC data to ASCII data. Figure 127 on page 532 shows how EZACIC14 translates a byte of EBCDIC data.

ASCII output by EZACIC14		second hex digit of byte of EBCDIC data															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
first hex digit of byte of EBCDIC data	0	00	01	02	03	9C	09	86	7F	97	8D	8E	0B	0C	0D	0E	0F
	1	10	11	12	13	9D	85	08	87	18	19	92	8F	1C	1D	1E	1F
	2	80	81	82	83	84	0A	17	1B	88	89	8A	8B	8C	05	06	07
	3	90	91	16	93	94	95	96	04	98	99	9A	9B	14	15	9E	1A
	4	20	A0	E2	E4	E0	E1	E3	E5	E7	F1	A2	2E	3C	28	2B	7C
	5	26	E9	EA	EB	E8	ED	EE	EF	EC	DF	21	24	2A	29	3B	5E
	6	2D	2F	C2	C4	C0	C1	C3	C5	C7	D1	A6	2C	25	5F	3E	3F
	7	F8	C9	CA	CB	C8	CD	CE	CF	CC	60	3A	23	40	27	3D	22
	8	D8	61	62	63	64	65	66	67	68	69	AB	BB	F0	FD	FE	B1
	9	B0	6A	6B	6C	6D	6E	6F	70	71	72	AA	BA	E6	B8	C6	A4
	A	B5	7E	73	74	75	76	77	78	79	7A	A1	BF	D0	5B	DE	AE
	B	AC	A3	A5	B7	A9	A7	B6	BC	BD	BE	DD	A8	AF	5D	B4	D7
	C	7B	41	42	43	44	45	46	47	48	49	AD	F4	F6	F2	F3	F5
	D	7D	4A	4B	4C	4D	4E	4F	50	51	52	B9	FB	FC	F9	FA	FF
	E	5C	F7	53	54	55	56	57	58	59	5A	B2	D4	D6	D2	D3	D5
	F	30	31	32	33	34	35	36	37	38	39	B3	DB	DC	D9	DA	9F

Figure 127. EZACIC14 EBCDIC-to-ASCII table

Figure 128 on page 532 shows an example of EZACIC14 call instructions.

```

WORKING-STORAGE SECTION.
    01 OUT-BUFFER    PIC X(length of output).
    01 LENGTH        PIC 9(8) BINARY.

PROCEDURE DIVISION.
    CALL 'EZACIC14' USING OUT-BUFFER LENGTH.

```

Figure 128. EZACIC14 call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions” on page 392](#).

OUT-BUFFER

A buffer that contains the following data:

- When called, EBCDIC data
- Upon return, ASCII data

LENGTH

Specifies the length of the data to be translated.

EZACIC15

The EZACIC15 program is an alternative to EZACIC05, which translates ASCII data to EBCDIC data. Figure 129 on page 533 shows how EZACIC15 translates a byte of ASCII data.

EBCDIC output by EZACIC15		second hex digit of byte of ASCII data															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
first hex digit of byte of ASCII data	0	00	01	02	03	37	2D	2E	2F	16	05	25	0B	0C	0D	0E	0F
	1	10	11	12	13	3C	3D	32	26	18	19	3F	27	1C	1D	1E	1F
	2	40	5A	7F	7B	5B	6C	50	7D	4D	5D	5C	4E	6B	60	4B	61
	3	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	7A	5E	4C	7E	6E	6F
	4	7C	C1	C2	C3	C4	C5	C6	C7	C8	C9	D1	D2	D3	D4	D5	D6
	5	D7	D8	D9	E2	E3	E4	E5	E6	E7	E8	E9	AD	E0	BD	5F	6D
	6	79	81	82	83	84	85	86	87	88	89	91	92	93	94	95	96
	7	97	98	99	A2	A3	A4	A5	A6	A7	A8	A9	C0	4F	D0	A1	07
	8	20	21	22	23	24	15	06	17	28	29	2A	2B	2C	09	0A	1B
	9	30	31	1A	33	34	35	36	08	38	39	3A	3B	04	14	3E	FF
	A	41	AA	4A	B1	9F	B2	6A	B5	BB	B4	9A	8A	B0	CA	AF	BC
	B	90	8F	EA	FA	BE	A0	B6	B3	9D	DA	9B	8B	B7	B8	B9	A9
	C	64	65	62	66	63	67	9E	68	74	71	72	73	78	75	76	77
	D	AC	69	ED	EE	EB	EF	EC	BF	80	FD	FE	FB	FC	BA	AE	59
	E	44	45	42	46	43	47	9C	48	54	51	52	53	58	55	56	57
	F	8C	49	CD	CE	CB	CF	CC	E1	70	DD	DE	DB	DC	8D	8E	DF

Figure 129. EZACIC15 ASCII-to-EBCDIC table

Figure 130 on page 533 shows an example of EZACIC15 call instructions.

```

WORKING-STORAGE SECTION.
    01 OUT-BUFFER    PIC X(length of output).
    01 LENGTH        PIC 9(8) BINARY.

PROCEDURE DIVISION.
    CALL 'EZACIC15' USING OUT-BUFFER LENGTH.

```

Figure 130. EZACIC15 call instruction example

For equivalent PL/I and assembler language declarations, see [“Converting parameter descriptions”](#) on page 392.

OUT-BUFFER

A buffer that contains the following data:

- When called, ASCII data
- Upon return, EBCDIC data

LENGTH

Specifies the length of the data to be translated.

Call interface sample programs

This information provides sample programs for the call interface that you can use for a PL/I or COBOL application program.

The following are the sample programs that are available in the SEZAINST data set:

Program	Description
EZASOKPS	PL/I call interface sample IPv4 server program
EZASOKPC	PL/I call interface sample IPv4 client program
EZASO6PS	PL/I call interface sample IPv6 server program
EZASO6PC	PL/I call interface sample IPv6 client program
CBLOCK	PL/I common variables
EZACOBOL	COBOL common variables
EZASO6CS	COBOL call interface sample IPv6 server program
EZASO6CC	COBOL call interface sample IPv6 client program

Sample code for IPv4 server program

The EZASOKPS PL/I sample program is a server program that shows you how to use the following calls:

- ACCEPT
- BIND
- CLOSE
- GETSOCKNAME
- INITAPI
- LISTEN
- READ
- SOCKET
- TERMAPI
- WRITE

```
/******  
/*  
/*  MODULE NAME:  EZASOKPS - THIS IS A VERY SIMPLE IPV4 SERVER  */  
/*  
/*  Copyright:    Licensed Materials - Property of IBM          */  
/*  
/*                "Restricted Materials of IBM"                 */  
/*  
/*                5694-A01                                       */  
/*  
/*                (C) Copyright IBM Corp. 1994, 2005           */  
/*  
/*                US Government Users Restricted Rights -      */  
/*                Use, duplication or disclosure restricted by   */  
/*                GSA ADP Schedule Contract with IBM Corp.     */  
/*  
/*  Status:       CSV1R7                                         */  
/******  
EZASOKPS: PROC OPTIONS(MAIN);  
  
/* INCLUDE CBLOCK - common variables                             */  
% include CBLOCK;  
  
ID.TCPNAME = 'TCPIP';          /* Set TCP to use          */  
ID.ADSNAME = 'EZASOKPS';      /* and address space name */
```

```

open file(driver);

/*****
/*
/* Execute INITAPI
/*
*****/

/*****
/*
/* Uncomment this code to set max sockets to the maximum.
/*
/*
/* MAXSOC_INPUT = 65535;
/* MAXSOC_FWD = MAXSOC_INPUT;
*****/

call ezasocket(INITAPI, MAXSOC, ID, SUBTASK,
                MAXSNO, ERRNO, RETCODE);
if retcode < 0 then do;
    msg = 'FAIL: initapi' || errno;
    write file(driver) from (msg);
    goto getout;
end;

/*****
/*
/* Execute SOCKET
/*
*****/

call ezasocket(SOCKET, AF_INET, TYPE_STREAM, PROTO,
                ERRNO, RETCODE);
if retcode < 0 then do;
    msg = blank;
    msg = 'FAIL: socket, stream, internet' || errno;
    write file(driver) from (msg);
    goto getout;
end;
else sock_stream = retcode;

/*****
/*
/* Execute BIND
/*
*****/

name_id.port = 8888;
name_id.address = '01234567'BX; /* internet address
call ezasocket(BIND, SOCK_STREAM, NAME_ID,
                ERRNO, RETCODE);
if retcode < 0 then do;
    msg = blank;
    msg = 'FAIL: bind' || errno;
    write file(driver) from (msg);
    goto getout;
end;

/*****
/*
/* Execute GETSOCKNAME
/*
*****/

name_id.port = 8888;
name_id.address = '01234567'BX; /* internet address
call ezasocket(GETSOCKNAME, SOCK_STREAM,
                NAME_ID, ERRNO, RETCODE);
msg = blank;
if retcode < 0 then do;
    msg = 'FAIL: getsockname, stream, internet' || errno;
    write file(driver) from (msg);
end;
else do;
    msg = 'getsockname = ' || name_id.address;
    write file(driver) from (msg);
end;

/*****
/*
/* Execute LISTEN
/*
*****/

```

```

backlog = 5;
call ezasocket(LISTEN, SOCK_STREAM, BACKLOG,
               ERRNO, RETCODE);
if retcode < 0 then do;
    msg = blank;
    msg = 'FAIL: listen w/ backlog = 5' || errno;
    write file(driver) from (msg);
    goto getout;
end;

/*****
/*
/* Execute ACCEPT
/*
/*
*****/

name_id.port = 8888;
name_id.address = '01234567'BX;
call ezasocket(ACCEPT, SOCK_STREAM,
               NAME_ID, ERRNO, RETCODE);
msg = blank;
if retcode < 0 then do;
    msg = 'FAIL: accept' || errno;
    write file(driver) from (msg);
end;
else do;
    accpsock = retcode;
    msg = 'accept socket = ' || accpsock;
    write file(driver) from (msg);
end;

/*****
/*
/* Execute READ
/*
/*
*****/

nbyte = length(bufin);
call ezasocket(READ, ACCPSOCK,
               NBYTE, BUFIN, ERRNO, RETCODE);
msg = blank;
if retcode < 0 then do;
    msg = 'FAIL: read' || errno;
    write file(driver) from (msg);
end;
else do;
    msg = 'read = ' || bufin;
    write file(driver) from (msg);
    bufout = bufin;
    nbyte = retcode;
end;

/*****
/*
/* Execute WRITE
/*
/*
*****/

call ezasocket(WRITE, ACCPSOCK, NBYTE, BUFOUT,
               ERRNO, RETCODE);
msg = blank;
if retcode < 0 then do;
    msg = 'FAIL: write' || errno;
    write file(driver) from (msg);
end;
else do;
    msg = 'write = ' || bufout;
    write file(driver) from (msg);
end;

/*****
/*
/* Execute CLOSE accept socket
/*
/*
*****/

call ezasocket(CLOSE, ACCPSOCK,
               ERRNO, RETCODE);
if retcode < 0 then do;
    msg = blank;
    msg = 'FAIL: close, accept sock' || errno;

```

```

        write file(driver) from (msg);
end;

/*****
/*
/* Execute TERMAPI
/*
/*
*****/

getout:
call ezasocket(TERMAPI);

close file(driver);
end ezasokps;

```

Figure 131. EZASOKPS PL/I sample server program for IPv4

Sample program for IPv4 client program

The EZASOKPC PL/I sample program is a client program that shows you how to use the following calls provided by the call socket interface:

- CONNECT
- GETPEERNAME
- INITAPI
- READ
- SHUTDOWN
- SOCKET
- TERMAPI
- WRITE

```

/*****
/*
/* MODULE NAME:  EZASOKPC - THIS IS A VERY SIMPLE IPV4 CLIENT
/*
/* Copyright:    Licensed Materials - Property of IBM
/*
/*              "Restricted Materials of IBM"
/*
/*              5694-A01
/*
/*              (C) Copyright IBM Corp. 1994, 2002
/*
/*              US Government Users Restricted Rights -
/*              Use, duplication or disclosure restricted by
/*              GSA ADP Schedule Contract with IBM Corp.
/*
/* Status:       CSV1R4
/*
*****/
EZASOKPC: PROC OPTIONS(MAIN);

/* INCLUDE CBLOCK - common variables
% include CBLOCK;

ID.TCPNAME = 'TCPIP';           /* Set TCP to use
ID.ADSNAME = 'EZASOKPC';       /* and address space name
open file(driver);

/*****
/*
/* Execute INITAPI
/*
/*
*****/

call ezasocket(INITAPI, MAXSOC, ID, SUBTASK,
               MAXSNO, ERRNO, RETCODE);
if retcode < 0 then do;
    msg = 'FAIL: initapi' || errno;
    write file(driver) from (msg);
    goto getout;

```

```

end;

/*****
/*
/* Execute SOCKET
/*
/*
*****/

call ezasocket(SOCKET, AF_INET, TYPE_STREAM, PROTO,
               ERRNO, RETCODE);
if retcode < 0 then do;
    msg = blank; /* clear field */
    msg = 'FAIL: socket, stream, internet' || errno;
    write file(driver) from (msg);
    goto getout;
end;
sock_stream = retcode; /* save socket descriptor */

/*****
/* Execute CONNECT
/*
/*
*****/

name_id.port = 8888;
name_id.address = '01234567'BX; /* internet address */
call ezasocket(CONNECT, SOCK_STREAM, NAME_ID,
               ERRNO, RETCODE);
if retcode < 0 then do;
    msg = blank; /* clear field */
    msg = 'FAIL: connect, stream, internet' || errno;
    write file(driver) from (msg);
    goto getout;
end;

/*****
/*
/* Execute GETPEERNAME
/*
/*
*****/

call ezasocket(GETPEERNAME, SOCK_STREAM,
               NAME_ID, ERRNO, RETCODE);
msg = blank; /* clear field */
if retcode < 0 then do;
    msg = 'FAIL: getpeername' || errno;
    write file(driver) from (msg);
end;
else do;
    msg = 'getpeername =' || name_id.address;
    write file(driver) from (msg);
end;

/*****
/*
/* Execute WRITE
/*
/*
*****/

bufout = message;
nbyte = length(message);
call ezasocket(WRITE, SOCK_STREAM, NBYTE, BUFOUT,
               ERRNO, RETCODE);
msg = blank; /* clear field */
if retcode < 0 then do;
    msg = 'FAIL: write' || errno;
    write file(driver) from (msg);
end;
else do;
    msg = 'write = ' || bufout;
    write file(driver) from (msg);
end;

/*****
/*
/* Execute READ
/*
/*
*****/

nbyte = length(bufin);
call ezasocket(READ, SOCK_STREAM,
               NBYTE, BUFIN, ERRNO, RETCODE);
msg = blank; /* clear field */

```

```

if retcode < 0 then do;
    msg = 'FAIL: read' || errno;
    write file(driver) from (msg);
end;
else do;
    msg = 'read = ' || bufin;
    write file(driver) from (msg);
end;

/*****
/*
/* Execute SHUTDOWN from/to
/*
/*
*****/

getout:
how = 2;
call ezasocket(SHUTDOWN, SOCK_STREAM, HOW,
               ERRNO, RETCODE);
if retcode < 0 then do;
    msg = blank;
    msg = 'FAIL: shutdown' || errno;
    write file(driver) from (msg);
end;

/*****
/*
/* Execute TERMAPI
/*
/*
*****/

call ezasocket(TERMAPI);

close file(driver);
end ezasokpc;

```

Figure 132. EZASOKPC PL/1 sample client program for IPv4

Sample code for IPv6 server program

The EZASO6PS PL/I sample program is a server program that shows you how to use the following calls provided by the call socket interface:

- ACCEPT
- BIND
- CLOSE
- EZACIC09
- FREEADDRINFO
- GETADDRINFO
- GETHOSTNAME
- GETSOCKNAME
- INITAPI
- LISTEN
- NTOP
- PTON
- READ
- SOCKET
- TERMAPI
- WRITE

```

/*****
/*
/* MODULE NAME: EZASO6PS - THIS IS A VERY SIMPLE IPV6 SERVER
/*
/*
*****/

```

```

/* Copyright:    Licensed Materials - Property of IBM          */
/*                                                     */
/*             "Restricted Materials of IBM"                */
/*                                                     */
/*             5694-A01                                     */
/*                                                     */
/*             (C) Copyright IBM Corp. 2002, 2005          */
/*                                                     */
/*             US Government Users Restricted Rights -      */
/*             Use, duplication or disclosure restricted by  */
/*             GSA ADP Schedule Contract with IBM Corp.    */
/*                                                     */
/* Status:       CSV1R7                                     */
/*                                                     */
/*****
EZAS06PS: PROC OPTIONS(MAIN);

/* INCLUDE CBLOCK - common variables          */
% include CBLOCK;

ID.TCPNAME = 'TCPCS';          /* Set TCP to use          */
ID.ADSNAME = 'EZAS06PS';      /* and address space name */
open file(driver);

/*****
/*
/* Execute INITAPI
/*
/*****

/*****
/*
/* Uncomment this code to set max sockets to the maximum.
/*
/*
/* MAXSOC_INPUT = 65535;
/* MAXSOC_FWD = MAXSOC_INPUT;
/*****

call ezasocket(INITAPI, MAXSOC, ID, SUBTASK,
               MAXSNO, ERRNO, RETCODE);
if retcode < 0 then do;
    msg = 'FAIL: initapi' || errno;
    write file(driver) from (msg);
    goto getout;
end;

/*****
/*
/* Execute SOCKET
/*
/*****

call ezasocket(SOCKET, AF_INET6, TYPE_STREAM, PROTO,
               ERRNO, RETCODE);
if retcode < 0 then do;
    msg = blank;          /* clear field          */
    msg = 'FAIL: socket, stream, internet' || errno;
    write file(driver) from (msg);
    goto getout;
end;
else sock_stream = retcode;
/*****
/*
/* Execute PTON
/*
/*
/* PRESENTABLE_ADDR = IPV6_LOOPBACK;          /* Set IP address to use */
PRESENTABLE_ADDR_LEN = LENGTH(PRESENTABLE_ADDR) ; /* and its length */
call ezasocket(PTON, AF_INET6, PRESENTABLE_ADDR,
               PRESENTABLE_ADDR_LEN, NUMERIC_ADDR,
               ERRNO, RETCODE);
if retcode < 0 then do;
    msg = blank;          /* clear field          */
    msg = 'FAIL: pton' || errno;
    write file(driver) from (msg);
    goto getout;
end;
name6_id.address = NUMERIC_ADDR;          /* IPV6 internet address */
/*****
/*
/* Execute GETHOSTNAME
/*
/*****

```



```

/*****
call ezasocket(GETHOSTNAME, HOSTNAME_LEN, HOSTNAME,
                ERRNO, RETCODE);
msg = blank;                                /* clear field */
if retcode < 0 then do;
    msg = 'FAIL: gethostname' || errno;
    write file(driver) from (msg);
    goto getout;
end;
else do;
    msg = 'gethostname = ' || HOSTNAME;
    write file(driver) from (msg);
    GAI_NODE = HOSTNAME;    /* Set host name for getaddrinfo to use */
end;

/*****
/*
/* Execute GETADDRINFO
/*
/*
/*****
GAI_SERVLEN = 0;                                /* set service length */
GAI_HINTS.FLAGS = ai_CANONNAMEOK;    /* Request canonical name */
HINTS = ADDR(GAI_HINTS);    /* Set results pointer */
call ezasocket(GETADDRINFO,
                GAI_NODE, GAI_NODELEN,
                GAI_SERVICE, GAI_SERVLEN,
                HINTS, RES,
                CANONNAME_LEN,
                ERRNO, RETCODE);
msg = blank;                                /* clear field */
if retcode < 0 then do;
    msg = 'FAIL: getaddrinfo' || errno;
    write file(driver) from (msg);
end;
else do;                                /* process returned RES */

/*****
/*
/* Call EZACIC09 to format the returned result address information
/*
/*
/*****
call ezacic09(RES, OPNAMELEN, OPCANON, OPNAME, OPNEXT,
              RETCODE);
msg = blank;                                /* clear field */
if retcode ^= 0 then do;
    msg = 'FAIL: EZACIC09' || RETCODE;
    write file(driver) from (msg);
end;
else do;
    msg = 'OPCANON = ' || OPCANON;
    write file(driver) from (msg);
end;

/*****
/*
/* Execute FREEADDRINFO
/*
/*
/*****
call ezasocket(FREEADDRINFO, RES,
                ERRNO, RETCODE);
msg = blank;                                /* clear field */
if retcode < 0 then do;
    msg = 'FAIL: freeaddrinfo' || errno;
    write file(driver) from (msg);
end;

end;    /* end from getaddrinfo */
/*****
/*
/* Execute BIND
/*
/*
/*****

name6_id.port = 8888;
call ezasocket(BIND, SOCK_STREAM, NAME6_ID,
                ERRNO, RETCODE);
if retcode < 0 then do;
    msg = blank;                                /* clear field */
    msg = 'FAIL: bind' || errno;
    write file(driver) from (msg);
    goto getout;

```

```

end;

/*****
/*
/* Execute GETSOCKNAME
/*
/*
*****/

call ezasocket(GETSOCKNAME, SOCK_STREAM,
                NAME6_ID, ERRNO, RETCODE);
msg = blank; /* clear field */
if retcode < 0 then do;
    msg = 'FAIL: getsockname, stream, internet' || errno;
    write file(driver) from (msg);
end;

/*****
/*
/* Execute LISTEN
/*
/*
*****/

backlog = 5;
call ezasocket(LISTEN, SOCK_STREAM, BACKLOG,
                ERRNO, RETCODE);
if retcode < 0 then do;
    msg = blank; /* clear field */
    msg = 'FAIL: listen w/ backlog = 5' || errno;
    write file(driver) from (msg);
    goto getout;
end;

/*****
/*
/* Execute ACCEPT
/*
/*
*****/

call ezasocket(ACCEPT, SOCK_STREAM,
                NAME6_ID, ERRNO, RETCODE);
msg = blank; /* clear field */
if retcode < 0 then do;
    msg = 'FAIL: accept' || errno;
    write file(driver) from (msg);
end;
else do;
    accpsock = retcode;
    msg = 'accept socket = ' || accpsock;
    write file(driver) from (msg);
end;

/*****
/*
/* Execute NTOP
/*
/*
*****/

call ezasocket(NTOP, AF_INET6, NUMERIC_ADDR,
                PRESENTABLE_ADDR, PRESENTABLE_ADDR_LEN,
                ERRNO, RETCODE);
msg = blank; /* clear field */
if retcode < 0 then do;
    msg = 'FAIL: ntop' || errno;
    write file(driver) from (msg);
    goto getout;
end;
else do;
    msg = 'presentable address = ' || PRESENTABLE_ADDR;
    write file(driver) from (msg);
end; /*

/*****
/*
/* Execute READ
/*
/*
*****/

nbyte = length(bufin);
call ezasocket(READ, ACCPSOCK,
                NBYTE, BUFIN, ERRNO, RETCODE);
msg = blank; /* clear field */
if retcode < 0 then do;
    msg = 'FAIL: read' || errno;

```

```

        write file(driver) from (msg);
    end;
    else do;
        msg = 'read = ' || bufin;
        write file(driver) from (msg);
        bufout = bufin;
        nbyte = retcode;
    end;

/*****
/*
/* Execute WRITE
/*
/*
*****/

call ezasocket(WRITE, ACCPSOCK, NBYTE, BUFOUT,
                ERRNO, RETCODE);
msg = blank;                                /* clear field          */
if retcode < 0 then do;
    msg = 'FAIL: write' || errno;
    write file(driver) from (msg);
end;
else do;
    msg = 'write = ' || bufout;
    write file(driver) from (msg);
end;

/*****
/*
/* Execute CLOSE accept socket
/*
/*
*****/

call ezasocket(CLOSE, ACCPSOCK,
                ERRNO, RETCODE);
if retcode < 0 then do;
    msg = blank;                                /* clear field          */
    msg = 'FAIL: close, accept sock' || errno;
    write file(driver) from (msg);
end;

/*****
/*
/* Execute TERMAPI
/*
/*
*****/

getout:
call ezasocket(TERMAPI);

close file(driver);
end EZASO6PS;

```

Figure 133. EZASO6PS PL/1 sample server program for IPv6

Sample program for IPv6 client program

The EZASO6PC PL/I sample program is a client program that shows you how to use the following calls provided by the call socket interface:

- CONNECT
- GETNAMEINFO
- GETPEERNAME
- INITAPI
- PTON
- READ
- SHUTDOWN
- SOCKET
- TERMAPI
- WRITE

```

/*****
/*
/*  MODULE NAME:  EZAS06PC - THIS IS A VERY SIMPLE IPV6 CLIENT
/*
/*  Copyright:    Licensed Materials - Property of IBM
/*
/*                "Restricted Materials of IBM"
/*
/*                5694-A01
/*
/*                (C) Copyright IBM Corp. 2002
/*
/*                US Government Users Restricted Rights -
/*                Use, duplication or disclosure restricted by
/*                GSA ADP Schedule Contract with IBM Corp.
/*
/*  Status:       CSV1R4
/*
*****/
EZAS06PC: PROC OPTIONS(MAIN);

/* INCLUDE CBLOCK - common variables
% include CBLOCK;

ID.TCPNAME = 'TCPCS';           /* Set TCP to use
ID.ADSNAME = 'EZAS06PS';       /* and address space name
open file(driver);

/*****
/*
/*  Execute INITAPI
/*
*****/

call ezasocket(INITAPI, MAXSOC, ID, SUBTASK,
               MAXSNO, ERRNO, RETCODE);
if retcode < 0 then do;
    msg = 'FAIL: initapi' || errno;
    write file(driver) from (msg);
    goto getout;
end;

/*****
/*
/*  Execute SOCKET
/*
*****/

call ezasocket(SOCKET, AF_INET6, TYPE_STREAM, PROTO,
               ERRNO, RETCODE);
if retcode < 0 then do;
    msg = blank;           /* clear field
    msg = 'FAIL: socket, stream, internet' || errno;
    write file(driver) from (msg);
    goto getout;
end;
sock_stream = retcode;     /* save socket descriptor

/*****
/*  Execute PTON
/*
*****/
PRESENTABLE_ADDR = IPV6_LOOPBACK; /* Set the address to use
PRESENTABLE_ADDR_LEN = LENGTH(PRESENTABLE_ADDR) ; /* and it's length
call ezasocket(PTON, AF_INET6, PRESENTABLE_ADDR,
               PRESENTABLE_ADDR_LEN, NUMERIC_ADDR,
               ERRNO, RETCODE);
msg = blank;           /* clear field
if retcode < 0 then do;
    msg = 'FAIL: pton' || errno;
    write file(driver) from (msg);
    goto getout;
end;
msg = 'SUCCESS: pton converted ' || PRESENTABLE_ADDR;
name6_id.address = NUMERIC_ADDR; /* IPV6 internet address

/*****
/*  Execute CONNECT
/*
*****/

```

```

name6_id.port = 8888;
call ezasocket(CONNECT, SOCK_STREAM, NAME6_ID,
               ERRNO, RETCODE);
if retcode < 0 then do;
    msg = blank;                                /* clear field */
    msg = 'FAIL: connect, stream, internet' || errno;
    write file(driver) from (msg);
    goto getout;
end;

/*****
/*
/*  Execute GETPEERNAME
/*
/*
/* *****/

call ezasocket(GETPEERNAME, SOCK_STREAM,
               NAME6_ID, ERRNO, RETCODE);
msg = blank;                                /* clear field */
if retcode < 0 then do;
    msg = 'FAIL: getpeername' || errno;
    write file(driver) from (msg);
end;

/*****
/*
/*  Execute GETNAMEINFO
/*
/*
/* *****/

NAMELEN = 28 ;                                /* Set length of NAME */
GNI_HOST = blank;                             /* Clear Host name */
GNI_HOSTLEN = LENGTH(GNI_HOST); /* Set Host name length */
GNI_SERVICE = blank;                         /* Clear Service name */
GNI_SERVLEN = LENGTH(GNI_SERVICE); /* Set Service name length */
GNI_FLAGS = NI_NAMEREQD; /* Set an error if name is not found */
call ezasocket(GETNAMEINFO, NAME6_ID, NAMELEN,
               GNI_HOST, GNI_HOSTLEN,
               GNI_SERVICE, GNI_SERVLEN,
               GNI_FLAGS,
               ERRNO, RETCODE);
msg = blank;                                /* clear field */
if retcode < 0 then do;
    msg = 'FAIL: getnameinfo' || errno;
    write file(driver) from (msg);
end;
else do;
    msg = 'getnameinfo host=' || GNI_HOST ;
    write file(driver) from (msg);
    msg = 'getnameinfo service=' || GNI_SERVICE ;
    write file(driver) from (msg);
end;

/*****
/*
/*  Execute WRITE
/*
/*
/* *****/

bufout = message;
nbyte = length(message);
call ezasocket(WRITE, SOCK_STREAM, NBYTE, BUFOUT,
               ERRNO, RETCODE);
msg = blank;                                /* clear field */
if retcode < 0 then do;
    msg = 'FAIL: write' || errno;
    write file(driver) from (msg);
end;
else do;
    msg = 'write = ' || bufout;
    write file(driver) from (msg);
end;

/*****
/*
/*  Execute READ
/*
/*
/* *****/

nbyte = length(bufin);
call ezasocket(READ, SOCK_STREAM,

```

```

                                NBYTE, BUFIN, ERRNO, RETCODE);
msg = blank;                                /* clear field */
if retcode < 0 then do;
    msg = 'FAIL: read' || errno;
    write file(driver) from (msg);
end;
else do;
    msg = 'read = ' || bufin;
    write file(driver) from (msg);
end;

/*****
/*
/* Execute SHUTDOWN from/to
/*
/*
*****/

getout:
how = 2;
call ezasocket(SHUTDOWN, SOCK_STREAM, HOW,
               ERRNO, RETCODE);
if retcode < 0 then do;
    msg = blank;                                /* clear field */
    msg = 'FAIL: shutdown' || errno;
    write file(driver) from (msg);
end;

/*****
/*
/* Execute TERMAPI
/*
/*
*****/

call ezasocket(TERMAPI);

close file(driver);
end ezaso6pc;

```

Figure 134. EZASO6PC PL/I sample client program for IPv6

Common variables used in PL/I sample programs

The CBLOCK common storage area contains the variables that are used in the PL/I programs in this section.

```

/*****
/*
/* MODULE NAME:  CBLOCK - SOKET COMMON VARIABLES
/*
/* Copyright:    Licensed Materials - Property of IBM
/*
/*              "Restricted Materials of IBM"
/*
/*              5694-A01
/*
/*              Copyright IBM Corp. 1994, 2010
/*
/*              US Government Users Restricted Rights -
/*              Use, duplication or disclosure restricted by
/*              GSA ADP Schedule Contract with IBM Corp.
/*
/* Part Type:    Enterprise PL/1 for z/OS
/*
/* Status:       CSV1R12
/*
*****/
/*****
/*
/* SOKET COMMON VARIABLES
/*
/* Change Activity:
/* Flag Reason   Release   Date     Origin     Description
/* -----
/* $A1= PH39376   HIP6250   210830   tevalle:   Identify internal use
/*              only interfaces with
/*              IFF_RESTRICTED (87294)
/*
*****/

```

```

DCL ABS      BUILTIN;
DCL ADDR     BUILTIN;
DCL ACCEPT   CHAR(16) INIT('ACCEPT');
DCL ACPSOCK   FIXED BIN(15); /* temporary ACCEPT socket */
DCL AF_INET   FIXED BIN(31) INIT(2); /* internet domain */
DCL AF_INET6   FIXED BIN(31) INIT(19); /* internet v6 domain */
DCL AF_IUCV   FIXED BIN(31) INIT(17); /* iucv domain */
/* Mapping of GAI_HINTS/GAI_ADDRINFO FLAGS */
DCL ai_PASSIVE BIT(32) INIT('00000001'BX); /* flag: getaddrinfo hints */
DCL ai_CANONNAMEOK BIT(32) INIT('00000002'BX); /* flag: getaddrinfo hints */
DCL ai_NUMERICHOST BIT(32) INIT('00000004'BX); /* flag: getaddrinfo hints */
DCL ai_NUMERICSERV BIT(32) INIT('00000008'BX); /* flag: getaddrinfo hints */
DCL ai_V4MAPPED BIT(32) INIT('00000010'BX); /* flag: getaddrinfo hints */
DCL ai_ALL BIT(32) INIT('00000020'BX); /* flag: getaddrinfo hints */
DCL ai_ADDRCONFIG BIT(32) INIT('00000040'BX); /* flag: getaddrinfo hints */
DCL ai_EXTFLAGS BIT(32) INIT('00000080'BX); /* flag: getaddrinfo hints */
DCL ai_ALLFLAGMASK BIT(32) INIT('FFFFFF00'BX); /* flag: getaddrinfo hints */
DCL ALIAS CHAR(255); /* alternate NAME */
DCL APITYPE FIXED BIN(15) INIT(2); /* default API type */
DCL BACKLOG FIXED BIN(31); /* max length of pending queue */
DCL BADNAME CHAR(20); /* temporary name */
DCL BIND CHAR(16) INIT('BIND');
DCL BIND2ADDRSEL CHAR(16) INIT('BIND2ADDRSEL');
DCL BIT BUILTIN;
DCL BITZERO BIT(1); /* bit zero value */
DCL BLANK255 CHAR(255) INIT(' '); /* */
DCL BLANK CHAR(100) INIT(' '); /* */
DCL BUF CHAR(80) INIT(' '); /* macro READ/WRITE buffer */
DCL BUFF CHAR(15) INIT(' '); /* short buffer */
DCL BUFFER CHAR(32767) INIT(' '); /* BUFFER */
DCL BUFIN CHAR(32767) INIT(' '); /* Read buffer */
DCL BUFOUT CHAR(32767) INIT(' '); /* WRITE buffer */
DCL NCHBUFF CHAR(3200) INIT(' '); /* BUFFER */
DCL CANONNAME_LEN FIXED BIN(31); /* getaddrinfo canonical name length */
DCL 1 CLIENT, /* socket addr of connection peer */
    2 DOMAIN FIXED BIN(31) INIT(2), /* domain of client (AF_INET) */
    2 NAME CHAR(8) INIT(' '), /* addr identifier for client */
    2 TASK CHAR(8) INIT(' '), /* task identifier for client */
    2 RESERVED CHAR(20) INIT(' '); /* reserved */
DCL CLOSE CHAR(16) INIT('CLOSE');
DCL COMMAND FIXED BIN(31) INIT(3); /* Query FNDELAY flag */
DCL CONNECT CHAR(16) INIT('CONNECT');
DCL COUNT FIXED BIN(31) INIT(100); /* elements in GRP_IOCTL_TABLE */
DCL DATA_SOCKET FIXED BIN(15); /* temporary datagram socket */
DCL DEF FIXED BIN(31) INIT(0); /* default protocol */
DCL DONE_SENDING CHAR(1); /* ready flag */
DCL DRIVER FILE OUTPUT UNBUF ENV(FB RECSIZE(100)) RECORD;
DCL ERETMASK CHAR(4); /* indicate exception events */
DCL ERR FIXED BIN(31); /* error number variable */
DCL ERRNO FIXED BIN(31) INIT(0); /* error number */
DCL ESNDSK CHAR(4); /* check for pending exception events */
DCL EXIT LABEL; /* common exit point */
DCL EZACIC05 ENTRY OPTIONS(ASM,INTER) EXT; /* translate ascii>ebcdic */
DCL EZACIC09 ENTRY OPTIONS(ASM,INTER) EXT; /* format getaddrinfo res */
DCL EZASOKET ENTRY OPTIONS(ASM,INTER) EXT; /* socket call */
DCL FCNTL CHAR(16) INIT('FCNTL');
DCL FIONBIO BIT(32) INIT('8004A77E'BX); /* flag: nonblocking */
DCL FIONREAD BIT(32) INIT('4004A77F'BX); /* flag: #readable bytes */
DCL FLAGS FIXED BIN(31) INIT(0); /* default: no flags */
/* 1 = OOB, SEND OUT-OF-BAND */
/* 4 = DON'T ROUTE */
DCL FREEADDRINFO CHAR(16) INIT('FREEADDRINFO');
DCL GAI_NODE CHAR(255) INIT(' '); /* getaddrinfo node */
DCL GAI_NODELEN FIXED BIN(31) INIT(255); /* getaddrinfo node length */
DCL GAI_SERVICE CHAR(32) INIT(' '); /* getaddrinfo service */
DCL GAI_SERVLEN FIXED BIN(31) INIT(32); /* getaddrinfo service length */
DCL 1 GAI_HINTS, /* getaddrinfo hints addrinfo */
    2 FLAGS FIXED BIN(31) INIT(0), /* hints flags, see defs */
    /* starting at ai_PASSIVE */
    2 AF FIXED BIN(31) INIT(0), /* hints family */
    2 SOCTYPE FIXED BIN(31) INIT(0), /* hints socket type */

```

```

2 PROTO      FIXED BIN(31) INIT(0), /* hints protocol */
2 NAMELEN    FIXED BIN(31) INIT(0),
2 *          CHAR(4),
2 *          CHAR(4),
2 CANONNAME  FIXED BIN(31) INIT(0),
2 *          CHAR(4),
2 NAME       FIXED BIN(31) INIT(0),
2 *          CHAR(4),
2 NEXT       FIXED BIN(31) INIT(0),
2 EFLAGS     FIXED BIN(31) INIT(0); /* see definitions after */
/* IPV6_ADDR_PREFERENCES */
DCL 1 GAI_ADDRINFO BASED(RES), /* getaddrinfo RES addrinfo */
2 FLAGS      FIXED BIN(31), /* see ai_PASSIVE & following defs*/
2 AF         FIXED BIN(31),
2 SOCTYPE    FIXED BIN(31),
2 PROTO      FIXED BIN(31),
2 NAMELEN    FIXED BIN(31), /* RES socket address struct length*/
2 *          CHAR(4),
2 *          CHAR(4),
2 CANONNAME  POINTER, /* RES canonical name */
2 *          CHAR(4),
2 NAME       POINTER, /* RES socket address structure */
2 *          CHAR(4),
2 NEXT       POINTER, /* RES next addrinfo, zero if none.*/
2 EFLAGS     FIXED BIN(31); /* see definitions that follow the */
/* IPV6_ADDR_PREFERENCES definition*/
DCL 1 GAI_NAME_ID BASED(GAI_ADDRINFO.NAME),
2 LEN        BIT(8),
2 FAMILY     BIT(8),
2 PORT       BIT(16),
2 ADDRESS    BIT(32),
2 RESERVED1  CHAR(8);
DCL 1 GAI_NAME6_ID BASED(GAI_ADDRINFO.NAME),
2 LEN        BIT(8),
2 FAMILY     BIT(8),
2 PORT       BIT(16),
2 FLOWINFO   FIXED BIN(31),
2 ADDRESS    CHAR(16),
2 SCOPEID    FIXED BIN(31);
DCL GETADDRINFO CHAR(16) INIT('GETADDRINFO');
DCL GETCLIENTID CHAR(16) INIT('GETCLIENTID');
DCL GETHOSTBYADDR CHAR(16) INIT('GETHOSTBYADDR');
DCL GETHOSTBYNAME CHAR(16) INIT('GETHOSTBYNAME');
DCL GETHOSTNAME CHAR(16) INIT('GETHOSTNAME');
DCL GETHOSTID CHAR(16) INIT('GETHOSTID');
DCL GETIBMOPT CHAR(16) INIT('GETIBMOPT');
DCL GETNAMEINFO CHAR(16) INIT('GETNAMEINFO');
DCL GETPEERNAME CHAR(16) INIT('GETPEERNAME');
DCL GETSOCKNAME CHAR(16) INIT('GETSOCKNAME');
DCL GETSOCKOPT CHAR(16) INIT('GETSOCKOPT');
DCL GIVESOCKET CHAR(16) INIT('GIVESOCKET');
DCL GLOBAL CHAR(16) INIT('GLOBAL');
DCL GNI_FLAGS FIXED BIN(31); /* getnameinfo flags */
DCL GNI_HOST CHAR(255); /* getnameinfo host */
DCL GNI_HOSTLEN FIXED BIN(31); /* getnameinfo host length */
DCL GNI_SERVICE CHAR(32); /* getnameinfo service */
DCL GNI_SERVLEN FIXED BIN(31); /* getnameinfo service length */
DCL 1 GROUP_FILTER4 BASED, /* Group_Filter for IPv4 */
2 GF4_HEADER, /* Header portion */
3 GF4_INTERFACE FIXED BIN(31), /* Interface index */
3 * CHAR(4), /* Padding */
3 GF4_GROUP, /* Group Multi Address */
4 GF4 SOCK_LEN BIT(8), /* Socket len */
4 GF4 SOCK_FAMILY BIT(8), /* Socket family */
4 GF4 SOCK_SIN_PORT BIT(16), /* Socket port */
4 GF4 SOCK_SIN_ADDR BIT(32), /* Socket address */
4 GF4_RESERVED1 CHAR(8), /* Unused */
4 * CHAR(112), /* */
3 GF4_FMODE FIXED BIN(31), /* Filter mode */
3 GF4_NUMSRC FIXED BIN(31), /* Num of sources */
2 GF4_SLIST CHAR(0); /* Source list */
DCL 1 GF4_SRCENTRY BASED, /* Source Entry */
2 GF4_SRCADDR, /* Source IP address */
3 GF4 SOCK_LEN BIT(8), /* Socket len */
3 GF4 SOCK_FAMILY BIT(8), /* Socket family */
3 GF4 SOCK_SIN_PORT BIT(16), /* Socket port */
3 GF4 SOCK_SIN_ADDR BIT(32), /* Socket address */
3 GF4_RESERVED1 CHAR(8), /* Unused */
3 * CHAR(112); /* */
DCL 1 GROUP_FILTER6 BASED, /* Group_Filter for IPv6 */
2 GF6_HEADER, /* Header portion */

```



```

3 GF6_INTERFACE FIXED BIN(31), /* Interface index */
3 * CHAR(4), /* Padding */
3 GF6_GROUP, /* Group Multi Address */
4 GF6_SOCK_LEN BIT(8), /* Socket len */
4 GF6_SOCK_FAMILY BIT(8), /* Socket family */
4 GF6_SOCK_SIN6_PORT BIT(16), /* Socket port */
4 GF6_SOCK_SIN6_FLOWINFO FIXED BIN(31), /* flow info */
4 GF6_SOCK_SIN6_ADDRESS CHAR(16), /* Socket address */
4 GF6_SOCK_SIN6_SCOPEID FIXED BIN(31), /* Socket scopeid */
4 * CHAR(100), /*
3 GF6_FMODE FIXED BIN(31), /* Filter mode */
3 GF6_NUMSRC FIXED BIN(31), /* Num of sources */
2 GF6_SLIST CHAR(0); /* Source list */
DCL 1 GF6_SRCENTRY BASED, /* Source Entry */
2 GF6_SRCADDR, /* Source IP address */
3 GF6_SOCK_LEN BIT(8), /* Socket len */
3 GF6_SOCK_FAMILY BIT(8), /* Socket family */
3 GF6_SOCK_SIN6_PORT BIT(16), /* Socket port */
3 GF6_SOCK_SIN6_FLOWINFO FIXED BIN(31), /* flow info */
3 GF6_SOCK_SIN6_ADDRESS CHAR(16), /* Socket address */
3 GF6_SOCK_SIN6_SCOPEID FIXED BIN(31), /* Socket scopeid */
3 * CHAR(100); /*
DCL 1 GROUP_REQ4 BASED, /* Group Req for IPv4 */
2 GR4_INTERFACE FIXED BIN(31), /* Interface index */
2 * CHAR(4), /* Padding */
2 GR4_SOCK_LEN BIT(8), /* Socket len */
2 GR4_SOCK_FAMILY BIT(8), /* Socket family */
2 GR4_SOCK_SIN_PORT BIT(16), /* Socket port */
2 GR4_SOCK_SIN_ADDR BIT(32), /* Socket address */
2 GR4_RESERVED1 CHAR(8), /* Unused */
2 * CHAR(112); /*
DCL 1 GROUP_REQ6 BASED, /* Group Req for IPv6 */
2 GR6_INTERFACE FIXED BIN(31), /* Interface index */
2 * CHAR(4), /* Padding */
2 GR6_SOCK_LEN BIT(8), /* Socket len */
2 GR6_SOCK_FAMILY BIT(8), /* Socket family */
2 GR6_SOCK_SIN6_PORT BIT(16), /* Socket port */
2 GR6_SOCK_SIN6_FLOWINFO FIXED BIN(31), /* flow info */
2 GR6_SOCK_SIN6_ADDRESS CHAR(16), /* Socket address */
2 GR6_SOCK_SIN6_SCOPEID FIXED BIN(31), /* Socket scopeid */
2 * CHAR(100); /*
DCL 1 GROUP_SOURCE_REQ4 BASED, /* Group Source Req for IPv4 */
2 GSR4_INTERFACE FIXED BIN(31), /* Interface index */
2 * CHAR(4), /* Padding */
2 GSR4_GROUP, /* Multicast group addr */
3 GSR4_SOCK_LEN BIT(8), /* Socket len */
3 GSR4_SOCK_FAMILY BIT(8), /* Socket family */
3 GSR4_SOCK_SIN_PORT BIT(16), /* Socket port */
3 GSR4_SOCK_SIN_ADDR BIT(32), /* Socket address */
3 GSR4_RESERVED1 CHAR(8), /* Unused */
3 * CHAR(112), /*
2 GSR4_SOURCE, /* Source IP address */
3 GSR4_SOCK_LEN BIT(8), /* Socket len */
3 GSR4_SOCK_FAMILY BIT(8), /* Socket family */
3 GSR4_SOCK_SIN_PORT BIT(16), /* Socket port */
3 GSR4_SOCK_SIN_ADDR BIT(32), /* Socket address */
3 GSR4_RESERVED1 CHAR(8), /* Unused */
3 * CHAR(112); /*
DCL 1 GROUP_SOURCE_REQ6 BASED, /* Group Source Req for IPv6 */
2 GSR6_INTERFACE FIXED BIN(31), /* Interface index */
2 * CHAR(4), /* Padding */
2 GSR6_GROUP, /* Multicast group addr */
3 GSR6_SOCK_LEN BIT(8), /* Socket len */
3 GSR6_SOCK_FAMILY BIT(8), /* Socket family */
3 GSR6_SOCK_SIN6_PORT BIT(16), /* Socket port */
3 GSR6_SOCK_SIN6_FLOWINFO FIXED BIN(31), /* flow info */
3 GSR6_SOCK_SIN6_ADDRESS CHAR(16), /* Socket address */
3 GSR6_SOCK_SIN6_SCOPEID FIXED BIN(31), /* Socket scopeid */
3 * CHAR(100), /*
2 GSR6_SOURCE, /* Source IP address */
3 GSR6_SOCK_LEN BIT(8), /* Socket len */
3 GSR6_SOCK_FAMILY BIT(8), /* Socket family */
3 GSR6_SOCK_SIN6_PORT BIT(16), /* Socket port */
3 GSR6_SOCK_SIN6_FLOWINFO FIXED BIN(31), /* flow info */
3 GSR6_SOCK_SIN6_ADDRESS CHAR(16), /* Socket address */
3 GSR6_SOCK_SIN6_SCOPEID FIXED BIN(31), /* Socket scopeid */
3 * CHAR(100); /*
DCL HINTS POINTER; /*getaddrinfo hints addrinfo pointer*/
DCL 1 HOMEIF, /* Home Interface Structure */
2 ADDRESS CHAR(16); /* Home Interface Address */
DCL HOSTADDR BIT(32); /* host internet address */
DCL HOSTNAME CHAR(24); /* host name from GETHOSTNAME */

```

```

DCL HOSTNAME_LEN FIXED BIN(31) INIT(24);
/* host name length GETHOSTNAME */
DCL HOW          FIXED BIN(31) INIT(2); /* how shutdown is to be done */
Dcl 1 HOSTENT     Based, /* Host entry */
      3 H_NAME     POINTER, /* Official name of host */
      3 H_ALIASES   POINTER, /* Alias list address */
      3 H_ADDRTYPE  BIT(32), /* Host address type */
      3 H_LENGTH    FIXED BIN(31), /* Length of address */
      3 H_ADDR_LIST POINTER; /* List of addresses from */
/* name server */
DCL I             FIXED BIN(15); /* loop index */
DCL ICMP          FIXED BIN(31) INIT(2); /* prototype icmp ??? */
DCL 1 ID, /*
      2 TCPNAME CHAR(8) INIT('TCPIP'), /* remote address space */
      2 ADNSNAME CHAR(8) INIT('USER9'); /* local address space */
DCL IDENT         POINTER; /* TCP/IP Addr Space */
DCL IFCONF        CHAR(255); /* configuration structure */
DCL 1 IF_NAMEINDEX,
      2 IF_NIHEADER,
      3 IF_NITOTALIF FIXED BIN(31), /*Total Active Interfaces on Sys. */
      3 IF_NIENTRIES FIXED BIN(31), /* Number of entries returned */
      2 IF_NITABLE(10) CHAR(24);
DCL 1 IF_NAMEINDEXENTRY,
      2 IF_NIINDEX FIXED BIN(31), /* Interface Index */
      2 IF_NI_NAME CHAR(16), /* Interface Name, blank padded */
      2 IF_NIEXT,
      3 IF_NI_NAME_TERM CHAR(1), /* Null for C for Name len=16 */
      3 IF_RESERVED CHAR(3); /* Reserved */
DCL 1 IFREQ, /* Interface Structure */
      2 IFR_NAME CHAR(16), /* Interface Name, blank padded */
      2 IFR_IFR UNION,
      3 IFR_ADDR, /* Interface IP Address */
      4 IFR_ADDR_LEN BIT(8), /* Socket Len */
      4 IFR_ADDR_FAMILY BIT(8), /* Socket Family */
      4 IFR_ADDR_PORT BIT(16), /* Socket Port */
      4 IFR_ADDR_ADDR BIT(32), /* Socket Address */
      4 IFR_ADDR_RSVD CHAR(8), /* Socket Reserved */
      3 IFR_DSTADDR, /* Interface Dest IP Addr */
      4 IFR_DSTADDR_LEN BIT(8), /* Socket Len */
      4 IFR_DSTADDR_FAMILY BIT(8), /* Socket Family */
      4 IFR_DSTADDR_PORT BIT(16), /* Socket Port */
      4 IFR_DSTADDR_ADDR BIT(32), /* Socket Address */
      4 IFR_DSTADDR_RSVD CHAR(8), /* Socket Reserved */
      3 IFR_BROADCAST, /* Interface Broadcast IP Addr */
      4 IFR_BROADCAST_LEN BIT(8), /* Socket Len */
      4 IFR_BROADCAST_FAMILY BIT(8), /* Socket Family */
      4 IFR_BROADCAST_PORT BIT(16), /* Socket Port */
      4 IFR_BROADCAST_ADDR BIT(32), /* Socket Address */
      4 IFR_BROADCAST_RSVD CHAR(8), /* Socket Reserved */
      3 IFR_FLAGS BIT(16), /* Interface Flags */
      3 IFR_METRIC FIXED BIN(31), /* Interface Metric */
      3 IFR_DATA FIXED BIN(31), /* Interface Data */
      3 IFR_MTU FIXED BIN(31); /* Interface MTU */

/* The following constants are for use with the IFR_FLAGS field */
/* in structure IFREQ.
DCL IFF_UP BIT(16) INIT('0001'BX); /* interface is UP */
DCL IFF_BROADCAST BIT(16) INIT('0002'BX); /* broadcast addr valid */
DCL IFF_DEBUG BIT(16) INIT('0004'BX); /* turn on debugging */
DCL IFF_RESTRICTED BIT(16) INIT('0004'BX); /* overload debug for
restricted interfaces
since we don't allow
debugging @A1A*/
DCL IFF_LOOPBACK BIT(16) INIT('0008'BX); /* software loopback */
DCL IFF_POINTOPOINT BIT(16) INIT('0010'BX); /* point-to-point link */
DCL IFF_NOTRAILERS BIT(16) INIT('0020'BX); /* avoid use trailers */
DCL IFF_RUNNING BIT(16) INIT('0040'BX); /* resources allocated */
DCL IFF_NOARP BIT(16) INIT('0080'BX); /* no ARP */
DCL IFF_PROMISC BIT(16) INIT('0100'BX); /* receive all packets */
DCL IFF_ALLMULTI BIT(16) INIT('0200'BX); /* multicast packets */
DCL IFF_MULTICAST BIT(16) INIT('0400'BX); /* multicast capable */
DCL IFF_POINTOMULTIPT BIT(16) INIT('0800'BX); /* pt-to-multipoint */
DCL IFF_BRIDGE BIT(16) INIT('1000'BX); /* support token ring */
DCL IFF_SNAP BIT(16) INIT('2000'BX); /* support extended SAP */
DCL IFF_VIRTUAL BIT(16) INIT('4000'BX); /* virtual interface */
DCL IFF_SAMEHOST BIT(16) INIT('8000'BX); /* Samehost */

DCL INDEX         BUILTIN;
DCL IOCTL         CHAR(16) INIT('IOCTL');
DCL IOCTL_CMD     FIXED BIN(31); /* ioctl command */
DCL IOCTL_REQARG  POINTER; /* send pointer to data area */
DCL IOCTL_RETARG  POINTER; /* return pointer to data area */

```

```

DCL IOCTL_REQ000    FIXED BIN(31);          /* command request argument */
DCL IOCTL_REQ004    FIXED BIN(31);          /* command request argument */
DCL IOCTL_REQ008    FIXED BIN(31);          /* command request argument */
DCL IOCTL_REQ032    CHAR(32) INIT(' ');     /* command request argument */
DCL IOCTL_RET00     FIXED BIN(31);          /* command return argument */
DCL IOCTL_RET04     FIXED BIN(31);          /* command return argument */
DCL_INET6_IS_SRCADDR CHAR(16) INIT('INET6_IS_SRCADDR');
DCL_INITAPI CHAR(16) INIT('INITAPI'); /*
DCL_IP             FIXED BIN(31) INIT(1);   /* prototype ip   ??? */
DCL 1 IP_MREQ,
    2 IMR_MULTIADDR BIT(32),                /* IP multicast addr of group */
    2 IMR_INTERFACE BIT(32);                /* local IP addr of interface */
DCL 1 IPV6_MREQ,
    2 IPV6MR_MULTIADDR CHAR(16),
    2 IPV6MR_INTERFACE FIXED BIN(31);
DCL 1 IP_MREQ_SOURCE BASED,                 /* Multi source API structure */
    2 IMRS_MULTIADDR BIT(32),                /* IP multicast addr of grp */
    2 IMRS_SOURCEADDR BIT(32),               /* IP source addr */
    2 IMRS_INTERFACE BIT(32);               /* local IP addr of intf */
DCL 1 IP_MSFILTER BASED,                    /* IP_MsFilter */
    2 IMSF_HEADER,                           /* Header portion */
    3 IMSF_MULTIADDR BIT(32),                /* Multicast address */
    3 IMSF_INTERFACE BIT(32),               /* Interface address */
    3 IMSF_FMODE     FIXED BIN(31),         /* Filter mode */
    3 IMSF_NUMSRC     FIXED BIN(31),         /* Num of sources */
    2 IMSF_SLIST     CHAR(0);               /* Source list */
DCL 1 IMSF_SRCENTRY BASED,                  /* Source Entry */
    2 IMSF_SRCADDR BIT(32);                 /* Source IP address */
DCL_IP_MULTICAST_TTL BIT(32) INIT('00100003'BX);
DCL_IP_MULTICAST_LOOP BIT(32) INIT('00100004'BX);
DCL_IP_MULTICAST_IF BIT(32) INIT('00100007'BX);
DCL_IP_ADD_MEMBERSHIP BIT(32) INIT('00100005'BX);
DCL_IP_DROP_MEMBERSHIP BIT(32) INIT('00100006'BX);
DCL_IP_BLOCK_SOURCE BIT(32) INIT('0010000A'BX);
DCL_IP_UNBLOCK_SOURCE BIT(32) INIT('0010000B'BX);
DCL_IP_ADD_SOURCE_MEMBERSHIP BIT(32) INIT('0010000C'BX);
DCL_IP_DROP_SOURCE_MEMBERSHIP BIT(32) INIT('0010000D'BX);
DCL_IPRES     POINTER;                      /* EZACIC09 RES addrinfo ptr */
DCL_IPV6_ADDR_PREFERENCES BIT(32) INIT('00010020'BX);
/* getsockopt/setsockopt OPTNAME */
/*****
/* Mapping of GAI_HINTS/GAI_ADDRINFO EFLAGS flags and
/* IPV6_ADDR_PREFERENCES getsockopt, setsockopt OPTVAL flags, and
/* inet6_is_srcaddr flags
*****/
/* Prefer home IPv6 address over care-of IPv6 address
DCL_IPV6_PREFER_SRC_HOME     BIT(32) INIT('00000001'BX);
/* Prefer care-of IPv6 address over home IPv6 address
DCL_IPV6_PREFER_SRC_COA     BIT(32) INIT('00000002'BX);
/* Prefer temporary IPv6 address over public IPv6 address
DCL_IPV6_PREFER_SRC_TMP     BIT(32) INIT('00000004'BX);
/* Prefer public IPv6 address over temporary IPv6 address
DCL_IPV6_PREFER_SRC_PUBLIC BIT(32) INIT('00000008'BX);
/* Prefer cryptographic address over non-cryptographic address
DCL_IPV6_PREFER_SRC_CGA     BIT(32) INIT('00000010'BX);
/* Prefer non-cryptographic address over cryptographic address
DCL_IPV6_PREFER_SRC_NONCGA BIT(32) INIT('00000020'BX);
/* Invalid EFLAGS or IPV6_ADDR_PREFERENCES OPTVAL flags
DCL_IPV6_PREFERENCES_FLAGS_MASKINVALID BIT(32) INIT('FFFFFFC0'BX);
*****/
DCL_IPV6_JOIN_GROUP BIT(32) INIT('00010005'BX);
DCL_IPV6_LEAVE_GROUP BIT(32) INIT('00010006'BX);
DCL_IPV6_LOOPBACK CHAR(3) INIT('::1');
DCL_IPV6_MULTICAST_HOPS BIT(32) INIT('00010009'BX);
DCL_IPV6_MULTICAST_IF BIT(32) INIT('00010007'BX);
DCL_IPV6_MULTICAST_LOOP BIT(32) INIT('00010004'BX);
DCL_IPV6_UNICAST_HOPS BIT(32) INIT('00010003'BX);
DCL_IPV6_V6ONLY BIT(32) INIT('0001000A'BX);

```

```

DCL J      FIXED BIN(15);          /* loop index          */
DCL K      FIXED BIN(15);          /* loop index          */
DCL LENGTH BUILTIN;
DCL LABL    CHAR(9);
DCL LISTEN  CHAR(16) INIT('LISTEN');
DCL MAXSNO  FIXED BIN(31) INIT(0); /* max descriptor assigned */
DCL 1 MAXSOC_INPUT FIXED BIN(31) INIT(0);
DCL 1 MAXSOC_FWD,
  2 MAXSOC_IGNORE FIXED BIN(15) INIT(0),
  2 MAXSOC_FIXED BIN(15) INIT(255); /* largest sock # checked */
DCL MCAST_JOIN_GROUP BIT(32) INIT('00100028'BX);
/* getsockopt/setsockopt OPTNAME */
DCL MCAST_LEAVE_GROUP BIT(32) INIT('00100029'BX);
/* getsockopt/setsockopt OPTNAME */
DCL MCAST_JOIN_SOURCE_GROUP BIT(32) INIT('0010002A'BX);
/* getsockopt/setsockopt OPTNAME */
DCL MCAST_LEAVE_SOURCE_GROUP BIT(32) INIT('0010002B'BX);
/* getsockopt/setsockopt OPTNAME */
DCL MCAST_BLOCK_SOURCE BIT(32) INIT('0010002C'BX);
/* getsockopt/setsockopt OPTNAME */
DCL MCAST_UNBLOCK_SOURCE BIT(32) INIT('0010002D'BX);
/* getsockopt/setsockopt OPTNAME */
DCL MCAST_EXCLUDE BIT(32) INIT('00000001'BX);
DCL MCAST_INCLUDE BIT(32) INIT('00000000'BX);
DCL MCAST_NUMSRC_MAX BIT(32) INIT('00000040'BX);
DCL MESSAGE CHAR(50) INIT('I love my 1 @ Rottweiler!'); /* message */
DCL MSG      CHAR(100) INIT(' '); /* message text */
DCL 1 NAME_ID, /* socket addr of connection peer */
  2 FAMILY FIXED BIN(15) INIT(2), /*addr'g family TCP/IP def */
  2 PORT BIT(16), /* system assigned port # */
  2 ADDRESS BIT(32), /* 32-bit internet */
  2 RESERVED CHAR(8); /* reserved */
DCL 1 NAME6_ID, /* socket addr of connection peer */
  2 FAMILY FIXED BIN(15) INIT(19), /* NAMELN IGNORED & FAMILY */
  2 PORT BIT(16), /* port # */
  2 FLOWINFO FIXED BIN(31), /* Flow info */
  2 ADDRESS CHAR(16), /* IPv6 internet address */
  2 SCOPEID FIXED BIN(31); /* Scope ID */

DCL NAMEL CHAR(255) VARYING; /* name field, long */
DCL NAMES CHAR(24); /* name field, short */
DCL NAMELEN FIXED BIN(31); /* length of name/alias field */
DCL NBYTE FIXED BIN(31); /* Number of bytes in buffer */
DCL 1 NETCONFHDR, /* Network Configuration Hdr */
  2 NCHEYECATCHER CHAR(4) INIT('6NCH'), /* Eye Catcher '6NCH' */
  2 NCHIOCTL BIT(32) INIT('C014F608'BX),
/* The IOCTL being processed */
/* with this instance of the */
/* NetConfHdr. (RAS item) */
  2 NCHBUFFERLENGTH FIXED BIN(31) INIT(3200), /* Buffer Length */
  2 NCHBUFFERPTR POINTER, /* Buffer Pointer */
  2 NCHNUMENTRYRET FIXED BIN(31); /* Number of HomeIF returned via */
/* SIOCGHOMEIF6 or the number of */
/* GRT6RtEntry's returned via */
/* SIOCGRT6TABLE. */

DCL NI_NOFQDN FIXED BIN(31) INIT(1);
/* flag: getnameinfo */
DCL NI_NUMERICHOST FIXED BIN(31) INIT(2);
/* flag: getnameinfo */
DCL NI_NAMEREQD FIXED BIN(31) INIT(4);
/* flag: getnameinfo */
DCL NI_NUMERICSERV FIXED BIN(31) INIT(8);
/* flag: getnameinfo */
DCL NI_DGRAM FIXED BIN(31) INIT(16);
/* flag: getnameinfo */
DCL NI_NUMERICSCOPE FIXED BIN(31) INIT(32);
/* flag: getnameinfo */
DCL NOTE(3) CHAR(25) INIT('Now is the time for 198 g',
  'ood people to come to the',
  'aid of their parties!');

DCL NS      FIXED BIN(15); /* socket descriptor, new */
DCL NTOP    CHAR(16) INIT('NTOP'); /* Numeric to Presentation */
DCL NULL    BUILTIN;
DCL 1 NUMERIC_ADDR CHAR(16); /* NTOP/PTON Numeric address */
DCL OPNAMELEN FIXED BIN(31); /* Socket address structure length */
DCL OPCANON CHAR(256); /* Canonical name */
DCL OPNAME POINTER; /* Socket address structure */
DCL OPNEXT POINTER; /* Next result address info in chain */
DCL OPTL FIXED BIN(31); /* length of OPTVAL string */
DCL OPTLEN FIXED BIN(31); /* length of OPTVAL string */
DCL OPTN CHAR(15); /* OPTNAME value (macro) */
DCL OPTNAME FIXED BIN(31); /* OPTNAME value (call) */

```

```

DCL OPTVAL CHAR(255); /* GETSOCKOPT option data */
DCL OPTVALD FIXED BIN(31); /* SETSOCKOPT option data */
DCL 1 OPT_STRUC, /* structure for option */
    2 ON_OFF FIXED BIN(31) INIT(1), /* enable option */
    2 TIME FIXED BIN(31) INIT(5); /* time-out in seconds */
DCL 1 OPT_STRUCT, /* structure for option */
    2 ON FIXED BIN(31), /* used for getsockopt */
    2 TIMEOUT FIXED BIN(31); /* time-out in seconds */
DCL PLITEST BUILTIN; /* debug tool */
DCL PRESENTABLE_ADDR CHAR(45); /* NTOP/PTON presentable address */
DCL PRESENTABLE_ADDR_LEN FIXED BIN(15); /* NTOP/PTON presentable address length */
DCL PROTO FIXED BIN(31) INIT(0); /* prototype default */
DCL PTON CHAR(16) INIT('PTON'); /* Presentation to numeric */
DCL READ CHAR(16) INIT('READ');
DCL READV CHAR(16) INIT('READV');
DCL RECV CHAR(16) INIT('RECV');
DCL RECVFROM CHAR(16) INIT('RECVFROM');
DCL RECVMSG CHAR(16) INIT('RECVMSG');
DCL REUSE FIXED BIN(31) INIT('4'); /* toggle, reuse local addr */
DCL REQARG FIXED BIN(31); /* command request argument */
DCL RES POINTER; /* getaddrinfo RES addrinfo ptr */
DCL RETC FIXED BIN(31); /* return code variable */
DCL RETARG CHAR(255); /* return argument data area */
DCL RETCODE FIXED BIN(31) INIT(0); /* return code */
DCL RETLEN FIXED BIN(31); /* return area data length */
DCL RRETMASK CHAR(4); /* indicate READ EVENTS */
DCL RSNDSK CHAR(4); /* check for pending read events */
DCL RTENTRY CHAR(50) INIT('dummy table'); /* router entry */
DCL SAVEFAM FIXED BIN(15); /* temporary family name */
DCL SELECTB CHAR(4) INIT('1');
DCL SELECT CHAR(16) INIT('SELECT');
DCL SELECTEX CHAR(16) INIT('SELECTEX');
DCL SEND CHAR(16) INIT('SEND');
DCL SENDMSG CHAR(16) INIT('SENDMSG');
DCL SENDTO CHAR(16) INIT('SENDTO');
DCL SETADEYE1 CHAR(8) INIT('SETAPPLD');
DCL SETADVER FIXED BIN(15) INIT(1);
DCL SETADCONTLEN FIXED BIN(15) INIT(48);
DCL SETADBUFLN FIXED BIN(15) INIT(40);
DCL 1 SETAPPLDATA,
    2 SETAD_EYE1 CHAR(8),
    2 SETAD_VER FIXED BIN(15),
    2 SETAD_LEN FIXED BIN(15),
    2 * CHAR(4),
    2 SETAD_PTR64 ,
    3 SETAD_PTRHW CHAR(4),
    3 SETAD_PTR POINTER;
DCL SETADEYE2 CHAR(8) INIT('APPLDATA');
DCL 1 SETADCONTAINER,
    2 SETAD_EYE2 CHAR(8),
    2 SETAD_BUFFER CHAR(40);
DCL SETSOCKOPT CHAR(16) INIT('SETSOCKOPT');
DCL SHUTDOWN CHAR(16) INIT('SHUTDOWN');
DCL SIOCADDRT BIT(32) INIT('8030A70A' BX); /* flag: add routing entry */
DCL SIOCATMARK BIT(32) INIT('4004A707' BX); /* flag: out-of-band data */
DCL SIOCDELRT BIT(32) INIT('8030A70B' BX); /* flag: delete routing */
DCL SIOCGIFADDR BIT(32) INIT('C020A70D' BX); /* flag: network int addr */
DCL SIOCGHOMEIF6 BIT(32) INIT('C014F608' BX); /* flag: netw int config */
DCL SIOCGIFBRDADDR BIT(32) INIT('C020A712' BX); /* flag: net broadcast */
DCL SIOCGIFCONF BIT(32) INIT('C008A714' BX); /* flag: netw int config */
DCL SIOCGIFDSTADDR BIT(32) INIT('C020A70F' BX); /* flag: net des addr */
DCL SIOCGIFFLAGS BIT(32) INIT('C020A711' BX); /* flag: net intf flags */
DCL SIOCGIFMETRIC BIT(32) INIT('C020A717' BX); /* flag: get rout metr */
DCL SIOCGIFMTU BIT(32) INIT('C020A726' BX); /* flag: get intf mtu */
DCL SIOCGIFNAMEINDEX BIT(32) INIT('4000F603' BX); /* flag: name and indexes */
DCL SIOCGIFNETMASK BIT(32) INIT('C020A715' BX); /* flag: network mask */
DCL SIOCGIFNONSENSE BIT(32) INIT('B669FD2E' BX); /* flag: nonsense */
DCL SIOCSIFMETRIC BIT(32) INIT('8020A718' BX); /* flag: set rout metr */
DCL SIOCSAPPLDATA BIT(32) INIT('8018D90C' BX); /* Set APPLDATA */
DCL SIOCGIPMSFILTER BIT(32) INIT('C000A724' BX); /* flag: get multicast src filter */
DCL SIOCSIPMSFILTER BIT(32) INIT('8000A725' BX); /* flag: set multicast src filter */
DCL SIOCGMSFILTER BIT(32) INIT('C000F610' BX); /* flag: get multicast src filter */
DCL SIOCSMSFILTER BIT(32) INIT('8000F611' BX); /* flag: set multicast src filter */
/* The following constant is defined in EZBZTSL1, but is also */
/* included here for completeness. */
DCL SIOCTLCTL BIT(32) INIT('C038D90B' BX);

```

```

/* flag: ttls */
/* The following constants are defined in EZBPINF1, but is also */
/* included here for completeness. */
/* DCL SIOCSPARTNERINFO BIT(32) INIT('8004F613'BX); */
/* DCL SIOCGPARTNERINFO BIT(32) INIT('C000F612'BX); */
/* flag: PartnerInfo */
DCL SOCK FIXED BIN(15); /* socket descriptor */
DCL SOCKET CHAR(16) INIT('SOCKET');
DCL SOCK_DATAGRAM FIXED BIN(15); /* socket descriptor datagram */
DCL SOCK_RAW FIXED BIN(15); /* socket descriptor raw */
DCL SOCK_STREAM FIXED BIN(15); /* stream socket descriptor */
DCL SOCK_STREAM_1 FIXED BIN(15); /* stream socket descriptor */
DCL SO_BROADCAST FIXED BIN(31) INIT(32); /* toggle, broadcast msg */
DCL SO_ERROR FIXED BIN(31) INIT(4103); /* check/clear async error */
DCL SO_KEEPALIVE FIXED BIN(31) INIT(8); /* request status of stream */
DCL SO_LINGER FIXED BIN(31) INIT(128); /* toggle, linger on close */
DCL SO_OOBLINER FIXED BIN(31) INIT(256); /* toggle, out-of-bound data */
DCL SO_RCVTIMEO BIT(32) INIT('00001006'BX);
DCL SO_REUSEADDR FIXED
    BIN(31) INIT(4); /* toggle, local address reuse */
DCL SO_SNDBUF FIXED BIN(31) INIT(4097);
DCL SO_SNDTIMEO BIT(32) INIT('00001005'BX);
DCL SO_TYPE FIXED BIN(31) INIT(4104); /* return type of socket */
DCL STRING BUILTIN;
DCL SUBSTR BUILTIN;
DCL SUBTASK CHAR(8) INIT('ANYNAME'); /* task/path identifier */
DCL SYNC CHAR(16) INIT('SYNC');
DCL TAKESOCKET CHAR(16) INIT('TAKESOCKET');
DCL TASK CHAR(16) INIT('TASK');
DCL TERMAPI CHAR(16) INIT('TERMAPI'); /*
DCL TIME BUILTIN;
DCL 1 TIMEOUT,
    2 TIME_SEC FIXED BIN(31), /* value in secs */
    2 TIME_MSEC FIXED BIN(31); /* value in millisecs */
DCL 1 TIMEVAL,
    2 TV_SEC BIT(32), /* value in secs */
    2 TV_USEC BIT(32); /* value in microseconds */
DCL TYPE_DATAGRAM FIXED BIN(31) INIT(2); /* fixed length connectionless */
DCL TYPE_RAW FIXED BIN(31) INIT(3); /* internal protocol interface */
DCL TYPE_STREAM FIXED BIN(31) INIT(1); /* two-way byte stream */
DCL WRETMASK CHAR(4); /* indicate WRITE EVENTS */
DCL WRITE CHAR(16) INIT('WRITE');
DCL WRITEV CHAR(16) INIT('WRITEV');
DCL WNDMSK CHAR(4); /* check for pending write events */
DCL TCP_KEEPALIVE BIT(32) INIT('80000008'BX);
DCL TCP_NODELAY BIT(32) INIT('80000001'BX);

```

Figure 135. CBLOCK PL/1 common variables

Common variables used in COBOL sample programs

The EZACOBOL common storage area contains the variables that are used in the COBOL programs in this section.

```

*****
*
*   MODULE NAME:  EZACOBOL - COBOL COMMON VARIABLES
*
*   Copyright:    Licensed Materials - Property of IBM
*
*               "Restricted Materials of IBM"
*
*               5694-A01
*
*               Copyright IBM Corp. 2007, 2010
*
*               US Government Users Restricted Rights -
*               Use, duplication or disclosure restricted by
*               GSA ADP Schedule Contract with IBM Corp.
*
*   Note:         COBOL variable names can contain a maximum of
*               30 characters.
*
*   Status:       CSV1R12
*
*   Change Activity:
*   Flag Reason  Release  Date    Origin    Description

```

```

* ----- *
* $A1= PH39376 HIP6250 210830 tevallor: Identify internal use *
* only interfaces with *
* IFF_RESTRICTED (87294)*
*
*****
*****
*
* COBOL COMMON VARIABLES
*
*****
*
* Socket option values.
*
01 IP-ADD-MEMBERSHIP PIC X(4) VALUE X'00100005'.
01 IP-ADD-SOURCE-MEMBERSHIP PIC X(4) VALUE X'0010000C'.
01 IP-BLOCK-SOURCE PIC X(4) VALUE X'0010000A'.
01 IP-DROP-MEMBERSHIP PIC X(4) VALUE X'00100006'.
01 IP-DROP-SOURCE-MEMBERSHIP PIC X(4) VALUE X'0010000D'.
01 IP-MULTICAST-IF PIC X(4) VALUE X'00100007'.
01 IP-MULTICAST-LOOP PIC X(4) VALUE X'00100004'.
01 IP-MULTICAST-TTL PIC X(4) VALUE X'00100003'.
01 IP-UNBLOCK-SOURCE PIC X(4) VALUE X'0010000B'.
01 IPV6-ADDR-PREFERENCES PIC X(4) VALUE X'00010020'.
01 IPV6-JOIN-GROUP PIC X(4) VALUE X'00010005'.
01 IPV6-LEAVE-GROUP PIC X(4) VALUE X'00010006'.
01 IPV6-MULTICAST-HOPS PIC X(4) VALUE X'00010009'.
01 IPV6-MULTICAST-IF PIC X(4) VALUE X'00010007'.
01 IPV6-MULTICAST-LOOP PIC X(4) VALUE X'00010004'.
01 IPV6-UNICAST-HOPS PIC X(4) VALUE X'00010003'.
01 IPV6-V6ONLY PIC X(4) VALUE X'0001000A'.
01 MCAST-BLOCK-SOURCE PIC X(4) VALUE X'0010002C'.
01 MCAST-JOIN-GROUP PIC X(4) VALUE X'00100028'.
01 MCAST-JOIN-SOURCE-GROUP PIC X(4) VALUE X'0010002A'.
01 MCAST-LEAVE-GROUP PIC X(4) VALUE X'00100029'.
01 MCAST-LEAVE-SOURCE-GROUP PIC X(4) VALUE X'0010002B'.
01 MCAST-UNBLOCK-SOURCE PIC X(4) VALUE X'0010002D'.
01 SO-RCVTIMEO PIC X(4) VALUE X'00001006'.
01 SO-SNDTIMEO PIC X(4) VALUE X'00001005'.
*
* IOCTL Commands
*
01 SIOCGIFMTU PIC X(4) VALUE X'C020A726'.
01 SIOCGIPMSFILTER PIC X(4) VALUE X'C000A724'.
01 SIOCSIPMSFILTER PIC X(4) VALUE X'8000A725'.
01 SIOCGMSFILTER PIC X(4) VALUE X'C000F610'.
01 SIOCSMSFILTER PIC X(4) VALUE X'8000F611'.
01 SIOCSAPPLDATA PIC X(4) VALUE X'8018D90C'.
*
* Structure allows applications to allocate space for
* either form of inet socket address
*
01 SOCKADDR-STORAGE.
   05 SS-LEN PIC X(1).
   05 SS-FAMILY PIC X(1).
   05 SS-DATA PIC X(126).
*
* IP-MREQ for IP_ADD_MEMBERSHIP and IP_DROP_MEMBERSHIP
*
01 IP-MREQ.
   05 IMR-MULTIADDR PIC 9(8) BINARY.
   05 IMR-INTERFACE PIC 9(8) BINARY.
*
* IP-MREQ-SOURCE for
* IP_ADD_SOURCE_MEMBERSHIP
* IP_DROP_SOURCE_MEMBERSHIP
* IP_BLOCK_SOURCE
* IP_UNBLOCK_SOURCE
*
01 IP-MREQ-SOURCE.
   05 IMR-MULTIADDR PIC 9(8) BINARY.
   05 IMR-SOURCEADDR PIC 9(8) BINARY.
   05 IMR-INTERFACE PIC 9(8) BINARY.
*
* IPV6-MREQ for IPV6_JOIN_GROUP and IPV6_LEAVE_GROUP
*
01 IPV6-MREQ.
   05 IPV6MR-MULTIADDR.
      10 FILLER PIC 9(16) BINARY.
      10 FILLER PIC 9(16) BINARY.
   05 IPV6MR-INTERFACE PIC 9(8) BINARY.
*

```

```

* GROUP-REQ for
* MCAST_JOIN_GROUP
* MCAST_LEAVE_GROUP
*
01 GROUP-REQ.
05 GR-INTERFACE PIC 9(8) BINARY.
05 FILLER PIC X(4).
05 GR-GROUP PIC X(128).
05 GR-GROUP-R REDEFINES GR-GROUP.
10 GR-GROUP-SOCK-LEN PIC X(1).
10 GR-GROUP-SOCK-FAMILY PIC X(1).
10 GR-GROUP-SOCK-DATA PIC X(26).
10 GR-GROUP-SOCK-SIN REDEFINES GR-GROUP-SOCK-DATA.
15 GR-GROUP-SOCK-SIN-PORT PIC 9(4) BINARY.
15 GR-GROUP-SOCK-SIN-ADDR PIC 9(8) BINARY.
15 FILLER PIC X(8).
15 FILLER PIC X(12).
10 GR-GROUP-SOCK-SIN6 REDEFINES GR-GROUP-SOCK-DATA.
15 GR-GROUP-SOCK-SIN6-PORT PIC 9(4) BINARY.
15 GR-GROUP-SOCK-SIN6-FLOWINFO PIC 9(8) BINARY.
15 GR-GROUP-SOCK-SIN6-ADDR.
20 FILLER PIC 9(16) BINARY.
20 FILLER PIC 9(16) BINARY.
15 GR-GROUP-SOCK-SIN6-SCOPEID PIC 9(8) BINARY.
10 FILLER PIC X(100).

*
* GROUP-SOURCE-REQ for
* MCAST_BLOCK_SOURCE
* MCAST_UNBLOCK_SOURCE
* MCAST_JOIN_SOURCE_GROUP
* MCAST_LEAVE_SOURCE_GROUP
*
01 GROUP-SOURCE-REQ.
05 GSR-INTERFACE PIC 9(8) BINARY.
05 FILLER PIC X(4).
05 GSR-GROUP PIC X(128).
05 GSR-GROUP-R REDEFINES GSR-GROUP.
10 GSR-GROUP-SOCK-LEN PIC X(1).
10 GSR-GROUP-SOCK-FAMILY PIC X(1).
10 GSR-GROUP-SOCK-DATA PIC X(26).
10 GSR-GROUP-SOCK-SIN REDEFINES GSR-GROUP-SOCK-DATA.
15 GSR-GROUP-SOCK-SIN-PORT PIC 9(4) BINARY.
15 GSR-GROUP-SOCK-SIN-ADDR PIC 9(8) BINARY.
15 FILLER PIC X(8).
15 FILLER PIC X(12).
10 GSR-GROUP-SOCK-SIN6 REDEFINES GSR-GROUP-SOCK-DATA.
15 GSR-GROUP-SOCK-SIN6-PORT PIC 9(4) BINARY.
15 GSR-GROUP-SOCK-SIN6-FLOWINFO PIC 9(8) BINARY.
15 GSR-GROUP-SOCK-SIN6-ADDR.
20 FILLER PIC 9(16) BINARY.
20 FILLER PIC 9(16) BINARY.
15 GSR-GROUP-SOCK-SIN6-SCOPEID PIC 9(8) BINARY.
10 FILLER PIC X(100).
05 GSR-SOURCE PIC X(128).
05 GSR-SOURCE-R REDEFINES GSR-SOURCE.
10 GSR-SOURCE-SOCK-LEN PIC X(1).
10 GSR-SOURCE-SOCK-FAMILY PIC X(1).
10 GSR-SOURCE-SOCK-DATA PIC X(26).
10 GSR-SOURCE-SOCK-SIN REDEFINES GSR-SOURCE-SOCK-DATA.
15 GSR-SOURCE-SOCK-SIN-PORT PIC 9(4) BINARY.
15 GSR-SOURCE-SOCK-SIN-ADDR PIC 9(8) BINARY.
15 FILLER PIC X(8).
15 FILLER PIC X(12).
10 GSR-SOURCE-SOCK-SIN6 REDEFINES GSR-SOURCE-SOCK-DATA.
15 GSR-SOURCE-SOCK-SIN6-PORT PIC 9(4) BINARY.
15 GSR-SOURCE-SOCK-SIN6-FLOWINFO PIC 9(8) BINARY.
15 GSR-SOURCE-SOCK-SIN6-ADDR.
20 FILLER PIC 9(16) BINARY.
20 FILLER PIC 9(16) BINARY.
15 GSR-SOURCE-SOCK-SIN6-SCOPEID PIC 9(8) BINARY.
10 FILLER PIC X(100).

*
* MULTICAST CONSTANTS
*
77 MCAST-INCLUDE PIC 9(8) BINARY VALUE 0.
77 MCAST-EXCLUDE PIC 9(8) BINARY VALUE 1.
77 MCAST-NUMSRC-MAX PIC 9(8) BINARY VALUE 64.

*
* IP-MSFILTER
*
01 IP-MSFILTER.
02 IMSF-HEADER.

```



```

03 IMSF-MULTIADDR      PIC 9(8) BINARY.
03 IMSF-INTERFACE      PIC 9(8) BINARY.
03 IMSF-FMODE          PIC 9(8) BINARY.
    88 IMSF-FMODE-INCLUDE VALUE 0.
    88 IMSF-FMODE-EXCLUDE VALUE 1.
03 IMSF-NUMSRC         PIC 9(8) BINARY.
02 IMSF-SLIST.
    03 IMSF-SRCENTRY    OCCURS 1 TO 64 TIMES
                        DEPENDING ON IMSF-NUMSRC.
                        PIC 9(8) BINARY.
    05 IMSF-SRCADDR     PIC 9(8) BINARY.
*
* GROUP-FILTER
*
01 GROUP-FILTER.
    02 GF-HEADER.
        03 GF-INTERFACE      PIC 9(8) BINARY.
        03 FILLER            PIC X(4).
        03 GF-GROUP          PIC X(128).
        03 GF-GROUP-R        REDEFINES GF-GROUP.
            05 GF-GROUP-SOCK-LEN PIC X(1).
            05 GF-GROUP-SOCK-FAMILY PIC X(1).
            05 GF-GROUP-SOCK-DATA PIC X(26).
            05 GF-GROUP-SOCK-SIN REDEFINES GF-GROUP-SOCK-DATA.
                10 GF-GROUP-SOCK-SIN-PORT PIC 9(4) BINARY.
                10 GF-GROUP-SOCK-SIN-ADDR PIC 9(8) BINARY.
                10 FILLER                PIC X(8).
                10 FILLER                PIC X(12).
            05 GF-GROUP-SOCK-SIN6 REDEFINES GF-GROUP-SOCK-DATA.
                10 GF-GROUP-SOCK-SIN6-PORT PIC 9(4) BINARY.
                10 GF-GROUP-SOCK-SIN6-FLOWINFO PIC 9(8) BINARY.
                10 GF-GROUP-SOCK-SIN6-ADDR.
                    15 FILLER            PIC 9(16) BINARY.
                    15 FILLER            PIC 9(16) BINARY.
                10 GF-GROUP-SOCK-SIN6-SCOPEID PIC 9(8) BINARY.
            05 FILLER            PIC X(100).
        03 GF-FMODE            PIC 9(8) BINARY.
            88 GF-FMODE-INCLUDE VALUE 0.
            88 GF-FMODE-EXCLUDE VALUE 1.
        03 GF-NUMSRC          PIC 9(8) BINARY.
    02 GF-SLIST.
        03 GF-SRCENTRY        OCCURS 1 TO 64 TIMES
                            DEPENDING ON GF-NUMSRC.
                            PIC X(128).
        05 GF-SRCADDR         PIC X(128).
        05 GF-SRCADDR-R       REDEFINES GF-SRCADDR.
            10 GF-SLIST-SOCK-LEN PIC X(1).
            10 GF-SLIST-SOCK-FAMILY PIC X(1).
            10 GF-SLIST-SOCK-DATA PIC X(26).
            10 GF-SLIST-SOCK-SIN REDEFINES GF-SLIST-SOCK-DATA.
                15 GF-SLIST-SOCK-SIN-PORT PIC 9(4) BINARY.
                15 GF-SLIST-SOCK-SIN-ADDR PIC 9(8) BINARY.
                15 FILLER                PIC X(8).
                15 FILLER                PIC X(12).
            10 GF-SLIST-SOCK-SIN6 REDEFINES GF-SLIST-SOCK-DATA.
                15 GF-SLIST-SOCK-SIN6-PORT PIC 9(4) BINARY.
                15 GF-SLIST-SOCK-SIN6-FLOWINFO PIC 9(8) BINARY.
                15 GF-SLIST-SOCK-SIN6-ADDR.
                    20 FILLER            PIC 9(16) BINARY.
                    20 FILLER            PIC 9(16) BINARY.
                15 GF-SLIST-SOCK-SIN6-SCOPEID PIC 9(8) BINARY.
            10 FILLER            PIC X(100).
*
* Structure for setting APPLDATA when using the SIOCSAPPLDATA
* ioctl.
*
77 SETADEYE1           PIC X(8) VALUE 'SETAPPLD'.
77 SETADVER            PIC 9(4) BINARY VALUE 1.
01 SETAPPLDATA.
    02 SETAD-EYE1       PIC X(8).
    02 SETAD-VER        PIC 9(4) BINARY.
    02 SETAD-LEN        PIC 9(4) BINARY.
    02 FILLER           PIC X(4).
    02 SETAD-PTR64      PIC 9(16) BINARY.
    02 SETAD-PTR31 REDEFINES SETAD-PTR64.
    03 SETAD-PTRHW      PIC 9(8) BINARY.
    03 SETAD-PTR        USAGE IS POINTER.
*
* Structure for containing the actual application data being set
* by the SIOCSAPPLDATA ioctl.
*
77 SETADEYE2           PIC X(8) VALUE 'APPLDATA'.
01 SETADCONTAINER.
    02 SETAD-EYE2       PIC X(8).

```

```

    02 SETAD-BUFFER          PIC X(40).
*
* TIMEVAL for SO_RCVTIMEO and SO_SNDTIMEO
*
01 TIMEVAL.
    02 TV-SEC                PIC 9(8) BINARY.
    02 TV-USEC               PIC 9(8) BINARY.
*
* IFREQ structure for SIOCGIFxxxx ioctls.
*
01 IFREQ.
    05 IFR-NAME              PIC X(16).
    05 IFR-IFR               PIC X(16).
    05 IFR-ADDR              REDEFINES IFR-IFR.
        10 IFR-ADDR-LEN      PIC X(1).
        10 IFR-ADDR-FAMILY   PIC X(1).
        10 IFR-ADDR-PORT     PIC 9(4) BINARY.
        10 IFR-ADDR-ADDR     PIC 9(8) BINARY.
        10 FILLER            PIC X(8).
    05 IFR-DSTADDR           REDEFINES IFR-IFR.
        10 IFR-DSTADDR-LEN   PIC X(1).
        10 IFR-DSTADDR-FAMILY PIC X(1).
        10 IFR-DSTADDR-PORT  PIC 9(4) BINARY.
        10 IFR-DSTADDR-ADDR  PIC 9(8) BINARY.
        10 FILLER            PIC X(8).
    05 IFR-BROADADDR         REDEFINES IFR-IFR.
        10 IFR-BROADADDR-LEN PIC X(1).
        10 IFR-BROADADDR-FAMILY PIC X(1).
        10 IFR-BROADADDR-PORT PIC 9(4) BINARY.
        10 IFR-BROADADDR-ADDR PIC 9(8) BINARY.
        10 FILLER            PIC X(8).
    05 IFR-FLAGS-R           REDEFINES IFR-IFR.
        10 IFR-FLAGS         PIC X(2).
        10 FILLER            PIC X(14).
    05 IFR-METRIC-R          REDEFINES IFR-IFR.
        10 IFR-METRIC        PIC 9(8) BINARY.
        10 FILLER            PIC X(12).
    05 IFR-DATA-R            REDEFINES IFR-IFR.
        10 IFR-DATA          PIC 9(8) BINARY.
        10 FILLER            PIC X(12).
    05 IFR-MTU-R             REDEFINES IFR-IFR.
        10 IFR-MTU           PIC 9(8) BINARY.
        10 FILLER            PIC X(12).
*
* Constants for use with the IFR_FLAGS field in structure IFREQ.
*
01 IFF-UP                    PIC X(2) VALUE X'0001'.
01 IFF-BROADCAST             PIC X(2) VALUE X'0002'.
01 IFF-DEBBUG                PIC X(2) VALUE X'0004'.
01 IFF-RESTRICTED            PIC X(2) VALUE X'0004'.
01 IFF-LOOPBACK              PIC X(2) VALUE X'0008'.
01 IFF-POINTOPOINT           PIC X(2) VALUE X'0010'.
01 IFF-NOTRAILERS            PIC X(2) VALUE X'0020'.
01 IFF-RUNNING                PIC X(2) VALUE X'0040'.
01 IFF-NOARP                 PIC X(2) VALUE X'0080'.
01 IFF-PROMISC                PIC X(2) VALUE X'0100'.
01 IFF-ALLMULTI              PIC X(2) VALUE X'0200'.
01 IFF-MULTICAST              PIC X(2) VALUE X'0400'.
01 IFF-POINTMULTIPT          PIC X(2) VALUE X'0800'.
01 IFF-BRIDGE                PIC X(2) VALUE X'1000'.
01 IFF-SNAP                   PIC X(2) VALUE X'2000'.
01 IFF-VIRTUAL                PIC X(2) VALUE X'4000'.
01 IFF-SAMEHOST              PIC X(2) VALUE X'8000'.
*
* HOSTENT structure
*
01 HOSTENT.
* Official name of host
    03 H-NAME                 PIC S9(8) BINARY.
* Alias list address
    03 H-ALIASES              PIC S9(8) BINARY.
* Host address type
    03 H-ADDRTYPE             PIC S9(8) BINARY.
* Length of address
    03 H-LENGTH               PIC S9(8) BINARY.
* List of addresses from name server
    03 H-ADDR-LIST            PIC S9(8) BINARY.
*
* Address information structure
*
01 ADDRINFO.

```

```

* Flags
  03 AI-FLAGS PIC S9(8) BINARY.
* Socket family
  03 AI-FAMILY PIC S9(8) BINARY.
* Socket type
  03 AI-SOCKETTYPE PIC S9(8) BINARY.
* Protocol
  03 AI-PROTOCOL PIC S9(8) BINARY.
* Length of AI-ADDR value
  03 AI-ADDRLLEN PIC S9(8) BINARY.
* Pad to double word boundary
  03 FILLER PIC X(4).
  03 FILLER PIC X(4).
* Canonical name
  03 AI-CANONNAME PIC S9(8) BINARY.
  03 FILLER PIC X(4).
* Binary address, sockaddr_in(6)
  03 AI-ADDR PIC S9(8) BINARY.
  03 FILLER PIC X(4).
* Next addrinfo structure
  03 AI-NEXT PIC S9(8) BINARY.
* Extended flags
  03 AI-EFLAGS PIC S9(8) BINARY.
*
* AI-FLAGS mappings
*
77 AI-PASSIVE PIC X(4) VALUE X'00000001'.
77 AI-PASSIVE-BIT PIC S9(8) BINARY VALUE 1.
77 AI-CANONNAMEOK PIC X(4) VALUE X'00000002'.
77 AI-CANONNAMEOK-BIT PIC S9(8) BINARY VALUE 2.
77 AI-NUMERICHOST PIC X(4) VALUE X'00000004'.
77 AI-NUMERICHOST-BIT PIC S9(8) BINARY VALUE 4.
77 AI-NUMERICSERV PIC X(4) VALUE X'00000008'.
77 AI-NUMERICSERV-BIT PIC S9(8) BINARY VALUE 8.
77 AI-V4MAPPED PIC X(4) VALUE X'00000010'.
77 AI-V4MAPPED-BIT PIC S9(8) BINARY VALUE 16.
77 AI-ALL PIC X(4) VALUE X'00000020'.
77 AI-ALL-BIT PIC S9(8) BINARY VALUE 32.
77 AI-ADDRCONFIG PIC X(4) VALUE X'00000040'.
77 AI-ADDRCONFIG-BIT PIC S9(8) BINARY VALUE 64.
77 AI-EXTFLAGS PIC X(4) VALUE X'00000080'.
77 AI-EXTFLAGS-BIT PIC S9(8) BINARY VALUE 128.
77 AI-ALLFLAGMASK PIC X(4) VALUE X'FFFFFFFF'.
77 AI-ALLFLAGMASK-BITS PIC S9(8) VALUE -256.
*
* AI-EFLAGS mappings
* Also maps OPTVAL for getsockopt and setsockopt when
* OPTNAME is IPV6-ADDR-PREFERENCES
* Also maps FLAGS for inet6_is_srcaddr
*
77 IPV6-PREFER-SRC-HOME PIC S9(8) BINARY VALUE 1.
77 IPV6-PREFER-SRC-COA PIC S9(8) BINARY VALUE 2.
77 IPV6-PREFER-SRC-TMP PIC S9(8) BINARY VALUE 4.
77 IPV6-PREFER-SRC-PUBLIC PIC S9(8) BINARY VALUE 8.
77 IPV6-PREFER-SRC-CGA PIC S9(8) BINARY VALUE 16.
77 IPV6-PREFER-SRC-NONCGA PIC S9(8) BINARY VALUE 32.
77 IPV6-PREFER-SRC-INVALIDBITS PIC S9(8) BINARY VALUE -64.
*
* NI_FLAGS mappings
*
77 NI-NOFQDN PIC X(4) VALUE X'00000001'.
77 NI-NUMERICHOST PIC X(4) VALUE X'00000002'.
77 NI-NAMEREQD PIC X(4) VALUE X'00000004'.
77 NI-NUMERICSERV PIC X(4) VALUE X'00000008'.
77 NI-DGRAM PIC X(4) VALUE X'00000010'.
77 NI-NUMERICSCOPE PIC X(4) VALUE X'00000020'.
*
* End of EZACOBOL - COBOL COMMON VARIABLES
*
*****

```

Figure 136. EZACOBOL COBOL common variables

COBOL call interface sample IPv6 server program

The EZASO6CS program is a server program that shows you how to use the following calls provided by the call socket interface:

- ACCEPT

- BIND
- CLOSE
- EZACIC09
- FREEADDRINFO
- GETADDRINFO
- GETCLIENTID
- GETHOSTNAME
- INITAPI
- LISTEN
- NTOP
- PTON
- READ
- SOCKET
- TERMAPI
- WRITE

```

*****
*
*   MODULE NAME:  EZAS06CS - THIS IS A VERY SIMPLE IPV6 SERVER
*
*   Copyright:    Licensed Materials - Property of IBM
*
*               "Restricted Materials of IBM"
*
*               5694-A01
*
*               Copyright IBM Corp. 2002, 2008
*
*               US Government Users Restricted Rights -
*               Use, duplication or disclosure restricted by
*               GSA ADP Schedule Contract with IBM Corp.
*
*   Note:        COBOL variable names can contain a maximum of
*               30 characters.
*
*   Status:      CSV1R10
*
*   LANGUAGE:    COBOL
*
*****
Identification Division.
*****

Program-id. EZAS06CS.

*****
Environment Division.
*****

*****
Data Division.
*****

Working-storage Section.
*-----*
* Socket interface function codes
*-----*
01  socket-functions.
    02 socket-accept      pic x(16) value 'ACCEPT'      '.
    02 socket-bind       pic x(16) value 'BIND'         '.
    02 socket-close      pic x(16) value 'CLOSE'        '.
    02 socket-connect    pic x(16) value 'CONNECT'      '.
    02 socket-fcntl      pic x(16) value 'FCNTL'        '.
    02 socket-freeaddrinfo pic x(16) value 'FREEADDRINFO' '.

```

```

02 soket-getaddrinfo      pic x(16) value 'GETADDRINFO'
02 soket-getclientid      pic x(16) value 'GETCLIENTID'
02 soket-gethostbyaddr    pic x(16) value 'GETHOSTBYADDR'
02 soket-gethostbyname    pic x(16) value 'GETHOSTBYNAME'
02 soket-gethostid        pic x(16) value 'GETHOSTID'
02 soket-gethostname      pic x(16) value 'GETHOSTNAME'
02 soket-getnameinfo      pic x(16) value 'GETNAMEINFO'
02 soket-getpeername      pic x(16) value 'GETPEERNAME'
02 soket-getsockname      pic x(16) value 'GETSOCKNAME'
02 soket-getsockopt       pic x(16) value 'GETSOCKOPT'
02 soket-givesocket       pic x(16) value 'GIVESOCKET'
02 soket-initapi          pic x(16) value 'INITAPI'
02 soket-ioctl            pic x(16) value 'IOCTL'
02 soket-listen           pic x(16) value 'LISTEN'
02 soket-ntop             pic x(16) value 'NTOP'
02 soket-pton             pic x(16) value 'PTON'
02 soket-read             pic x(16) value 'READ'
02 soket-recv             pic x(16) value 'RECV'
02 soket-recvfrom         pic x(16) value 'RECVFROM'
02 soket-select           pic x(16) value 'SELECT'
02 soket-send             pic x(16) value 'SEND'
02 soket-sendto           pic x(16) value 'SENDTO'
02 soket-setsockopt       pic x(16) value 'SETSOCKOPT'
02 soket-shutdown         pic x(16) value 'SHUTDOWN'
02 soket-socket           pic x(16) value 'SOCKET'
02 soket-takesocket       pic x(16) value 'TAKESOCKET'
02 soket-termapi          pic x(16) value 'TERMAPI'
02 soket-write            pic x(16) value 'WRITE'
*-----*
* Work variables                                           *
*-----*
01 errno                 pic 9(8) binary value zero.
01 retcode                pic s9(8) binary value zero.
01 client-ipaddr-dotted   pic x(15) value space.
01 server-ipaddr-dotted   pic x(15) value space.
01 ezaconn-function       pic x value space.
88 CONNECTED             value 'Y'.
01 saved-message-id       pic x(8) value space.
88 close-down-message-received value '*CLSDWN*'.
01 Terminate-Options      pic x value space.
88 Opened-API             value 'A'.
88 Opened-Socket          value 'S'.
01 saved-message-id-len   pic 9(8) Binary value 8.
01 Cur-time .
02 Hour                  pic 9(2).
02 Minute                pic 9(2).
02 Second                 pic 9(2).
02 Hund-Sec              pic 9(2).
01 S                     pic 9(4) comp.
*-----*
* Variables used for the INITAPI call                       *
*-----*
01 maxsoc-fwd             pic 9(8) Binary.
01 maxsoc-rdf redefines maxsoc-fwd.
02 filler                 pic x(2).
02 maxsoc                 pic 9(4) Binary.
01 initapi-ident.
05 tcpname                pic x(8) Value 'TCPCS '.
05 asname                 pic x(8) Value space.
01 subtask                pic x(8) value 'EZAS06CS'.
01 maxsno                 pic 9(8) Binary Value 1.
*-----*
* Variables returned by the GETCLIENTID Call               *
*-----*
01 clientid.
05 clientid-domain        pic 9(8) Binary value 19.
05 clientid-name          pic x(8) value space.
05 clientid-task          pic x(8) value space.
05 filler                 pic x(20) value low-value.
*-----*
* Variables used for the SOCKET call                       *
*-----*
01 AF-INET                pic 9(8) Binary Value 2.
01 AF-INET6               pic 9(8) Binary Value 19.
01 SOCK-STREAM            pic 9(8) Binary Value 1.
01 SOCK-DATAGRAM          pic 9(8) Binary Value 2.
01 SOCK-RAW               pic 9(8) Binary Value 3.
01 IPPROTO-IP             pic 9(8) Binary Value zero.
01 IPPROTO-TCP            pic 9(8) Binary Value 6.
01 IPPROTO-UDP            pic 9(8) Binary Value 17.
01 IPPROTO-IPV6           pic 9(8) Binary Value 41.
01 socket-descriptor      pic 9(4) Binary Value zero.

```

```

*-----*
* Variables returned by the GETHOSTNAME Call *
*-----*
01 host-name-len          pic 9(8) binary.
01 host-name              pic x(24).
01 host-name-char-count   pic 9(4) binary.
01 host-name-unstrung      pic x(24) value spaces.
*-----*
* Variables used/returned by the GETADDRINFO Call *
*-----*
01 node-name              pic x(255).
01 node-name-len          pic 9(8) binary.
01 service-name           pic x(32).
01 service-name-len       pic 9(8) binary.
01 canonical-name-len     pic 9(8) binary.
01 ai-passive             pic 9(8) binary value 1.
01 ai-canonnameok         pic 9(8) binary value 2.
01 ai-numerichost         pic 9(8) binary value 4.
01 ai-numericserver       pic 9(8) binary value 8.
01 ai-v4mapped            pic 9(8) binary value 16.
01 ai-all                pic 9(8) binary value 32.
01 ai-addrconfig          pic 9(8) binary value 64.
*-----*
* Variables used for the BIND call *
*-----*
01 server-socket-address.
   05 server-family       pic 9(4) Binary value 19.
   05 server-port         pic 9(4) Binary value 1031.
   05 server-flowinfo     pic 9(8) Binary value 0.
   05 server-ipaddr.
      10 filler           pic 9(16) Binary value 0.
      10 filler           pic 9(16) Binary value 0.
   05 server-scopeid      pic 9(8) Binary value 0.
01 NBYTE                  PIC 9(8) COMP value 80.
01 BUF                    PIC X(80).
01 BACKLOG                PIC S9(8) COMP VALUE 10.
*-----*
* Variables used/returned by the EZACIC09 call *
*-----*
01 input-addrinfo-ptr     usage is pointer.
01 output-name-len        pic 9(8) binary.
01 output-canonical-name  pic x(256).
01 output-name            usage is pointer.
01 output-next-addrinfo   usage is pointer.
*-----*
* Variables used for the LISTEN call *
*-----*
01 backlog-level          pic 9(4) Binary Value zero.
*-----*
* Variables used for the ACCEPT call *
*-----*
01 socket-descriptor-new  pic 9(4) Binary Value zero.
*-----*
* Variables used for the NTOP/PTON call *
*-----*
01 IN6ADDR-ANY            pic x(45)
                           value '::'.
01 IN6ADDR-LOOPBACK       pic x(45)
                           value '::1'.
01 ntop-family            pic 9(8) Binary.
01 pton-family            pic 9(8) Binary.
01 presentable-addr       pic x(45) value spaces.
01 presentable-addr-len   pic 9(4) Binary value 45.
01 numeric-addr.
   05 filler              pic 9(16) Binary Value 0.
   05 filler              pic 9(16) Binary Value 0.
*-----*
* Variables used by the RECV Call *
*-----*
01 client-socket-address.
   05 client-family       pic 9(4) Binary Value 19.
   05 client-port         pic 9(4) Binary Value 1032.
   05 client-flowinfo     pic 9(8) Binary Value zero.
   05 client-ipaddr.
      10 filler           pic 9(16) Binary Value 0.
      10 filler           pic 9(16) Binary Value 0.
   05 client-scopeid      pic 9(8) Binary Value zero.
*-----*
* Buffer and length field for recv and send operation *
*-----*
01 send-request-len       pic 9(8) Binary Value zero.
01 read-request-len       pic 9(8) Binary Value zero.

```

```

01 read-buffer                                pic x(4000) value space.
01 filler redefines read-buffer.
   05 message-id                             pic x(8).
   05 filler                                  pic x(3992).
*-----*
* recv and send flags                         *
*-----*
01 send-flag                                pic 9(8) Binary value zero.
01 recv-flag                               pic 9(8) Binary value zero.
*-----*
* Error message for socket interface errors   *
*-----*
77 failure                                  pic S9(8) comp.
01 ezaerror-msg.
   05 filler                                  pic x(9) Value 'Function='.
   05 ezaerror-function                     pic x(16) Value space.
   05 filler                                  pic x value ' '.
   05 filler                                  pic x(8) Value 'Retcode='.
   05 ezaerror-retcode                      pic ---99.
   05 filler                                  pic x value ' '.
   05 filler                                  pic x(9) Value 'Errorno='.
   05 ezaerror-errno                        pic zzz99.
   05 filler                                  pic x value ' '.
   05 ezaerror-text                         pic x(50) value ' '.

*=====
Linkage Section.
*=====
01 L1.
   03 hints-addrinfo.
       05 hints-ai-flags                    pic 9(8) binary.
       05 hints-ai-family                   pic 9(8) binary.
       05 hints-ai-socktype                 pic 9(8) binary.
       05 hints-ai-protocol                 pic 9(8) binary.
       05 filler                           pic 9(8) binary.
       05 filler                           pic 9(8) binary.
       05 filler                           pic 9(8) binary.
       05 filler                           pic 9(8) binary.
   03 hints-addrinfo-ptr                   usage is pointer.
   03 results-addrinfo-ptr                 usage is pointer.

*
* Results address info
*
01 results-addrinfo.
   05 results-ai-flags                    pic 9(8) binary.
   05 results-ai-family                   pic 9(8) binary.
   05 results-ai-socktype                 pic 9(8) binary.
   05 results-ai-protocol                 pic 9(8) binary.
   05 results-ai-addr-len                 pic 9(8) binary.
   05 results-ai-canonical-name           usage is pointer.
   05 results-ai-addr-ptr                 usage is pointer.
   05 results-ai-next-ptr                 usage is pointer.

*
* Socket address structure from EZACIC09.
*
01 output-name-ptr                        usage is pointer.
01 output-ip-name.
   03 output-ip-family                    pic 9(4) Binary.
   03 output-ip-port                      pic 9(4) Binary.
   03 output-ip-sock-data                 pic x(24).
   03 output-ipv4-sock-data redefines
      output-ip-sock-data.
       05 output-ipv4-ipaddr              pic 9(8) Binary.
       05 filler                          pic x(20).
   03 output-ipv6-sock-data redefines
      output-ip-sock-data.
       05 output-ipv6-flowinfo            pic 9(8) Binary.
       05 output-ipv6-ipaddr.
          10 filler                       pic 9(16) Binary.
          10 filler                       pic 9(16) Binary.
       05 output-ipv6-scopeid             pic 9(8) Binary.

*=====
Procedure Division using L1.
*=====

*~~~~~*
*          P R O C E D U R E      C O N T R O L S          *
*~~~~~*

Perform Initialize-API                    thru    Initialize-API-Exit.
Perform Get-ClientID                     thru    Get-ClientID-Exit.

```

```

Perform Sockets-Descriptor thru Sockets-Descriptor-Exit.
Perform Presentation-To-Numeric thru Presentation-To-Numeric-Exit.
Perform Get-Host-Name thru Get-Host-Name-Exit.
Perform Get-Address-Info thru Get-Address-Info-Exit.
Perform Bind-Socket thru Bind-Socket-Exit.
Perform Listen-To-Socket thru Listen-To-Socket-Exit.
Perform Accept-Connection thru Accept-Connection-Exit.
Move 45 to presentable-addr-len.
Move spaces to presentable-addr.
Move server-ipaddr to numeric-addr.
Move 19 to ntop-family.
Perform Numeric-TO-Presentation thru Numeric-To-Presentation-Exit.
Perform Read-Message thru Read-Message-Exit.
Perform Write-Message thru Write-Message-Exit.
Perform Close-Socket thru Exit-Now.

*-----*
* Initialize socket API *
*-----*
Initialize-API.
Move soket-initapi to ezaerror-function.
*-----*
* If you want to set maxsoc to the max, uncomment the next line.*
*-----*
* Move 65535 to maxsoc-fwd.
Call 'EZASOCKET' using soket-initapi maxsoc initapi-ident
subtask maxsno errno retcode.
Move 'Initapi failed' to ezaerror-text.
If retcode < 0 move 12 to failure.
Perform Return-Code-Check thru Return-Code-Exit.
Move 'A' to Terminate-Options.
Initialize-API-Exit.
Exit.

*-----*
* Let us see the client-id *
*-----*
Get-ClientID.
move soket-getclientid to ezaerror-function.
Call 'EZASOCKET' using soket-getclientid clientid errno
retcode.
Display 'Client ID = ' clientid-name
'task=' clientid-task.
Move 'Getclientid failed' to ezaerror-text.
If retcode < 0 move 24 to failure.
Perform Return-Code-Check thru Return-Code-Exit.
Get-ClientID-Exit.
Exit.

*-----*
* Get us a stream socket descriptor. *
*-----*
Sockets-Descriptor.
move soket-socket to ezaerror-function.
Call 'EZASOCKET' using soket-socket AF-INET6 SOCK-STREAM
IPPROTO-IP errno retcode.
Move 'Socket call failed' to ezaerror-text.
If retcode < 0 move 24 to failure.
Perform Return-Code-Check thru Return-Code-Exit.
Move retcode to socket-descriptor.
Move 'S' to Terminate-Options.
Sockets-Descriptor-Exit.
Exit.

*-----*
* Use PTON to create an IP address to bind to. *
*-----*
Presentation-To-Numeric.
move soket-pton to ezaerror-function.
move IN6ADDR-LOOPBACK to presentable-addr.
Call 'EZASOCKET' using soket-pton AF-INET6
presentable-addr presentable-addr-len
numeric-addr
errno retcode.
Move 'PTON call failed' to ezaerror-text.
If retcode < 0 move 24 to failure.
Perform Return-Code-Check thru Return-Code-Exit.
move numeric-addr to server-ipaddr.
Presentation-To-Numeric-Exit.
Exit.

```



```

*-----*
* Get the host name.                                     *
*-----*
Get-Host-Name.
    move soket-gethostname to ezaerror-function.
    move 24 to host-name-len.
    Call 'EZASOKET' using soket-gethostname
        host-name-len host-name
        errno retcode.
    display 'Host name = ' host-name.
    Move 'GETHOSTNAME call failed' to ezaerror-text.
    If retcode < 0 move 24 to failure.
    Perform Return-Code-Check thru Return-Code-Exit.
Get-Host-Name-Exit.
    Exit.

*-----*
* Get address information                                 *
*-----*
Get-Address-Info.
    move soket-getaddrinfo to ezaerror-function.
    move 0 to host-name-char-count.
    inspect host-name tallying host-name-char-count
        for characters before x'00'.
    unstring host-name delimited by x'00'
        into host-name-unstring
        count in host-name-char-count.
    string host-name-unstring delimited by ' '
        into node-name.
    move host-name-char-count to node-name-len
    display 'node-name-len: ' node-name-len.
    move spaces to service-name.
    move 0 to service-name-len.
    move 0 to hints-ai-family.
    move ai-canonnameok to hints-ai-flags
    move 0 to hints-ai-socktype.
    move 0 to hints-ai-protocol.
    display 'GETADDRINFO Input fields: '
    display 'Node name = ' node-name.
    display 'Node name length = ' node-name-len.
    display 'Service name = ' service-name.
    display 'Service name length = ' service-name-len.
    display 'Hints family = ' hints-ai-family.
    display 'Hints flags = ' hints-ai-flags.
    display 'Hints socktype = ' hints-ai-socktype.
    display 'Hints protocol = ' hints-ai-protocol.
    set address of results-addrinfo to results-addrinfo-ptr.
    move soket-getaddrinfo to ezaerror-function.
    set hints-addrinfo-ptr to address of hints-addrinfo.
    Call 'EZASOKET' using soket-getaddrinfo
        node-name node-name-len
        service-name service-name-len
        hints-addrinfo-ptr
        results-addrinfo-ptr
        canonical-name-len
        errno retcode.
    Move 'GETADDRINFO call failed' to ezaerror-text.
    If retcode < 0 move 24 to failure
        Perform Return-Code-Check thru Return-Code-Exit
    else
        Perform Return-Code-Check thru Return-Code-Exit
        display 'Address of results addrinfo is '
            results-addrinfo-ptr.
        set address of results-addrinfo to results-addrinfo-ptr
        set input-addrinfo-ptr to address of results-addrinfo
        display 'Address of input-addrinfo-ptr is '
            input-addrinfo-ptr.
        Perform Format-Result-AI thru Format-Result-AI-Exit
        Perform Set-Next-Addrinfo thru
            Set-Next-Addrinfo-Exit until
                output-next-addrinfo is equal to NULLS
        Perform Free-Address-Info thru Free-Address-Info-Exit.
Get-Address-Info-Exit.
    Exit.

*-----*
* Set next addrinfo address                             *
*-----*
Set-Next-Addrinfo.
    display 'Setting next addrinfo address as '
        results-ai-next-ptr.

```

```

        display 'Address of output-next-addrinfo as '
            output-next-addrinfo.
        set address of results-addrinfo to output-next-addrinfo.
        set input-addrinfo-ptr to address of results-addrinfo.
        display 'Address of input-addrinfo-ptr is '
            input-addrinfo-ptr.
        Perform Format-Result-AI thru Format-Result-AI-Exit.
Set-Next-Addrinfo-Exit.
Exit.

*-----*
* Format result address information                                     *
*-----*
Format-Result-AI.
    move 'EZACIC09' to ezaerror-function.
    move zeros to output-name-len.
    move spaces to output-canonical-name.
    set output-name to nulls.
    set output-next-addrinfo to nulls.
    Call 'EZACIC09' using input-addrinfo-ptr
        output-name-len
        output-canonical-name
        output-name
        output-next-addrinfo
        retcode.
    Move 'EZACIC09 call failed' to ezaerror-text.
    display 'EZACIC09 output:'
    display 'Canonical name = ' output-canonical-name.
    display 'name length   = ' output-name-len.
    display 'name         = ' output-name.
    display 'next addrinfo = ' output-next-addrinfo.
    If retcode < 0 move 24 to failure.
    Perform Return-Code-Check thru Return-Code-Exit.
    display 'Formatting result address ip address'.
    set address of output-ip-name to output-name.
    move results-ai-family to ntop-family.
    display 'ntop-family = ' ntop-family.
    if ntop-family = AF-INET then
        display 'Formatting ipv4 addres'
        move output-ipv4-ipaddr to numeric-addr
        move 16 to presentable-addr-len
    else
        display 'Formatting ipv6 addres'
        move output-ipv6-ipaddr to numeric-addr
        move 45 to presentable-addr-len.
    move spaces to presentable-addr.
    Perform Numeric-To-Presentation thru
        Numeric-To-Presentation-Exit.
Format-Result-AI-Exit.
Exit.

*-----*
* Release resolver storage                                           *
*-----*
Free-Address-Info.
    move soket-freeaddrinfo to ezaerror-function.
    Call 'EZASOKET' using soket-freeaddrinfo
        results-addrinfo-ptr
        errno retcode.
    Move 'FREEADDRINFO call failed' to ezaerror-text.
    If retcode < 0 move 24 to failure.
    Perform Return-Code-Check thru Return-Code-Exit.
Free-Address-Info-Exit.
Exit.

*-----*
* Bind socket to our server port number                             *
*-----*
Bind-Socket.
    Move soket-bind to ezaerror-function.
    Call 'EZASOKET' using soket-bind socket-descriptor
        server-socket-address errno retcode.
    Display 'Port = ' server-port
        ' Address = ' presentable-addr.
    Move 'Bind call failed' to ezaerror-text
    If retcode < 0 move 24 to failure.
    Perform Return-Code-Check thru Return-Code-Exit.
Bind-Socket-Exit.
Exit.

*-----*
* Listen to the socket                                              *
*-----*

```

```

*-----*
Listen-To-Socket.
    Move socket-listen to ezaerror-function.
    Call 'EZASOCKET' using socket-listen socket-descriptor
        backlog errno retcode.
    Display 'Backlog=' backlog.
    Move 'Listen call failed' to ezaerror-text.
    If retcode < 0 move 24 to failure.
    Perform Return-Code-Check thru Return-Code-Exit.
Listen-To-Socket-Exit.
    Exit.

*-----*
* Accept a connection request *
*-----*
Accept-Connection.
    Move socket-accept to ezaerror-function.
    Call 'EZASOCKET' using socket-accept socket-descriptor
        server-socket-address errno retcode.
    Move retcode to socket-descriptor-new.
    Display 'New socket=' retcode.
    Move 'Accept call failed' to ezaerror-text .
    If retcode < 0 move 24 to failure.
    Perform Return-Code-Check thru Return-Code-Exit.
Accept-Connection-Exit.
    Exit.

*-----*
* Use NTOP to display the IP address. *
*-----*
Numeric-To-Presentation.
    move socket-ntop to ezaerror-function.
    Call 'EZASOCKET' using socket-ntop ntop-family
        numeric-addr
        presentable-addr presentable-addr-len
        errno retcode.
    Display 'Presentable address = ' presentable-addr.
    Move 'NTOP call failed' to ezaerror-text.
    If retcode < 0 move 24 to failure.
    Perform Return-Code-Check thru Return-Code-Exit.
Numeric-T0-Presentation-Exit.
    Exit.

*-----*
* Read a message from the client. *
*-----*
Read-Message.
    move socket-read to ezaerror-function.
    move spaces to buf.
    display 'New socket descriptor = ' socket-descriptor-new.
    Call 'EZASOCKET' using socket-read socket-descriptor-new
        nbyte buf
        errno retcode.
    display 'Message received = ' buf.
    Move 'Read call failed' to ezaerror-text.
    If retcode < 0 move 24 to failure.
    Perform Return-Code-Check thru Return-Code-Exit.
Read-Message-Exit.
    Exit.

*-----*
* Write a message to the client. *
*-----*
Write-Message.
    move socket-write to ezaerror-function.
    move 'Message from EZAS06SC' to buf.
    Call 'EZASOCKET' using socket-write socket-descriptor-new
        nbyte buf
        errno retcode.
    Move 'Write call failed' to ezaerror-text
    If retcode < 0 move 24 to failure.
    Perform Return-Code-Check thru Return-Code-Exit.
Write-Message-Exit.
    Exit.

*-----*
* Close connected socket *
*-----*
Close-Socket.
    move socket-close to ezaerror-function
    Call 'EZASOCKET' using socket-close socket-descriptor-new

```

```

                                errno retcode.
Accept cur-time from time.
Display cur-time ' EZAS06CS : CLOSE RETCODE=' RETCODE
' ERRNO= ' ERRNO.
If retcode < 0 move 24 to failure
move 'Close call Failed' to ezaerror-text
perform write-ezaerror-msg thru write-ezaerror-msg-exit.
Close-Socket-Exit.
Exit.

*-----*
* Terminate socket API                                     *
*-----*

exit-term-api.
Call 'EZASOKET' using soket-termapi.

*-----*
* Terminate program                                       *
*-----*

exit-now.
move failure to return-code.
Goback.

*-----*
* Subroutine                                             *
* -----                                             *
* Write out an error message                             *
*-----*

write-ezaerror-msg.
move errno to ezaerror-errno.
move retcode to ezaerror-retcode.
display ezaerror-msg.
write-ezaerror-msg-exit.
exit.

*-----*
* Check Return Code after each Socket Call               *
*-----*
Return-Code-Check.
Accept Cur-Time from TIME.
move errno to ezaerror-errno.
move retcode to ezaerror-retcode.
Display Cur-Time ' EZAS06CS: ' ezaerror-function
' RETCODE= ' ezaerror-retcode
' ERRNO= ' ezaerror-errno.

IF RETCODE < 0
Perform Write-ezaerror-msg thru write-ezaerror-msg-exit
Move zeros to errno retcode
IF Opened-Socket Go to Close-Socket
ELSE IF Opened-API Go to exit-term-api
ELSE Go to exit-now.
Move zeros to errno retcode.
Return-Code-Exit.
Exit.

```

Figure 137. EZAS06CS COBOL call interface sample IPv6 server program

COBOL call interface sample IPv6 client program

The EZAS06CC program is a client module that shows you how to use the following calls provided by the call socket interface:

- CLOSE
- CONNECT
- GETCLIENTID
- GETNAMEINFO
- INITAPI
- NTOP
- PTON

- READ
- SHUTDOWN
- SOCKET
- TERMAPI
- WRITE

```

*****
*
*   MODULE NAME:  EZAS06CC - THIS IS A VERY SIMPLE IPV6 CLIENT
*
*   Copyright:    Licensed Materials - Property of IBM
*
*                 "Restricted Materials of IBM"
*
*                 5694-A01
*
*                 Copyright IBM Corp. 2002, 2008
*
*                 US Government Users Restricted Rights -
*                 Use, duplication or disclosure restricted by
*                 GSA ADP Schedule Contract with IBM Corp.
*
*
*   Note:         COBOL variable names can contain a maximum of
*                 30 characters.
*
*
*   Status:       CSV1R10
*
*   LANGUAGE:     COBOL
*
*****

Identification Division.
*****

Program-id. EZAS06CC.

*****
Environment Division.
*****

*****
Data Division.
*****

Working-storage Section.
*-----*
* Socket interface function codes
*-----*
01  socket-functions.
    02 socket-accept          pic x(16) value 'ACCEPT'          '.
    02 socket-bind           pic x(16) value 'BIND'             '.
    02 socket-close          pic x(16) value 'CLOSE'            '.
    02 socket-connect        pic x(16) value 'CONNECT'          '.
    02 socket-fcntl          pic x(16) value 'FCNTL'            '.
    02 socket-freeaddrinfo   pic x(16) value 'FREEADDRINFO'     '.
    02 socket-getaddrinfo    pic x(16) value 'GETADDRINFO'      '.
    02 socket-getclientid    pic x(16) value 'GETCLIENTID'     '.
    02 socket-gethostbyaddr  pic x(16) value 'GETHOSTBYADDR'    '.
    02 socket-gethostbyname  pic x(16) value 'GETHOSTBYNAME'    '.
    02 socket-gethostid      pic x(16) value 'GETHOSTID'        '.
    02 socket-gethostname    pic x(16) value 'GETHOSTNAME'      '.
    02 socket-getnameinfo    pic x(16) value 'GETNAMEINFO'      '.
    02 socket-getpeername    pic x(16) value 'GETPEERNAME'      '.
    02 socket-getsockname    pic x(16) value 'GETSOCKNAME'      '.
    02 socket-getsockopt     pic x(16) value 'GETSOCKOPT'       '.
    02 socket-givesocket     pic x(16) value 'GIVESOCKET'       '.
    02 socket-initapi        pic x(16) value 'INITAPI'          '.
    02 socket-ioctl          pic x(16) value 'IOCTL'            '.
    02 socket-listen         pic x(16) value 'LISTEN'           '.
    02 socket-ntop           pic x(16) value 'NTOP'             '.
    02 socket-pton           pic x(16) value 'PTON'             '.
    02 socket-read           pic x(16) value 'READ'             '.
    02 socket-recv           pic x(16) value 'RECV'             '.
```

```

02 soket-recvfrom      pic x(16) value 'RCVFROM'      '.
02 soket-select        pic x(16) value 'SELECT'       '.
02 soket-send          pic x(16) value 'SEND'         '.
02 soket-sendto        pic x(16) value 'SENDTO'       '.
02 soket-setsockopt    pic x(16) value 'SETSOCKOPT'   '.
02 soket-shutdown      pic x(16) value 'SHUTDOWN'     '.
02 soket-socket        pic x(16) value 'SOCKET'       '.
02 soket-takesocket    pic x(16) value 'TAKESOCKET'   '.
02 soket-termap        pic x(16) value 'TERMAPI'      '.
02 soket-write         pic x(16) value 'WRITE'       '.
*-----*
* Work variables
*-----*
01 errno               pic 9(8) binary value zero.
01 retcode             pic s9(8) binary value zero.
01 index-counter       pic 9(8) binary value zero.
01 buffer-element.
05 buffer-element-nbr  pic 9(5).
05 filler              pic x(3) value space.
01 server-ipaddr-dotted pic x(15) value space.
01 client-ipaddr-dotted pic x(15) value space.
01 close-server        pic 9(8) Binary value zero.
88 close-server-down   value 1.
01 Connect-Flag        pic x value space.
88 CONNECTED           value 'Y'.
01 Client-Server-Flag  pic x value space.
88 CLIENTS             value 'C'.
88 SERVERS             value 'S'.
01 Terminate-Options   pic x value space.
88 Opened-API          value 'A'.
88 Opened-Socket       value 'S'.
01 timer-accum         pic 9(8) Binary value zero.
01 timer-interval      pic 9(8) Binary value 2000.
01 Cur-time.
02 Hour               pic 9(2).
02 Minute             pic 9(2).
02 Second             pic 9(2).
02 Hund-Sec          pic 9(2).
77 Failure            Pic S9(8) comp.
*-----*
* Variables used for the INITAPI call
*-----*
01 maxsoc-fwd          pic 9(8) Binary.
01 maxsoc-rdf redefines maxsoc-fwd.
02 filler              pic x(2).
02 maxsoc              pic 9(4) Binary.
01 initapi-ident.
05 tcpname            pic x(8) Value 'TCPCS ' .
05 asname             pic x(8) Value space.
01 subtask             pic x(8) value 'EZS06CC'.
01 maxsno              pic 9(8) Binary Value 1.
*-----*
* Variables used by the SHUTDOWN Call
*-----*
01 how                 pic 9(8) Binary.
*-----*
* Variables returned by the GETCLIENTID Call
*-----*
01 clientid.
05 clientid-domain    pic 9(8) Binary value 19.
05 clientid-name       pic x(8) value space.
05 clientid-task       pic x(8) value space.
05 filler              pic x(20) value low-value.
*-----*
* Variables returned by the GETNAMEINFO Call
*-----*
01 name-len            pic 9(8) binary.
01 host-name           pic x(255).
01 host-name-len       pic 9(8) binary.
01 service-name        pic x(32).
01 service-name-len    pic 9(8) binary.
01 name-info-flags     pic 9(8) binary value 0.
01 ni-nofqdn           pic 9(8) binary value 1.
01 ni-numerichost      pic 9(8) binary value 2.
01 ni-namereqd         pic 9(8) binary value 4.
01 ni-numericserver    pic 9(8) binary value 8.
01 ni-dgram            pic 9(8) binary value 16.
*-----*
* Variables used for the SOCKET call
*-----*
01 AF-INET             pic 9(8) Binary Value 2.
01 AF-INET6            pic 9(8) Binary Value 19.

```

```

01 SOCK-STREAM                pic 9(8) Binary Value 1.
01 SOCK-DATAGRAM              pic 9(8) Binary Value 2.
01 SOCK-RAW                   pic 9(8) Binary Value 3.
01 IPPROTO-IP                 pic 9(8) Binary Value zero.
01 IPPROTO-TCP                pic 9(8) Binary Value 6.
01 IPPROTO-UDP                pic 9(8) Binary Value 17.
01 IPPROTO-IPV6               pic 9(8) Binary Value 41.
01 socket-descriptor          pic 9(4) Binary Value zero.
*-----*
* Server socket address structure                                     *
*-----*
01 server-socket-address.
    05 server-afinet           pic 9(4) Binary Value 19.
    05 server-port             pic 9(4) Binary Value 1031.
    05 server-flowinfo         pic 9(8) Binary Value zero.
    05 server-ipaddr.
        10 filler              pic 9(16) Binary Value 0.
        10 filler              pic 9(16) Binary Value 0.
    05 server-scopeid          pic 9(8) Binary Value zero.
01 NBYTE                      PIC 9(8) COMP value 80.
01 BUF                        PIC X(80).
*-----*
* Variables used by the BIND Call                                   *
*-----*
01 client-socket-address.
    05 client-family           pic 9(4) Binary Value 19.
    05 client-port             pic 9(4) Binary Value 1032.
    05 client-flowinfo         pic 9(8) Binary Value 0.
    05 client-ipaddr.
        10 filler              pic 9(16) Binary Value 0.
        10 filler              pic 9(16) Binary Value 0.
    05 client-scopeid          pic 9(8) Binary Value 0.
*-----*
* Buffer and length fields for send operation                       *
*-----*
01 send-request-length         pic 9(8) Binary value zero.
01 send-buffer.
    05 send-buffer-total       pic x(4000) value space.
    05 closedown-message redefines send-buffer-total.
        10 closedown-id        pic x(8).
        10 filler              pic x(3992).
    05 send-buffer-seq redefines send-buffer-total
        pic x(8) occurs 500 times.
*-----*
* Variables used for the NTOP/PTON call                             *
*-----*
01 IN6ADDR-ANY                 pic x(45)
                                value '::'.
01 IN6ADDR-LOOPBACK            pic x(45)
                                value '::1'.
01 presentable-addr            pic x(45) value spaces.
01 presentable-addr-len        pic 9(4) Binary value 45.
01 numeric-addr.
    05 filler                  pic 9(16) Binary Value 0.
    05 filler                  pic 9(16) Binary Value 0.
*-----*
* Buffer and length fields for recv operation                       *
*-----*
01 read-request-length         pic 9(8) Binary value zero.
01 read-buffer                 pic x(4000) value space.
*-----*
* Other fields for send and recvfrom operation                     *
*-----*
01 send-flag                   pic 9(8) Binary value zero.
01 recv-flag                   pic 9(8) Binary value zero.
*-----*
* Error message for socket interface errors                         *
*-----*
01 ezaerror-msg.
    05 filler                  pic x(9) Value 'Function='.
    05 ezaerror-function       pic x(16) Value space.
    05 filler                  pic x value ' '.
    05 filler                  pic x(8) Value 'Retcode='.
    05 ezaerror-retcode        pic ---99.
    05 filler                  pic x value ' '.
    05 filler                  pic x(9) Value 'Errorno='.
    05 ezaerror-errno          pic zzz99.
    05 filler                  pic x value ' '.
    05 ezaerror-text           pic x(50) value ' '.

Linkage Section.
*=====

```

```

*=====*
Procedure Division.
*=====*

*~::~::~::~::~::~::~::~::~::~::~::~::~::~::~::~*
*          P R O C E D U R E    C O N T R O L S          *
*~::~::~::~::~::~::~::~::~::~::~::~::~::~::~::~*

        Perform Initialize-API      thru    Initialize-API-Exit.
        Perform Get-Client-ID      thru    Get-Client-ID-Exit.
        Perform Sockets-Descriptor thru    Sockets-Descriptor-Exit.
        Perform Presentation-To-Numeric thru
                                Presentation-To-Numeric-Exit.
        Perform CONNECT-Socket     thru    CONNECT-Socket-Exit.
        Perform Numeric-TO-Presentation thru
                                Numeric-To-Presentation-Exit.
        Perform Get-Name-Information thru
                                Get-Name-Information-Exit.
        Perform Write-Message      thru    Write-Message-Exit.
        Perform Shutdown-Send      thru    Shutdown-Send-Exit.
        Perform Read-Message       thru    Read-Message-Exit.
        Perform Shutdown-Receive   thru    Shutdown-Receive-Exit.
        Perform Close-Socket       thru    Exit-Now.

*-----*
* Initialize socket API                                     *
*-----*
Initialize-API.
    Move soket-initapi to ezaerror-function.
    Call 'EZASOKET' using soket-initapi maxsoc initapi-ident
                                subtask maxsno errno retcode.
    Move 'Initapi failed' to ezaerror-text.
    If retcode < 0 move 12 to failure.
    Perform Return-Code-Check thru Return-Code-Exit.
    Move 'A' to Terminate-Options.
Initialize-API-Exit.
Exit.

*-----*
* Let us see the client-id                                 *
*-----*
Get-Client-ID.
    Move soket-getclientid to ezaerror-function.
    Call 'EZASOKET' using soket-getclientid clientid errno
                                retcode.
    Display 'Our client ID = ' clientid-name ' ' clientid-task.
    Move 'Getclientid failed' to ezaerror-text.
    If retcode < 0 move 24 to failure.
    Perform Return-Code-Check thru Return-Code-Exit.
    Move 'C' to client-server-flag.
Get-Client-ID-Exit.
Exit.

*-----*
* Get us a stream socket descriptor                         *
*-----*
Sockets-Descriptor.
    Move soket-socket to ezaerror-function.
    Call 'EZASOKET' using soket-socket AF-INET6 SOCK-STREAM
                                IPPROTO-IP errno retcode.
    Move 'Socket call failed' to ezaerror-text.
    If retcode < 0 move 60 to failure.
    Perform Return-Code-Check thru Return-Code-Exit.
    Move 'S' to Terminate-Options.
    Move retcode to socket-descriptor.
Sockets-Descriptor-Exit.
Exit.

*-----*
* Use PTON to create an IP address to bind to.            *
*-----*
Presentation-To-Numeric.
    move soket-pton to ezaerror-function.
    move IN6ADDR-LOOPBACK to presentable-addr.
    Call 'EZASOKET' using soket-pton AF-INET6
                                presentable-addr presentable-addr-len
                                numeric-addr
                                errno retcode.
    Move 'PTON call failed' to ezaerror-text.
    If retcode < 0 move 24 to failure.
    Perform Return-Code-Check thru Return-Code-Exit.

```



```

        move numeric-addr to server-ipaddr.
Presentation-To-Numeric-Exit.
Exit.

*-----*
* CONNECT                                     *
*-----*
Connect-Socket.
    Move space to Connect-Flag.
    Move zeros to errno retcode.
    move soket-connect to ezaerror-function.
    CALL 'EZASOKET' USING SOKET-CONNECT socket-descriptor
                        server-socket-address errno retcode.
    Move 'Connection call failed' to ezaerror-text.
    If retcode < 0 move 24 to failure.
    Perform Return-Code-Check thru Return-Code-Exit.
    If retcode = 0 Move 'Y' to Connect-Flag.
Connect-Socket-Exit.
Exit.

*-----*
* Use NTOP to display the IP address.         *
*-----*
Numeric-To-Presentation.
    move soket-ntop to ezaerror-function.
    move server-ipaddr to numeric-addr.
    move soket-ntop to ezaerror-function.
    Call 'EZASOKET' using soket-ntop AF_INET6
        numeric-addr
        presentable-addr presentable-addr-len
        errno retcode.
    Display 'Presentable address = ' presentable-addr.
    Move 'NTOp call failed' to ezaerror-text.
    If retcode < 0 move 24 to failure.
    Perform Return-Code-Check thru Return-Code-Exit.
Numeric-To-Presentation-Exit.
Exit.

*-----*
* Use GETNAMEINFO to get the host and service names *
*-----*
Get-Name-Information.
    move 28 to name-len.
    move 255 to host-name-len.
    move 32 to service-name-len.
    move ni-namereqd to name-info-flags.
    move soket-getnameinfo to ezaerror-function.
    Call 'EZASOKET' using soket-getnameinfo
        server-socket-address name-len
        host-name host-name-len
        service-name service-name-len
        name-info-flags
        errno retcode.
    Display 'Host name = ' host-name.
    Display 'Service = ' service-name.
    Move 'Getaddrinfo call failed' to ezaerror-text.
    If retcode < 0 move 24 to failure.
    Perform Return-Code-Check thru Return-Code-Exit.
Get-Name-Information-Exit.
Exit.

*-----*
* Write a message to the server               *
*-----*
Write-Message.
    Move soket-write to ezaerror-function.
    Move 'Message from EZAS06CC' to buf.
    Call 'EZASOKET' using soket-write socket-descriptor
        nbyte buf
        errno retcode.
    Move 'Write call failed' to ezaerror-text.
    If retcode < 0 move 84 to failure.
    Perform Return-Code-Check thru Return-Code-Exit.
Write-Message-Exit.
Exit.

*-----*
* Shutdown to pipe                           *
*-----*
Shutdown-Send.
    Move soket-shutdown to ezaerror-function.
    move 1 to how.

```

```

        Call 'EZASOKET' using soket-shutdown socket-descriptor
            how
            errno retcode.
        Move 'Shutdown call failed' to ezaerror-text.
        If retcode < 0 move 99 to failure.
        Perform Return-Code-Check thru Return-Code-Exit.
    Shutdown-Send-Exit.
    Exit.

*-----*
* Read a message from the server.                                     *
*-----*
Read-Message.
    Move soket-read to ezaerror-function.
    Move spaces to buf.
    Call 'EZASOKET' using soket-read socket-descriptor
        nbyte buf
        errno retcode.
    If retcode < 0
        Move 'Read call failed' to ezaerror-text
        move 120 to failure
        Perform Return-Code-Check thru Return-Code-Exit.
Read-Message-Exit.
Exit.

*-----*
* Shutdown receive pipe                                             *
*-----*
Shutdown-Receive.
    Move soket-shutdown to ezaerror-function.
    move 0 to how.
    Call 'EZASOKET' using soket-shutdown socket-descriptor
        how
        errno retcode.
    Move 'Shutdown call failed' to ezaerror-text.
    If retcode < 0 move 99 to failure.
    Perform Return-Code-Check thru Return-Code-Exit.
Shutdown-Receive-Exit.
Exit.

*-----*
* Close socket                                                     *
*-----*
Close-Socket.
    Move soket-close to ezaerror-function.
    Call 'EZASOKET' using soket-close socket-descriptor
        errno retcode.
    Move 'Close call failed' to ezaerror-text.
    If retcode < 0 move 132 to failure
    perform write-ezaerror-msg thru
        write-ezaerror-msg-exit.
    Accept Cur-Time from TIME.
    Display Cur-Time ' EZAS06CC: ' ezaerror-function
        ' RETCODE=' RETCODE ' ERRNO=' ERRNO.
Close-Socket-Exit.
Exit.

*-----*
* Terminate socket API                                             *
*-----*
exit-term-api.
    ACCEPT cur-time from TIME.
    Display cur-time ' EZAS06CC: TERMAPI '
        ' RETCODE=' RETCODE ' ERRNO=' ERRNO.
    Call 'EZASOKET' using soket-termapi.

*-----*
* Terminate program                                               *
*-----*
exit-now.
    Move failure to return-code.
    Goback.

*-----*
* Subroutine.                                                     *
*-----*
* Write out an error message                                       *
*-----*
write-ezaerror-msg.
    Move errno to ezaerror-errno.
    Move retcode to ezaerror-retcode.
    Display ezaerror-msg.

```

```

write-ezaerror-msg-exit.
Exit.

*-----*
* Check Return Code after each Socket Call      *
*-----*
Return-Code-Check.
  Accept Cur-Time from TIME.
  Display Cur-Time ' EZAS06CC: ' ezaerror-function
                  ' RETCODE=' RETCODE ' ERRNO= ' ERRNO.
  IF RETCODE < 0
    Perform Write-ezaerror-msg thru write-ezaerror-msg-exit
    Move zeros to errno retcode
    IF Opened-Socket Go to Close-Socket
    ELSE IF Opened-API Go to exit-term-api
      ELSE Go to exit-now.
  Move zeros to errno retcode.
Return-Code-Exit.
Exit.

```

Figure 138. EZAS06CC COBOL call interface sample IPv6 client program

Chapter 14. REXX socket application programming interface

The z/OS Communications Server socket API for REXX supports IPv4 and IPv6 socket call instructions.

The REXX socket API uses the REXX built-in function RXSOCKET to access the z/OS Communications Server socket interface. This API provides similar socket functions to those in other high-level languages. When possible, the REXX socket functions match the C socket functions; an example of the corresponding LE C/C++ socket function is included when applicable.

Overview of the REXX socket API

This section contains introductory material about REXX socket APIs.

Supported REXX APIs

z/OS Communications Server supports the REXX Sockets API and the REXX FTP API.

The REXX FTP API is documented in [z/OS Communications Server: IP Programmer's Guide and Reference](#). The REXX Sockets API is documented in this deliverable. Unless noted in [z/OS Communications Server: New Function Summary](#), this REXX socket API is upward compatible. Application programs that use new functions will not be downward compatible.

Rule: Unless indicated otherwise, all socket commands that are listed are enabled for IPv4 and IPv6.

Tip: This API is compatible with compiled REXX.

Prerequisites for using REXX sockets

To use REXX sockets, the EZBRXSOC load module must be included.

The EZBRXSOC load module is defined with two load module aliases: RXSOCKET and SOCKET. When a program invokes the REXX socket function, either the load module or an alias is called. The load module and its aliases reside in the SEZALOAD library. The examples in this documentation use the SOCKET alias.

Requirement: The EZBRXSOC load module and its aliases must be included in one of the following items:

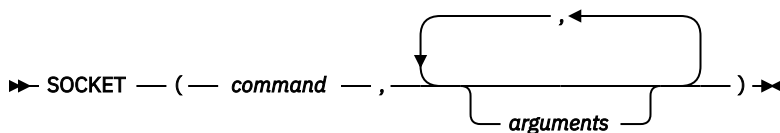
- The system LNKLIST
- The STEPLIB DD concatenation of the job that is running the program that uses REXX sockets

Format of the REXX socket function and return values

You can issue socket commands in REXX by calling the built-in socket function.

Format

This function uses the following format, which is similar to that used to invoke C sockets:



Parameters

command

The socket API command to be issued, for example, SEND.

arguments

One or more parameters separated by a comma. All parameters are passed as space-delimited strings.

Returned value

The socket function returns a space-delimited string. If the REXX socket library can issue the socket command, the return value consists of the REXX TCP/IP error number value, the return code, and any additional socket information. If the REXX socket library cannot process the socket command, the return value consists of a REXX socket library error value and information about what caused the error.

Tip: For an error condition, the REXX socket library returns both the numeric and text versions of the error, for example, 2009 ESOCKETNOTDEFINED Socket not defined.

See [Appendix B, “Socket call error return codes,” on page 743](#) for additional information about the numeric error codes that are returned by this command.

Example

Consider the following code sample:

```
src = socket("ACCEPT",l_socketid);
```

where:

- "ACCEPT" is the socket command.
- *l_socketid* is an argument that is required by the ACCEPT command. In this example, it specifies the socket descriptor of the listening socket.

When a new connection is available, the ACCEPT command returns the following string:

```
src = 0 45
```

where 0 is the return code, and 45 is the file descriptor of the new connection.

REXX programming hints and tips

This topic contains information that you might find helpful when you use REXX sockets.

Capitalization

Throughout the documentation, REXX socket API commands and constants are capitalized when they are used in descriptive text. For example, the LISTEN command places a socket descriptor in passive mode.

Quotation marks

Throughout the documentation, REXX socket API commands and constants are enclosed in quotation marks (") when they are used in code examples, for example, `src = socket("ACCEPT",sockfd);`.

Although the use of quotation marks is optional, consider using quotation marks to prevent programming errors. Using quotation marks forces the socket function to use string literals rather than REXX variables. When REXX encounters an uninitialized variable, it initializes that variable with the name of the variable. The command `socket(ACCEPT,sockid)` is valid because the uninitialized variable ACCEPT is initialized to the character string ACCEPT. However, if the program initializes the ACCEPT variable with a value other than the character string ACCEPT, the socket function fails.

Guideline: Unless otherwise indicated by a specific socket command, all socket commands and constant values must be passed as character strings.

Splitting a function over multiple lines

The socket function, with parameters, might exceed 80 characters. In such cases, you can split the function across multiple lines by using the REXX continuation character, a comma (,). Consider the following examples:

Continuing a string across two lines

```
l_string1 = "This is a string split between",  
           "two lines.";
```

Continuing a string when the continuation occurs on a parameter boundary

```
src = socket("GETADDRINFO","CHILE",,  
            23,"AI_CANONNAMEOK");
```

In this example, the first comma after the parameter "CHILE" indicates the end of the parameter. The second comma indicates that the REXX statement continues on the next line.

Return codes

To avoid problems, an application should check the return code of a socket command after each socket call. The examples in this topic do not always follow this recommendation. The examples are intended to show how to issue the socket command.

Guideline: Use either the REXX PARSE feature or the WORD function to access the return code. For example,

```
parse var l_retcode src remainder;  
if src = 0 then do  
    /* DO SOME STUFF */  
end;  
else do  
    /* Process the error */  
end;
```

Allocating and deallocating socket sets

To use the socket commands provided by the REXX socket function, a socket set must be active. To allocate a socket set, use the INITIALIZE socket command. The INITIALIZE command creates a socket set and can support multiple socket calls. The *subtaskid* value identifies the socket set and usually corresponds to the application name. The *service* value indicates the TCP/IP stack name to form an affinity with.

Restriction: When the INITIALIZE command is issued, the REXX socket API forms an affinity with the default TCP/IP stack. In an INET environment, the affinity is created with the active stack. In a CINET environment, the affinity is created with the default stack. The default stack is determined either by the first BPXPRMxx SUBFILESYSTYPE statement or by the SUBFILESYSTYPE statement with the keyword DEFAULT. For additional information, see *z/OS Communication Server: MVS Initialization and Tuning Reference*.

Guideline: Before you exit the program or when you do not need the socket environment, use the TERMINATE socket command to deallocate the socket set.

Blocking and nonblocking mode

A socket can be in blocking or nonblocking mode. In blocking mode, commands such as SEND and RECV block the caller until either the operation is completed successfully or an error occurs. In nonblocking mode, the caller is not blocked, but the operation ends immediately with either the 35 EWOULDBLOCK or 36 EINPROGRESS return code. Use the FCNTL or IOCTL commands to switch the socket between blocking and nonblocking modes.

When a socket is in nonblocking mode, you can use the SELECT command to monitor the socket for one or more socket events. The socket can be monitored for events that indicate that the socket is ready for writing or reading, or whether an exception has occurred.

Transferring a socket to a subtask

If the application uses the GIVESOCKET and TAKESOCKET commands to transfer a socket from a parent program to a subtask, both the parent and subtask must agree on a mechanism for exchanging the client ID and the socket descriptor. The parent program can use the SELECT command to monitor when the subtask takes the socket. After the subtask takes the socket, the parent then can close the socket that was given.

SO_ASCII and SO_EBCDIC socket options

The socket options SO_ASCII and SO_EBCDIC identify the socket data type for use by the REXX RXSOCKET program. Setting the SO_EBCDIC option to ON has no effect, and setting the SO_ASCII option to ON causes all incoming data on the socket to be translated from ASCII to EBCDIC and all outgoing data on the socket to be translated from EBCDIC to ASCII.

How structures are represented

Instead of using binary-based data structures, the REXX socket library represents all data structures as strings of space-delimited values, where each value represents a field in the data structure.

The REXX language is a type-independent language. All data is manipulated and represented using a character format. When an application passes data into the REXX socket library, the data must be a space-delimited string. When the REXX socket library returns socket information, it returns the information as a space-delimited string on the return value.

Rules:

- All IPv4 addresses are represented as strings in dotted decimal format.
- All IPv6 addresses are represented as strings in IPv6 colon hexadecimal format.
- When a NAME string is returned as a result, the IP address is in IPv4 dotted decimal or IPv6 colon hexadecimal format.

Tip: When you specify a NAME string as an input parameter to a command, you can specify the *ipaddress* field either as an IP address or as a host name to be resolved by a name server. For example, you can code NAME="AF_INET 1049 MYHOST", where AF_INET is the address family, 1049 is the port number, and MYHOST is the host name to be resolved to an IP address.

struct sockaddr_in

This structure represents an IPv4 socket address. In the REXX socket library, this structure is represented by the NAME string. The NAME string has the following format:

```
NAME = "domain portid ipaddress"
```

where:

domain

The number 2 or AF_INET

portid

The local or remote port to which the socket is to be bound or connected

ipaddress

The IPv4 address of the local or remote host to which the socket is to be bound or connected

The following string is an example of an IPv4 NAME string:

```
NAME = "AF_INET 24 10.11.103.1";
```


struct sockaddr_in6

This structure represents an IPv6 socket address. In the REXX socket library, this structure is represented by the NAME string. The NAME string has the following format:

```
NAME = "domain portid flowinfo ipaddress scopeid"
```

where:

domain

The number 19 or AF_INET6.

portid

The local or remote port to which the socket is to be bound or connected.

flowinfo

This value must be set to 0.

ipaddress

The IPv6 address of the local or remote host to which the socket is to be bound or connected.

scopeid

Identifies the interfaces that are applicable for the scope of the address that is specified in the *ipaddress* field. For a link-local IP address, the *scopeid* field can specify a link index, which identifies a set of interfaces. For all other scopes, the *scopeid* field must be set to 0. Setting the *scopeid* field to 0 indicates that any address type and scope can be specified.

The following string is an example of an IPv6 NAME string:

```
NAME = "AF_INET6 24 0 2001:10:11:103::1 0"
```

struct ip_mreq

This structure represents the mapping between an IPv4 multicast address and an IPv4 interface. In the REXX socket library, this structure is represented by the *ipmreq* string. The *ipmreq* string has the following format:

```
ipmreq = "maddress iaddress"
```

where:

maddress

An IPv4 multicast address

iaddress

The IPv4 interface address

The following string is an example of an IPv4 *ipmreq* string:

```
ipmreq = "224.224.224.1 10.123.21.3"
```

struct ipv6_mreq

This structure represents the mapping between an IPv6 multicast address and an interface index. In the REXX socket library, this structure is represented by the *ipmreq* string. The *ipmreq* string has the following format:

```
ipmreq = "maddress index"
```

where:

maddress

An IPv6 multicast address

index

An interface index number

The following string is an example of an IPv6 *ipmreq* string:

```
ipmreq = "FF05::101 34"
```

struct ip_mreq_source

This structure represents a multicast source filter. It is used with the IOCTL command to filter the multicast packets that an application wants to receive; it also defines the remote host from which the packets are sent. In the REXX socket library, this structure is represented by the *ip_mreq_source* string. The *ip_mreq_source* string has the following format:

```
ipmreqsource = "maddress saddress iaddress"
```

where:

maddress

An IPv4 multicast address

saddress

An IPv4 source address of a remote host

iaddress

An IPv4 interface address

The following string is an example of an *ip_mreq_source* string:

```
ipmreqsource = "224.224.224.2 10.1.2.3 10.123.21.3"
```

Restriction: This structure supports IPv4 addresses only.

struct group_req

This structure represents a protocol-independent mapping between a multicast IP address and an interface index. It is used with the IOCTL command to join and leave multicast groups. In the REXX socket library, this structure is represented by the *groupreq* string. The *groupreq* string has the following format:

```
groupreq = "index NAME"
```

where:

index

An interface index

NAME

The NAME string of a multicast socket address

The following strings are examples of *groupreq* strings:

```
groupreq = "15664 AF_INET 5000 224.224.224.2"  
groupreq = "15667 AF_INET6 5000 0 FF05::101 0"
```

struct group_req_source

This structure represents a protocol-independent mapping between a multicast IP address, an interface index, and a source address for a remote host. It is used with the IOCTL command to filter the multicast packets that an application wants to receive; it also defines the remote host from which the packets are sent. In the REXX socket library, this structure is represented by the *groupreqsource* string. The *groupreqsource* string has the following format:

```
groupreqsource = "index M_NAME S_NAME"
```

where:

index

An interface index

M_NAME

The NAME string of a multicast socket address

S_NAME

The NAME string of a source socket address

The following strings are examples of *groupreqsource* strings:

```
groupreqsource = "15664 AF_INET 5000 224.224.224.2 AF_INET 5000 10.1.2.3"
groupreqsource = "15667 AF_INET6 5000 0 FF05::101 0 AF_INET6 5000 0
                2001:10:1:2::3 0"
```

REXX runtime functions

This section describes the runtime functions that are supported by the REXX socket function. When applicable, the LE C/C++ Equivalent call is also shown.

Commands to process socket sets	
“INITIALIZE” on page 633	Use the INITIALIZE command to preallocate a socket set.
“SOCKETSET” on page 672	Use the SOCKETSET command to retrieve the name of the active socket set. If you specify the name of a socket set as a parameter, then that socket set becomes the active socket set.
“SOCKETSETLIST” on page 673	Use the SOCKETSETLIST command to list the names of all available socket sets that are currently defined by the application.
“SOCKETSETSTATUS” on page 674	Use the SOCKETSETSTATUS command to list information about a socket set.
“TERMINATE” on page 677	Use the TERMINATE command to close all sockets in the specified socket set and to release the socket set.
Commands to open, close, and manipulate sockets	
“ACCEPT” on page 586	Use the ACCEPT command to accept new connections from a client.
“BIND” on page 588	Use the BIND command to bind a local NAME string to a socket descriptor.
“BIND2ADDRSEL” on page 590	Use the BIND2ADDRSEL command to bind a socket to the local IP address that would be selected by the stack to communicate with the input destination IP address.
“CLOSE” on page 592	Use the CLOSE command to close a socket and release the resources that are associated with the socket descriptor.
“CONNECT” on page 593	A client application uses the CONNECT command to establish a connection between a local socket and a remote socket.
“GIVESOCKET” on page 629	Use the GIVESOCKET command to transfer a socket descriptor to another application that is running on the same host.

Commands to process socket sets	
“LISTEN” on page 644	Use the LISTEN command to determine whether a socket is ready to accept client connection requests.
“SHUTDOWN” on page 669	Use the SHUTDOWN command to shut down all or part of a duplex connection.
“SOCKET” on page 670	Use the SOCKET command to open a socket descriptor in the active socket set.
“TAKESOCKET” on page 675	Use the TAKESOCKET command to take a socket descriptor that is passed from another program using the GIVESOCKET command. A socket descriptor can be taken by an application only when the socket is in the same address family.
Commands to exchange data on sockets	
“READ” on page 645	Use the READ command to read data on the specified socket. The maximum amount of data to be read is specified by the <i>maxlength</i> parameter. If the socket is in blocking mode and data is not available on the socket, the command blocks until data arrives.
“RECV” on page 647	Use the RECV command to receive data on a specified socket. The RECV command can be issued only against connected sockets.
“RECVFROM” on page 649	Use the RECVFROM command to receive data on the specified socket.
“SEND” on page 655	Use the SEND command to send an outgoing message on the connected socket.
“SENDTO” on page 657	Use the SENDTO command to send an outgoing message on a socket descriptor. This command differs from the SEND command in that it includes the destination address as a parameter.
“WRITE” on page 678	Use the WRITE command to send an outgoing message on the connected socket. The WRITE command is similar to the SEND command, except that the WRITE command does not support the control flags that are available with the SEND command.
Commands to resolve host names and IP addresses	
“GETADDRINFO” on page 597	Use the GETADDRINFO command to resolve host or service name information.
“GETCLIENTID” on page 605	Use the GETCLIENTID command to retrieve the client ID for the calling application. The client ID is the identifier by which the calling application is known to the TCP/IP address space.
“GETDOMAINNAME” on page 607	Use the GETDOMAINNAME command to retrieve the name of the domain to which the current TCP/IP stack belongs.

Commands to process socket sets	
“GETHOSTBYADDR” on page 608	Use the GETHOSTBYADDR command to resolve an IPv4 address to a host name.
“GETHOSTBYNAME” on page 609	Use the GETHOSTBYNAME command to resolve a host name to an IPv4 address.
“GETHOSTID” on page 610	Use the GETHOSTID command to return the primary IPv4 address for the current host. The primary address is the IP address of the default home address that is specified in the TCP/IP configuration file for the stack with which the current socket set is associated.
“GETHOSTNAME” on page 611	Use the GETHOSTNAME command to return the name of the host on which the application is running
“GETNAMEINFO” on page 612	Use the GETNAMEINFO command to translate a socket address to a node name and service location.
“GETPEERNAME” on page 614	Use the GETPEERNAME command to return the name of the remote peer that is connected to the socket.
“GETPROTOBYNAME” on page 616	Use the GETPROTOBYNAME command to translate a network protocol name to a protocol number.
“GETPROTOBYNUMBER” on page 616	Use the GETPROTOBYNUMBER command to translate a network protocol number to a protocol name.
“GETSERVBYNAME” on page 617	Use the GETSERVBYNAME command to retrieve a service and port number.
“GETSERVBYPOR” on page 618	Use the GETSERVBYPOR command to translate a port number to the name of the service that is using the port.
“GETSOCKNAME” on page 619	Use the GETSOCKNAME command to retrieve the name of a bound socket.
“INET6ISSRCADDR” on page 631	Use the INET6ISSRCADDR command to indicate whether an input IPV6 socket address matches an address that is defined to the stack, which conforms to one or more input IPV6_ADDR_PREFERENCES flags.
“RESOLVE” on page 652	Use the RESOLVE command to resolve a host name or an IP address.
Commands to manage socket configuration, options, and modes	
“FCNTL” on page 596	Use the FCNTL command to control the operating characteristics of a socket.
“GETSOCKOPT” on page 621	Use the GETSOCKOPT command to retrieve the active socket options that were set by the SETSOCKOPT command.
“IOCTL” on page 634	Use the IOCTL command to perform control functions on sockets.

Commands to process socket sets	
“SELECT” on page 653	Use the SELECT command to monitor groups of sockets to determine when one or more of the sockets is ready for a read operation, is ready for a write operation, or has an exception pending.
“SETSOCKOPT” on page 659	Use the SETSOCKOPT command to set socket options.
“VERSION” on page 678	Use the VERSION command to retrieve the name, version number, and version date of the REXX socket library.

ACCEPT

Use the ACCEPT command to accept new connections from a client.

This command is valid only for stream sockets. Connection requests are processed in the order in which they are received. When a new connection is accepted, a new socket ID is created with the same properties as the listening socket ID. The new socket ID cannot be used to accept new connections. The original socket remains available to accept new connection requests.

This command supports both blocking and nonblocking sockets.

Rule: When the listening socket is in blocking mode, the ACCEPT command blocks the caller until a new connection request is received. When the listening socket is in nonblocking mode, the ACCEPT command immediately returns the 35 EWOULDBLOCK return code.

Tip: When you use blocking or nonblocking socket calls, use the SELECT command to check for new connection requests before you call the ACCEPT command. The availability of a new connection is reported as a READ event on the listening socket. The new socket should not be accepted until after the READ event is received.

Format

➤ SOCKET — (— "ACCEPT" — , — *socketid* —) ➤

Parameters

socketid

The socket descriptor of the listening socket where new connection requests are queued

Returned value

The command returns a string that contains the return code, the new *scopeid* value for the accepted connection, and the NAME string of the connecting client. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

For information about the format of the NAME string, see [“How structures are represented” on page 580](#).

Consider the following IPv4 and IPv6 examples:

IPv4

The string, 0 6 AF_INET 50000 10.1.2.3, is an example of what this command returns. 0 is the function return code, 6 is the new socket ID of the accepted connection, AF_INET is the address family to which the socket belongs, 50000 is the remote port from which the client is connecting, and 10.1.2.3 is the remote address from which the client is connecting.

IPv6

The string, 0 6 AF_INET6 5462 0 2001:10:11:103::1 0, is an example of what this command returns. 0 is the function return code, 6 is the new socket ID of the accepted connection, AF_INET6 is the address family to which the socket belongs, 5462 is the remote port from which the client is connecting, 0 is the *flowinfo* value, 2001:10:11:103::1 is the remote address from which the client is connecting, and 0 is the *scopeid* value. For IPv6 connections, the *flowinfo* and *scopeid* fields are set to 0.

See [Appendix B, “Socket call error return codes,”](#) on page 743 for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 1 EPERM
- 9 EBADF
- 22 EINVAL
- 35 EWOULDBLOCK

The following REXX socket API error numbers can be returned:

- 2001 EINVALIDRXSOCKETCALL
- 2006 ESOCKETNOTDEFINED
- 2007 EMAXSOCKETSREACHED
- 2009 ESOCKETNOTDEFINED
- 2012 EINVALIDNAME

LE C/C++ equivalent

```
int accept(int socket, struct sockaddr *address, int *address_len);
```

Code example

```
/* REXX EZARXR01 */
/*
 * This sample demonstrates the use of the INITIALIZE, SOCKET, BIND
 * LISTEN, ACCEPT, RECV, CLOSE, and TERMINATE socket commands.
 *
 * The program creates a listening socket and then goes into a
 * loop and blocks on the accept command. When a new connection is
 *
 * ACCEPTED the program will issue one receive command and then close
 * the connection. If the data received is the string "DONE", then the
 *
 * program will close the listening socket and terminate. Otherwise the
 * program will wait for the next connection.
 *
 * RESTRICTION: This program is designed to read 1 packet with a max
 * size of 512 bytes.
 *
 * GUIDELINE: It is generally recommended that a program loop around
 * the RECV command to ensure that all data is read off
 * the socket. This sample does not follow the guideline.
 */
src = socket("INITIALIZE","MYSET01",10);
if perror(src,"INITIALIZE") = 0 then do
  src = socket("SOCKET","AF_INET6","STREAM");
  if perror(src,"SOCKET") = 0 then do
    parse var src . l_sockid
    l_name6 = "AF_INET6 54004 0 ::0 0";
    src = socket("BIND", l_sockid, l_name6);
    if perror(src,"BIND") = 0 then do
      src = socket("LISTEN", l_sockid);
      if perror(src,"LISTEN") = 0 then do
        l_Done = "FALSE";
        do until l_Done = "TRUE";
          say "Listening on socket "l_sockid;
          src = socket("ACCEPT", l_sockid);
          if perror(src,"ACCEPT") = 0 then do
            parse var src . l_newsockid . ;
            src = socket("RECV",l_newsockid,512);
            parse var src l_retcode l_data len l_data
            if l_data = "DONE" | perror(src,"RECV") \= 0 then
              l_Done = "TRUE";
            src = socket("CLOSE",l_newsockid);
            src = perror(src,"Accepted socket close");
          end;
        else do
          src = socket("CLOSE",l_sockid);
          src = perror(src,"Listen close");
          l_Done = "TRUE";
        end; /* ACCEPT */
      end; /* DO */
    end; /* LISTEN */
  end; /* BIND */
end; /* SOCKET */
end; /* INITIALIZE */
src = socket("TERMINATE","MYSET01");
src = perror(src,"TERMINATE");
exit 0;
/* This routine returns -1 if the first word if arg 1 is not zero */
perror: if word(arg(1),1) = 0 then return 0; else
  Say arg(2) "Error : "arg(1);
  return -1;
```

Figure 139. ACCEPT command example

BIND

Use the BIND command to bind a local NAME string to a socket descriptor.

The format of the NAME string depends on the addressing family of the socket. An application can use the BIND command to specify the network interface from which the socket can receive TCP connection requests or UDP packets. A socket bound to a specific local-IP address receives only packets that are targeted to that address. Outgoing packets have as their source address the address that is used to bind

the socket. The BIND command supports both stream or datagram sockets, and it can be issued by both clients or servers.

Guidelines:

- Do not bind a socket to a specific interface address. Binding a socket to a specific interface address limits network access to the application and might result in unexpected outages. To enable TCP connections to be accepted and UDP datagrams to be received over any interface, specify INADDR_ANY or IN6ADDR_ANY in the *ipaddress* field of the NAME parameter.
- A server (an application that calls the LISTEN command) should always bind to the same well-known port. When the socket is bound and the LISTEN command is issued, the bound socket is marked as being passive. Passive sockets cannot be used to send or receive data. They are used to receive new connection requests from remote clients using the ACCEPT command.
- The 48 EADDRINUSE error message indicates that a previous application is using the port. This error also can be received when a listening server is restarted. The TCP/IP stack maintains state information from the previous instance of the server for a fixed time before it releases a port and address for reuse. To avoid this situation, use the SETSOCKOPT command to set the SO_REUSEADDR socket option on the listening socket.

Format

➤ SOCKET — (— "BIND" — , — *socketid* — , — *name* —) ➤

Parameters

socketid

The socket descriptor.

name

The socket address to which the socket is to be bound.

The format for the *name* parameter depends on the socket type:

AF_INET sockets (IPv4)

name = "domain portid ipaddress"

AF_INET6 sockets (IPv6)

name = "domain portid flowinfo ipaddress scopeid"

where:

- The *domain* value is the decimal number 2 for AF_INET and the decimal number 19 for AF_INET6.
- The *portid* value is 0 or the local port to which the socket will bind. When the *portid* field is set to 0, the stack selects the local port.
- The *ipaddress* value is the IP address to which the socket binds and the source address of outgoing packets. Other valid values are INADDR_ANY, IN6ADDR_ANY, INADDR_BROADCAST, BROADCAST, and LOOPBACK. When the *ipaddress* field is set to INADDR_ANY or IN6ADDR_ANY, the stack passes to the application any TCP connection requests or UDP datagrams that are received for the socket on any local interface. For outgoing packets, the stack selects the source address.

Tip: System administrators can override the INADDR_ANY or IN6ADDR_ANY value by specifying the BIND option on the PORT reservation statement in the TCPIP . PROFILE file. This is equivalent to coding the IP address on the *name* parameter. For more information, see [z/OS Communications Server: IP Configuration Reference](#).

- The *flowinfo* value must be 0.
- The *scopeid* value identifies the interfaces that are applicable for the scope of the address that is specified in the *ipaddress* field. For a link-local IP address, the *scopeid* field can specify a link index, which identifies a set of interfaces. For all other scopes, the *scopeid* field must be set to 0. Setting the *scopeid* field to 0 indicates that any address type and scope can be specified.

Returned value

The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

See [Appendix B, “Socket call error return codes,”](#) on page 743 for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 1 EPERM
- 9 EBAADF
- 22 EINVAL
- 47 EAFNOSUPPORT
- 48 EADDRINUSE
- 49 EADDRNOTAVAIL

The following REXX socket API error numbers can be returned:

- 2001 EINVALIDRXSOCKETCALL
- 2009 ESOCKETNOTDEFINED

LE C/C++ equivalent

```
int bind(int socket, struct sockaddr *address, int *address_len);
```

Returned value

See the [“ACCEPT” on page 586](#) command for an example of using the BIND command.

BIND2ADDRSEL

Use the BIND2ADDRSEL command to bind a socket to the local IP address that would be selected by the stack to communicate with the input destination IP address.

Use the BIND2ADDRSEL command when the application must verify that the local IP address assigned by the stack meets its address selection criteria as specified by the IPV6_ADDR_PREFERENCES socket option before the stack sends any packets to the remote host. In a TCP or UDP application, the BIND2ADDRSEL command usually follows the SETSOCKOPT command with option IPV6_ADDR_PREFERENCES and precedes any communication with a remote host.

Result: The stack attempts to select a local IP address according to your application preferences. However, a successful BIND2ADDRSEL command does not guarantee that all the selection preferences of your source IP address were met.

Guidelines:

- Use the SETSOCKOPT command to set the IPV6_ADDR_PREFERENCES option to indicate your selection preferences of source IP address before binding the socket and before allowing an implicit bind of the socket to occur.

Result: If a socket has not been explicitly bound to a local IP address with a BIND or BIND2ADDRSEL command when a CONNECT, SENDTO, or SENDMSG macro is issued, an implicit bind of the socket occurs. The stack chooses the local IP address used for outbound packets.

Requirement: When your application is using stream sockets, and must prevent the stack from sending any packets whatsoever (such as SYN) to the remote host before it can verify that the local IP address meets the values specified for the IPV6_ADDR_PREFERENCES option, do not allow the CONNECT command to implicitly bind the socket to a local IP address. Instead, bind the socket with the BIND2ADDRSEL command and test the local IP address assigned with the INET6_IS_SRCADDR command. If the assigned local IP address is satisfactory, you can then use the CONNECT command to establish communication with the remote host.

- After you successfully issue the BIND2ADDRSEL command, use the GETSOCKNAME command to obtain the local IP address that is bound to the socket. When the local IP address is obtained, use the INET6_IS_SRCADDR command to verify that the local IP address meets your address selection criteria.

Format

➤ SOCKET — (— "BIND2ADDRSEL" — , — *socketid* — , — *name* —) ➤

Parameters

socketid

The socket descriptor.

name

The socket address of the remote host machine that the application communicates with.

name = "*domain portid flowinfo ipaddress scopeid*"

where:

- The *domain* value must be the decimal number 19 or AF_INET6.
- The *portid* value is not used by the BIND2ADDRSEL command.

Requirement: You must specify a decimal numeral between 0 and 65535 for the *portid* value.

- The *flowinfo* value is not used by the BIND2ADDRSEL command.

Requirement: You must specify a decimal numeral between 0 and 2147483647 for the *flowinfo* value.

- The *ipaddress* value is the IPv6 address of the remote host machine the application will communicate with.

Rule: Specify an IPv4 address by using its IPV4 mapped IPV6 format.

- The *scopeid* value identifies the interfaces that are applicable for the scope of the address that is specified in the *ipaddress* field. For a link-local IP address, the *scopeid* field can specify a link index, which identifies a set of interfaces. For all other scopes, the *scopeid* field must be set to 0. Setting the *scopeid* field to 0 indicates that any address type and scope can be specified.

Returned value

The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

See [Appendix B, "Socket call error return codes," on page 743](#) for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 1 EPERM
- 9 EBAADF
- 22 EINVAL
- 41 EPROTOTYPE
- 45 EOPNOTSUPP
- 47 EAFNOSUPPORT
- 49 EADDRNOTAVAIL
- 65 EHOSTUNREACH

The following REXX socket API error numbers can be returned:

- 2001 EINVALIDRXSOCKETCALL

- 2009 ESOCKETNOTDEFINED

LE C/C++ equivalent

```
int bind2addrsel(int socket, struct sockaddr *address, int *address_len);
```

CLOSE

Use the CLOSE command to close a socket and release the resources that are associated with the socket descriptor.

If the socket descriptor is a connected stream socket, the connection to the remote host is also closed. If a connected stream socket is closed but has input data still pending on the socket, the CLOSE command causes the connection to be reset.

Format

➤ SOCKET — (— "CLOSE" — , — *socketid* —) ➤

Parameters

socketid

The socket descriptor

Returned value

The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

See [Appendix B, "Socket call error return codes,"](#) on page 743 for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 9 EBADF

The following REXX socket API error numbers can be returned:

- 2001 EINVALDRXSOCKETCALL
- 2009 ESOCKETNOTDEFINED

LE C/C++ equivalent

```
int close(int socket);
```

Code example

```
/* REXX EZARXR02 */
/*
 * This sample demonstrates the use of the CLOSE socket commands.
 * The program will open a STREAM socket, and if successful,
 * the socket will be closed.
 */
src = socket("INITIALIZE","MYSET01",10);
if perror(src,"INITIALIZE") = 0 then do
  src = socket("SOCKET","AF_INET6","STREAM");
  if perror(src,"SOCKET") = 0 then do
    parse var src l_retcode l_sockid
    src = perror(socket("CLOSE",l_sockid),"CLOSE");
  end; /* SOCKET */
end; /* INITIALIZE */
src = perror(socket("TERMINATE","MYSET01"),"TERMINATE");
exit 0;

/* This routine returns -1 if the first word of arg 1 is not zero */
perror: if word(arg(1),1) = 0 then return 0; else
  Say arg(2) "Error : "arg(1);
  return -1;
```

Figure 140. CLOSE command example

CONNECT

A client application uses the CONNECT command to establish a connection between a local socket and a remote socket.

The command supports both blocking and nonblocking sockets. When the socket is in blocking mode, the function does not return until a connection with the remote peer is established or until an error is received. When the socket is in nonblocking mode, the function returns immediately with either the 36 EINPROGRESS return code or an error.

The CONNECT command performs differently depending on the socket type:

Stream (TCP) sockets

If the application has not already issued an explicit bind, the CONNECT command completes the bind of the socket. The API then attempts to establish a connection to the remote socket that is specified by the *name* parameter. You can call the CONNECT command only once. Issuing additional CONNECT commands results in a 56 EISCONN error.

Datagram (UDP) sockets

The CONNECT command enables an application to associate a socket with the socket name of a peer. The socket then is considered to be a connected UDP socket. You can call the CONNECT command multiple times with different peer names to change the socket association.

Rules:

- Using the CONNECT command on a UDP socket does not change the UDP protocol from a connectionless to a connection-based protocol. The UDP socket remains connectionless. The primary benefit of using connected UDP sockets is to limit communication with a single remote application.
- When a UDP socket becomes a connected UDP socket, it can no longer use the SENDTO and RECVFROM commands. Connected UDP sockets use the socket commands READ, WRITE, SEND, or RECV to communicate with the remote peer, instead of using the SENDTO and RECVFROM commands.

Tips:

- For nonblocking sockets, use the SELECT command to determine when a connection has been established. Test for the ability to write to the socket.
- A connected UDP socket can revert back to an unconnected UDP socket by calling CONNECT with 0 or AF_UNSPEC specified in the *domain* field of the *name* parameter.

Format

➤ SOCKET — (— "CONNECT" — , — *socketid* — , — *name* —) ➤

Parameters

socketid

The descriptor of the local socket.

name

Identifies the remote socket.

The format for the *name* parameter depends on the socket type:

AF_INET sockets (IPv4)

name = "domain portid ipaddress"

AF_INET6 sockets (IPv6)

name = "domain portid flowinfo ipaddress scopeid"

where

- The *domain* value is the decimal number 2 for AF_INET and the decimal number 19 for AF_INET6.
- The *portid* value is the port number.
- The *ipaddress* value is the IP address of the remote host. It must be an IPv4 address for AF_INET and an IPv6 address for AF_INET6.
- The *flowinfo* value must be 0.
- The *scopeid* value identifies the interfaces that are applicable for the scope of the address that is specified in the *ipaddress* field. For a link-local IP address, the *scopeid* field can specify a link index, which identifies a set of interfaces. For all other scopes, the *scopeid* field must be set to 0. Setting the *scopeid* field to 0 indicates that any address type and scope can be specified.

Returned value

The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

Tip: When a connection attempt is made with a nonblocking socket, the string 36 EINPROGRESS is returned to the application. The program should check for this condition when using nonblocking sockets.

See [Appendix B, "Socket call error return codes,"](#) on page 743 for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 1 EPERM
- 9 EBAADF
- 35 EWOULDBLOCK
- 36 EINPROGRESS
- 37 EALREADY
- 47 EAFNOSUPPORT
- 48 EADDRINUSE
- 49 EADDRNOTAVAIL
- 51 ENETUNREACH
- 56 EISCONN
- 60 ETIMEDOUT
- 61 ECONNREFUSED

The following REXX socket API error numbers can be returned:

- 2001 EINVALDRXSOCKETCALL
- 2009 ESOCKETNOTDEFINED

LE C/C++ equivalent

```
int connect(int socket, struct sockaddr *address, int address_len);
```

Code example

```
/* REXX EZARXR03 */
/*
 * This sample demonstrates the use of the INITIALIZE, SOCKET,
 * CONNECT, GETSOCKNAME, SEND, RECV, CLOSE and TERMINATE
 * socket commands.
 *
 * The program will INITIALIZE a socket set and create a STREAM
 * socket. If successful an attempt will be made to connect to
 * port 7 using the loopback address and 7 bytes of data will be
 * sent. The program will then wait for the data to be echoed
 * back.
 *
 * Port 7 is the well known port for the ECHO server. The ECHO
 * Server for the z/OS Communication Server is the MISC SERV. For
 * Information on setting up the MISC SERV see the IP Configuration
 * Reference.
 *
 * The example EZARXR01 can be modified to echo data back by adding
 * a SEND command after the RECV command. This is left as an exercise.
 *
 * GUIDELINE: It is generally recommended that a program loop around
 * the RECV command to ensure that all data is read off
 * the socket.
 */
src = socket("INITIALIZE","MYSET01");
if perror(src,"INITIALIZE") = 0 then do
  src = socket("SOCKET","AF_INET","STREAM");
  if perror(src,"SOCKET") = 0 then do
    l_socketid = WORD(src,2);
    l_RMTname = "AF_INET 7 127.0.0.1";
    src = socket("CONNECT",l_socketid,l_RMTname);
    if perror(src,"CONNECT") = 0 then do
      src = socket("GETSOCKNAME",l_socketid);
      if perror(src,"GETSOCKNAME") = 0 then do
        l_LOCname = SUBWORD(src,2);
        Say "The local socket name is: "l_LOCname;
        src = socket("SEND",l_socketid,"*****");
        if perror(src,"SEND") = 0 then do
          src = socket("RECV",l_socketid);
          if perror(src,"RECV") = 0 then
            Say "Echoed data: " word(src,3);
          end; /* SEND*/
        end; /* GETSOCKNAME*/
      end; /* CONNECT */
    end; /* SOCKET */
    src = socket("CLOSE",l_socketid);
    src = perror(src,"CLOSE");
  end; /* INITIALIZE */
  src = socket("TERMINATE","MYSET01");
  src = perror(src,"TERMINATE");
  exit 0;

/* This routine returns -1 if the first word if arg 1 is not zero */
peror: if word(arg(1),1) = 0 then return 0; else
  Say arg(2) "Error : "arg(1);
  return -1;
```

Figure 141. CONNECT command example

FCNTL

Use the FCNTL command to control the operating characteristics of a socket.

Format

➤ SOCKET — (— "FCNTL" — , — *socketid* — , — *fcmd* — { — , — BLOCKING — } { — , — *fvalue* — }) ➤

Parameters

socketid

The socket descriptor.

fcmd

The command to be run. The following commands are available:

F_SETFL

Sets the status flags for the socket

F_GETFL

Retrieves the status flags of the socket

fvalue

One of the following flags:

BLOCKING

Puts a socket into blocking mode. If the targeted socket is in nonblocking mode and there is no data on the socket, issuing this command causes socket commands that support nonblocking socket descriptors to return the 35 EWOULDBLOCK error message. By default, the *fvalue* parameter is set to BLOCKING.

NON-BLOCKING

Puts a socket into nonblocking mode. The value FNDELAY is also accepted.

Returned value

The command returns a string that contains the return code. If the F_GETFL flag is issued, the string also contains the flag. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

The string 0 BLOCKING is an example of what this command might return.

See [Appendix B, "Socket call error return codes," on page 743](#) for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 1 EPERM
- 9 EBAADF
- 35 EWOULDBLOCK

The following REXX socket API error numbers can be returned:

- 2001 EINVALDRXSOCKETCALL
- 2009 ESOCKETNOTDEFINED

LE C/C++ equivalent

```
int fcntl(int socket, int cmd, ...);
```


Code example

```

/* REXX EXARXR04 */
/*
 * This sample demonstrates the use of the FCNTL
 * socket command.
 *
 * The program will open a STREAM socket and use the
 * FCNTL command to set the socket to NON-BLOCKING
 * mode.
 */
src = socket("INITIALIZE","MYSET01");
if perror(src,"INITIALIZE") = 0 then do
  src = socket("SOCKET","AF_INET","STREAM");
  if perror(src,"SOCKET") = 0 then do
    l_socketid = WORD(src,2);
    src = socket("FCNTL",l_socketid,F_SETFL,"NON-BLOCKING");
    src = socket("FCNTL",l_socketid,F_GETFL);
    Say src;
  end; /* SOCKET */
  src = socket("CLOSE",l_socketid);
  src = perror(src,"CLOSE");
end;
src = socket("TERMINATE","MYSET01");
src = perror(src,"TERMINATE");
exit 0;

/* This routine returns -1 if the first word of arg 1 is not zero */
perror: if word(arg(1),1) = 0 then return 0; else
  Say arg(2) "Error : "arg(1);
  return -1;

```

Figure 142. FCNTL command example

GETADDRINFO

Use the GETADDRINFO command to resolve host or service name information.

This command translates the name of a service location (host name) or a service name. The command returns a set of NAME strings and associated information; this information can be used to create a socket with which to address the specified service or to send a datagram to the specified service. For information about the format of the NAME string, see [“How structures are represented”](#) on page 580.

Guideline: Use the application to cycle through each NAME string until a successful connection is established. An example is provided in [“Code example”](#) on page 603.

Tip: You can use a resolver trace to determine why a resolver command failed. See [z/OS Communications Server: IP Diagnosis Guide](#).

Format

```

➔ SOCKET — ( — "GETADDRINFO" — , — node_service — , — flags — , —
    family — , — socktype — , — protocol — , — eflags — ) ➔

```

Parameters

node_service

This variable takes one of the following formats:

```

➔ node — , — service — ➔

```



where:

node

The host name or IP address. If the value of the *node* parameter is an IP address, you also must issue the AI_NUMERICHOST flag. The value of the *node* parameter can be up to 255 bytes in length.

Scope information can be appended to the host name, using the following format:

```
"node%scope information"
```

For example, you could set the *node* parameter to "MYNODE%23". For more information, see *z/OS Communication Server: IPv6 Network and Application Design Guide*.

service

The TCP/IP service that is queried. If the value of the *service* parameter is a port number, you must specify the AI_NUMERICSERV flag. The value of the *service* parameter can be up to 32 bytes in length.

flags

To specify multiple flags, code the *flags* parameter as a space-delimited string. The following flags are supported:

AI_PASSIVE

Specifies how to fill the returned socket NAME string. If this flag is set, the returned NAME string can be used with the BIND command to bind a socket for accepting new connection requests.

Rules:

- If the AI_PASSIVE flag is specified and the *node* parameter is not specified, the IP address portion of the NAME string is set to either the IPv4 address (INADDR_ANY) or to the IPv6 unspecified address (in6addr_any).
- If the AI_PASSIVE flag is not specified, then the returned NAME string can be used with the CONNECT or the SENDTO commands.
- If the AI_PASSIVE flag is not specified and the *node* parameter is not specified, the IP address portion of the NAME string is set to the default loopback address: 127.0.0.1 (IPv4) or ::1 (IPv6).

If you issue the following command:

```
src = socket("GETADDRINFO",,54123,"AI_PASSIVE","AF_UNSPEC")
```

Then the following string is returned:

```
0 '' AF_INET6 54123 0 ::0 0 AF_INET 54123 0.0.0.0
```

See example 2 in [“Code example” on page 603](#) for one method of how to use this flag.

AI_CANONNAMEOK

Specifies that the canonical name that corresponds to the *node* parameter is returned. The *node* parameter must also be issued.

If you issue the following command:

```
src = socket("GETADDRINFO","chile",21,"AI_CANONNAMEOK")
```

Then the following string is returned:

```
0 host.department.com AF_INET 21 10.11.103.1
```

Tip: See [“Code example” on page 603](#) for examples of how to use this flag.

AI_NUMERICHOST

If you specify this flag, the *node* parameter must be specified as an IP address, for example, 10.11.103.1. Otherwise, the command fails with a 1 EAI_NONAME return code.

If you issue the following command:

```
src = socket("GETADDRINFO", "10.11.103.1", , "AI_NUMERICHOST")
```

Then the following string is returned:

```
0 '' AF_INET 0 10.11.103.1
```

AI_NUMERICSERV

If you specify this flag, you must specify the *service* parameter as a port number, for example, 1821. Otherwise, the command fails with a 1 EAI_NONAME return code.

If you issue the following command

```
src = socket("GETADDRINFO", , 23, "AI_NUMERICSERV")
```

Then the following string is returned.

```
0 '' AF_INET6 23 0 ::1 0 AF_INET 23 127.0.0.1
```

AI_V4MAPPED

If you specify this flag and the value of the *family* parameter is AF_INET6 or AF_UNSPEC, the caller accepts IPv4-mapped IPv6 addresses.

Rules:

- If the value of the *family* parameter is AF_INET6, a query for IPv4 addresses is made if the AI_ALL flag is specified or if no IPv6 addresses are found. Any IPv4 addresses that are found are returned as IPv4-mapped IPv6 addresses.
- If the value of the *family* parameter is AF_UNSPEC, queries are made for both IPv6 and IPv4 addresses. If IPv4 addresses are found and IPv6 is supported on the system, the IPv4 addresses are returned as IPv4-mapped IPv6 addresses.
- Otherwise, this flag is ignored.

If you issue the following command:

```
src = socket("GETADDRINFO", "CHILE", , "AI_V4MAPPED AI_CANONNAMEOK");
```

Then the following string is returned:

```
0 host.department.com AF_INET6 0 0 ::FFFF:10.11.103.1 0
```

AI_ALL

If you specify this flag, the NAME strings that are returned depend on the value of the *family* parameter.

Rules:

- If the value of the *family* parameter is AF_INET6, the AI_V4MAPPED flag must also be set to indicate that both IPv4-mapped IPv6 addresses and IPv6 addresses are acceptable.
- If the value of the *family* parameter is AF_UNSPEC, AI_ALL is accepted, but has no impact on the processing. No matter AI_ALL is specified or not, two queries are made. The first query is for IPv6 addresses and if successful, the IPv6 addresses are returned. The second query is for IPv4 addresses:
 - If the AI_V4MAPPED flag is specified and the system supports IPv6, the IPv4 addresses are returned as IPv4-mapped IPv6 addresses.
 - If the AI_V4MAPPED flag is not specified or the system does not support IPv6, the IPv4 addresses are returned.

- Otherwise, this flag is ignored.

If you issue the following command:

```
src = socket("GETADDRINFO", "CHILE", , ,
"AI_ALL AI_V4MAPPED AI_CANONNAMEOK", ,
"AF_UNSPEC");
```

The following string is returned (all on one line):

```
0 CHILE.department.com AF_INET6 0 0 ::FFFF:10.11.103.1 0
AF_INET6 0 0 2001:10:11:103::1 0
```

AI_ADDRCONFIG

If you specify this flag, the node is queried if the resolver determines that one of the following conditions is true:

- The system is IPv6 enabled and has at least one IPv6 interface. In this case, the resolver makes a query for AAAA DNS records (IPv6).
- The system is IPv4 enabled and has at least one IPv4 interface. In this case, the resolver makes a query for A DNS records (IPv4).

The loopback address is not a valid interface for this flag.

AI_EXTFLAGS

If you specify this flag, the extended form of the `getaddrinfo` function is requested. The extended form allows additional hints to be passed to the resolver to determine the order of destination addresses that are returned. This flag affects only the order of IPv6 addresses that are returned, if any. If `AI_EXTFLAGS` flag is specified, the *eflags* parameter must be specified.

family

Limits the returned information to a specific address family. The following families are supported:

AF_UNSPEC

Any protocol family. The value 0 is accepted also.

AF_INET

IPv4 families. The value 2 is accepted also.

AF_INET6

IPv6 families. The value 19 is accepted also.

socktype

Limits the returned information to a specific socket type. If no socket type is specified, the command returns address information for all types. The following socket types are supported:

Type name	Decimal value	Description
SOCK_STREAM	1	Stream socket
SOCK_DGRAM	2	Datagram socket
SOCK_RAW	3	Raw-protocol interface

Consider the following points:

- If the value of the *socktype* parameter is set to any value other than `SOCK_STREAM`, `SOCK_DGRAM`, or `SOCK_RAW`, the `GETADDRINFO` command fails with a 9 `EAI_SOCKTYPE` return code.
- If the value of the *socktype* parameter is `SOCK_RAW`, the value of the *service* parameter must be numeric.
- If the value of the *socktype* parameter is set to `SOCK_STREAM` or `SOCK_DGRAM`, then the resolver searches the services file for a service name.

If the *socktype* and *protocol* parameters are both specified as 0, then the `GETADDRINFO` command is processed in the following way:

- If the value of the *service* parameter is null or numeric, then any returned address information has the default *socktype* value of SOCK_STREAM.
- If the value of the *service* parameter is a service name, for example, FTP, then the GETADDRINFO command searches the appropriate services file twice. The first search uses SOCK_STREAM as the protocol, and the second search uses SOCK_DGRAM as the protocol. There is no default socket-type provision in this case.

If both the *socktype* and *protocol* parameters are specified as nonzero values, then the values must be compatible, regardless of the value specified by the *service* parameter. In this case, compatible means one of the following combinations of parameters:

- The *socktype* parameter is SOCK_STREAM and the *protocol* parameter is IPPROTO_TCP.
- The *socktype* parameter is SOCK_DGRAM and the *protocol* parameter is IPPROTO_UDP.
- The *socktype* parameter is SOCK_RAW, in which case the *protocol* parameter can have any value.

protocol

Limits the returned information to a specific protocol. The value 0 indicates that the caller accepts any protocol. The following protocols are supported:

Protocol name	Decimal value	Description
IPPROTO_TCP	6	TCP
IPPROTO_UDP	16	UDP

Consider the following points:

- If the *socktype* parameter is 0 and the *protocol* parameter is not 0, the only acceptable input values for the *protocol* parameter are IPPROTO_TCP and IPPROTO_UDP. If any other values are issued for the *protocol* parameter, the GETADDRINFO command fails with a 7 EAI_BADFLAGS return code.
- If the *protocol* and *socktype* parameters are both specified as 0, then the GETADDRINFO command is processed in the following way:
 - If the *service* parameter value is null or numeric, then any returned address information assumes that the socket type is SOCK_STREAM.
 - If the *service* parameter is specified as a service name, for example, FTP, then the GETADDRINFO command searches the applicable services file twice. The first search uses the protocol SOCK_STREAM, and the second search uses the protocol SOCK_DGRAM. There is no default socket type provision in this case.
- If both the *protocol* and *socktype* parameters are specified as nonzero values, the values must be compatible, regardless of the value that is specified by the *service* parameter. In this context, compatible means one of the following combinations of parameters:
 - The *socktype* parameter is SOCK_STREAM and the *protocol* parameter is IPPROTO_TCP.
 - The *socktype* parameter is SOCK_DGRAM and the *protocol* parameter is IPPROTO_UDP.
 - The *socktype* parameter is SOCK_RAW, in which case the *protocol* parameter can have any value.
- If the lookup for the value specified by the *service* parameter fails (for example, the *service* name does not appear in the applicable services file), then the GETADDRINFO command fails with the 8 EAI_SERVICE return code.

eflags

A fullword binary field that specifies the source IPv6 address selection preferences. If AI_EXTFLAGS flag is specified in FLAGS, this field is required. To specify multiple eflags, code the *eflags* parameter as a space-delimited string.

This field must have the value of 0 or of one or more of the following values:

IPV6_PREFER_SRC_HOME

Requests that resolver returns destination IPv6 addresses that can be reached by a home IPv6 source address before it returns destinations that can be reached by a care-of IPv6 source address. This is the default behavior if AI_EXTFLAGS flag is not specified.

IPV6_PREFER_SRC_COA

Requests that resolver returns destination IPv6 addresses that can be reached by a care-of IPv6 source address before it returns destinations that can be reached by a home IPv6 source address.

IPV6_PREFER_SRC_PUBLIC

Requests that resolver returns destination IPv6 addresses that can be reached by a public IPv6 source address before it returns destinations that can be reached by a temporary IPv6 source address.

IPV6_PREFER_SRC_TMP

Requests that the resolver returns destination IPv6 addresses that can be reached by a temporary IPv6 source address before it returns destinations that can be reached by a public IPv6 source address.

IPV6_PREFER_SRC_CGA

Requests that resolver returns destination IPv6 addresses that can be reached by a cryptographically generated IPv6 source address before it returns destinations that can be reached by a non-cryptographically generated IPv6 source address.

IPV6_PREFER_SRC_NONCGA

Requests that resolver returns destination IPv6 addresses that can be reached by a non-cryptographically generated IPv6 source address before it returns destinations that can be reached by a cryptographically generated IPv6 source address.

ZERO

No preferences are specified.

Any EFLAGS specification other than the value of 0 or one of the options listed above causes an EINVALDRXSOCKETCALL error to be returned.

If contradictory EFLAGS, for example, IPV6_PREFER_SRC_TMP and IPV6_PREFER_SRC_PUBLIC, are specified, the GETADDRINFO call returns EINVALCOMBINATION.

Returned value

The command returns a string that contains the return code, canonical name, and a NAME string or list of NAME strings. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

The following string is an example of what is returned by the GETADDRINFO command:

```
0 RALEIGH.IBM.COM name1 name2 name3
```

In the example, 0 is the return code, RALEIGH.IBM.COM is the canonical name, and name1 name2 name3 is a list of NAME strings. Depending on the flags that were issued, these names can be IPv4 or IPv6 values.

Tip: For a description of the format of a socket name, see [“CONNECT” on page 593](#) or [“BIND” on page 588](#).

See [Appendix B, “Socket call error return codes,” on page 743](#) for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 1 EAI_NONAME
- 2 EAI_AGAIN
- 5 EAI_FAIMLY
- 6 EAI_MEMORY

- 7 EAI_BADFLAGS
- 8 EAI_SERVICE
- 9 EAI_SOCKTYPE
- 9 EBAADF
- 35 EWOULDBLOCK

The following REXX socket API error numbers can be returned:

- 2001 EINVALDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE
- 2006 ESOCKETNOTALLOCATED

LE C/C++ equivalent

```
int getaddrinfo(const char *nodename, const char *servname,
               const struct addrinfo *hints, struct addrinfo **res);
```

Code example

```
/* REXX EZARXR05 */
/*
 * This sample demonstrates a use of the GETADDRINFO command.
 * It is possible that the GETADDRINFO command will return
 * more then one name address. The program shows one
 * technique that can be used to parse the information
 * returned. After a successful connection has been established
 * the program will sends data to a server listeneing on port
 * 54777. The program then waits for a reply.
 *
 * HINT: The program code provided under the IOCTL command can
 * be used as a server for this sample.
 *
 * GUIDELINE: It is generally recommended that a program loop around
 * the RECV command to ensure that all data is read off
 * the socket. This sample does not follow the guideline.
 */
src = socket("INITIALIZE","MYSET01",10);
if perror(src,"INITIALIZE") = 0 then do
  src = socket("SOCKET","AF_INET6","SOCK_STREAM");
  if perror(src,"SOCKET") \= 0 then signal ENDPROGRAM
  parse var src l_retcode l_sockid
  /* *****
   * Issue GETADDRINFO command. Request that all IPv6 address
   * information be returned, and if possible provide the
   * canon name. IPV4 addresses are to be mapped to
   * IPv6 mapped addresses. Names will be returned as AF_INET6
   * NAMES.
   * *****/
  src = socket("GETADDRINFO","MVS150",54777,,
              "AI_ALL AI_CANONNAMEOK AI_NUMERICSERV",
              "AI_V4MAPPED","AF_INET6","SOCK_STREAM",,
              "IPPROTO_TCP");
  if perror(src,"GETADDRINFO") = 0 then do
    parse var src l_retcode l_canonname l_names
    l_LOCName = "AF_INET6 0 0 IN6ADDR_ANY 0";
    src = socket("BIND", l_sockid, l_LOCName);
    if perror(src,"BIND") \= 0 then signal ENDPROGRAM
    /* *****
     * It is possible that GETADDRINFO returned multiple
     * name structures. Cycle through them until a
     * successful connection is achived or none are left.
     * *****/
    l_done = "FALSE";
    l_connectOK = "FALSE";
    do while l_names \= '' | l_done = "FALSE"
      if word(l_names,1) = "AF_INET6" then do
        parse var l_names . l_port l_flow l_addr,
                  l_scope l_names
        l_RMTname = "AF_INET6 "l_port" "l_flow" "l_addr,
                  ||" "l_scope;
```

```

end; /* AF_INET6 */
else do
    parse var l_names . l_port l_addr l_names
    l_RMTname = "AF_INET "l_port" "l_addr;
end; /* AF_INET */
Say "Attempting connection using RMT NAME: ",
    l_RMTname;
src = socket("CONNECT",l_sockid,l_RMTname);
if perror(src,"CONNECT") = 0 then do
    Say "...Connected";
    l_connectOK = "TRUE";
    l_done = "TRUE";
end; /* CONNECT */

else do
    l_done = "TRUE";
end;
end; /* DO LOOP */
if l_connectOK = "TRUE" then do
    l_data = time() "**** *";
    src = socket("GETSOCKNAME",l_sockid);
    Say "GETSOCKNAME: "src;
    src = socket("SEND",l_sockid,l_data);
    if perror(src,"SEND") = 0 then do
        src = socket("RECV",l_sockid);
        if perror(src,"RECV") = 0 then do
            parse var src . l_amtdata l_data
            Say "Received "l_amtdata" bytes from ",
                l_addr " on" port " l_port";
            Say "The received data is: "l_data;
        end; /* RECV */
    end; /* SEND */
end; /* CONNECT OK */
else do /* CONNECT not OK */
    say "No Connection to remote host could be",
        "established";
end;
end; /* GETADDRINFO */
end; /* INITIALIZE */

ENDPROGRAM:
src=socket("CLOSE",l_sockid);
src=perror(src,"CLOSE");
src=socket("TERMINATE","MYSET01");
src=perror(src,"TERMINATE");
exit 0;

/* This routine returns -1 if the first word if arg 1 is not zero */
perror: if word(arg(1),1) = 0 then return 0; else
    Say arg(2) "Error : "arg(1);
    return -1;

/* REXX EZARXR32 */
/*
* This example shows how to use the GETADDRINFO command
* to obtain a list of NAME constructs suitable for
* use with the BIND command using the well-known port 54123.
* The addr fields of the NAME constructs returned will be
* set to
*
* This example also shows how to use the SELECT command to
* monitor listening sockets for for new connections.
*
* In the case of 2 name constructs being returned the
* first successful bind will be used.
*
* HINT: To limit the NAME structures to a specific
*       Address family change AF_UNSPEC to either
*       AF_INET or AF_INET6
*
* HINT: Coding a hostname for the node paramter will result
*       in the ipaddress fields of the NAME constructs to
*       have a specific interface address assigned.
*/

l_selectlist = '';
src = socket("INITIALIZE","MYSET01");
src = socket("GETADDRINFO",,54123,"AI_PASSIVE",

```


Parameters

domain

The domain. This parameter is optional for IPv4, but it is required for IPv6. If this parameter is not specified, by default, the *domain* parameter is set to AF_INET. The following domain values are supported:

- AF_UNSPEC or 0
- AF_INET or 2
- AF_INET6 or 19

Returned value

The command returns a string that contains the return code and the application identifier. The return code can be 0 or the REXX API error number. The return code 0 indicates that the requested socket command was completed successfully.

The application identifier has the following format:

```
clientid = domain userid socketset
```

The following examples show what is returned for the specific address families:

IPv4

```
0 AF_INET IBMUSER SOCSET01
```

IPv6

```
0 AF_INET6 IBMUSER SOCSET01
```

See Appendix B, “Socket call error return codes,” on page 743 for additional information about the numeric error codes that are returned by this command.

The following REXX socket API error numbers can be returned:

- 2001 EINVALDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE
- 2013 EINVALCLIENTID

LE C/C++ equivalent

```
int getclientid(int domain, struct clientid *clientid);
```

Code example

```
/* REXX EZARXR06 */
/*
 * This sample demonstrates the use of the GETCLIENTID
 * socket command.
 */
src = socket("INITIALIZE","MYSET01");
if perror(src,"INITIALIZE") \= 0 then signal ENDPROGRAM;
src = socket("GETCLIENTID");
Say src;

ENDPROGRAM:
src = socket("TERMINATE","MYSET01");
src = perror(src,"TERMINATE");

exit 0;

/* This routine returns -1 if the first word if arg 1 is not zero */
perror: if word(arg(1),1) = 0 then return 0; else
        Say arg(2) "Error : "arg(1);
        return -1;
```

Figure 143. GETCLIENTID command example

GETDOMAINNAME

Use the GETDOMAINNAME command to retrieve the name of the domain to which the current TCP/IP stack belongs.

Tip: You can use a resolver trace to determine why a resolver command failed. See [z/OS Communications Server: IP Diagnosis Guide](#).

Format

➤ SOCKET — (— "GETDOMAINNAME" —) ➤

Parameters

This command has no parameters.

Returned value

The command returns a string that contains the return code and the domain, for example, 0 RALEIGH.IBM.COM.

See Appendix B, “Socket call error return codes,” on page 743 for additional information about the numeric error codes that are returned by this command.

The following REXX socket API error number can be returned:

- 2005 ESUBTASKNOTACTIVE

LE C/C++ equivalent

This command has no LE C/C++ equivalent.

Code example

```
/* REXX EZARXR07 */
/*
 * This sample demonstrates the use of the GETDOMAINNAME
 * socket command.
 */
src = socket("INITIALIZE","MYSET01");
if perror(src,"INITIALIZE") = 0 then do
  Say socket("GETDOMAINNAME");
end;

src = socket("TERMINATE","MYSET01");
src = perror(src,"TERMINATE");
exit 0;

/* This routine returns -1 if the first word of arg 1 is not zero */
perror: if word(arg(1),1) = 0 then return 0; else
  Say arg(2) "Error : "arg(1);
  return -1;
```

Figure 144. GETDOMAINNAME command example

GETHOSTBYADDR

Use the GETHOSTBYADDR command to resolve an IPv4 address to a host name.

This command uses a domain name system (DNS) server. If the IP address is not resolved by the DNS server, then the resolver searches the local hosts tables. For information about the local host tables, see [z/OS Communications Server: IP Configuration Guide](#).

Restriction: This command does not support IPv6 addresses.

Tips:

- Use either the GETADDRINFO or GETNAMEINFO command to resolve a IPv6 address to a host name.
- You can use a resolver trace to determine why a resolver command failed. See [z/OS Communications Server: IP Diagnosis Guide](#).

Format

➔ SOCKET — (— "GETHOSTBYADDR" — , — *ipaddress* — , — *domain*) ➔

Parameters

ipaddress

The IP address of the remote host.

domain

The network domain to which the IP address belongs. The only supported domain is AF_INET or 2.

Returned value

This command returns a string that contains the return code and the host name, for example, 0 ABCD.RALEIGH.IBM.COM. The return code can be 0 or the REXX API error number. The return code 0 indicates that the requested socket command was completed successfully.

See [Appendix B, "Socket call error return codes," on page 743](#) for additional information about the numeric error codes that are returned by this command.

The following REXX socket API error numbers can be returned:

- 2001 EINVALDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE

- 2017 EIPADDRNOTFOUND

LE C/C++ equivalent

```
struct hostent *gethostbyaddr(char *address,
    int address_len, int domain);
```

Code example

```
/* REXX EZARXR08 */
/*
 * This sample demonstrates the use of the GETHOSTBYADDR
 * socket command.
 */
src = socket("INITIALIZE", "MYSET01");
if perror(src, "INITIALIZE") = 0 then do
    Say socket("GETHOSTBYADDR", "128.123.222.1");
end;

src = socket("TERMINATE", "MYSET01");
src = perror(src, "TERMINATE");
exit 0;

/* This routine returns -1 if the first word if arg 1 is not zero */
perror: if word(arg(1), 1) = 0 then return 0; else
    Say arg(2) "Error : "arg(1);
    return -1;
```

Figure 145. GETHOSTBYADDR command example

GETHOSTBYNAME

Use the GETHOSTBYNAME command to resolve a host name to an IPv4 address.

This command uses a DNS server. Any trailing blanks are removed from the host name. If the host is multi-homed, then this command returns a list of the space-delimited IP addresses that are associated with the host name. If the host name is not resolved by the DNS server, then the resolver searches the local hosts tables. For information about the local host tables, see [z/OS Communications Server: IP Configuration Guide](#).

Restriction: This command does not support IPv6 addresses.

Tips:

- Use either the GETADDRINFO or GETNAMEINFO command to resolve a IPv6 address to a host name.
- You can use a resolver trace to determine why a resolver command failed. See [z/OS Communications Server: IP Diagnosis Guide](#).

Format

```
➤ SOCKET — ( — "GETHOSTBYNAME" — , hostname ) ➤
                        Fully qualified hostname
```

Parameters

hostname

The name of the remote host. The parameter cannot be longer than 255 characters.

Fully qualified hostname

The fully qualified name of the host in the format *hostname.domainname*. The parameter cannot be longer than 255 characters.

Returned value

This command returns string that contains the return code and one or more space-delimited IPv4 addresses, for example, 0 1.2.3.4 5.4.3.2 1.2.3.5. The return code can be 0 or the REXX API error number. The return code 0 indicates that the requested socket command was completed successfully.

See [Appendix B, “Socket call error return codes,”](#) on page 743 for additional information about the numeric error codes that are returned by this command.

The following REXX socket API error numbers can be returned:

- 2001 EINVALIDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE
- 2016 EHOSTNOTFOUND
- 2019 ENORECOVERY

LE C/C++ equivalent

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

Code example

```
/* REXX EZARXR09 */
/*
 * This sample demonstrates the use of the GETHOSTBYNAME
 * socket command.
 */
src = socket("INITIALIZE","MYSET01");
if perror(src,"INITIALIZE") = 0 then do
    Say socket("GETHOSTBYNAME", "MYCOMPANY");
    Say socket("GETHOSTBYNAME", "MYCOMPANY.somewhere.com");
end;

src = socket("TERMINATE","MYSET01");
src = perror(src,"TERMINATE");
exit 0;

/* This routine returns -1 if the first word if arg 1 is not zero */
perror: if word(arg(1),1) = 0 then return 0; else
    Say arg(2) "Error : "arg(1);
    return -1;
```

Figure 146. GETHOSTBYNAME command example

GETHOSTID

Use the GETHOSTID command to return the primary IPv4 address for the current host. The primary address is the IP address of the default home address that is specified in the TCP/IP configuration file for the stack with which the current socket set is associated.

Restriction: This command does not support IPv6 addresses.

Format

➤ SOCKET — (— "GETHOSTID" —) ➤

Parameters

This command has no parameters.

Returned value

This command returns a string that contains the return code and the primary IPv4 address of the host, for example, "0 10.11.103.1". The return code can be 0 or the REXX API error number. The return code 0 indicates that the requested socket command was completed successfully.

See [Appendix B, "Socket call error return codes," on page 743](#) for additional information about the numeric error codes that are returned by this command.

The following REXX socket API error numbers can be returned:

- 2001 EINVALIDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE
- 2016 EHOSTNOTFOUND
- 2019 ENORECOVERY

LE C/C++ equivalent

```
int gethostid();
```

Code example

```
/* REXX EZARXR10 */
/*
 * This sample demonstrates the use of the GETHOSTID
 * socket command.
 */
src = socket("INITIALIZE","MYSET01");
if perror(src,"INITIALIZE") = 0 then do
    say socket("GETHOSTID");
end;
src = socket("TERMINATE","MYSET01");
src = perror(src,"TERMINATE");
exit 0;

/* This routine returns -1 if the first word if arg 1 is not zero */
pererror: if word(arg(1),1) = 0 then return 0; else
    Say arg(2) "Error : "arg(1);
    return -1;
```

Figure 147. GETHOSTID command example

GETHOSTNAME

Use the GETHOSTNAME command to return the name of the host on which the application is running.

Format

➤ SOCKET — (— "GETHOSTNAME" —) ➤

Parameters

This command has no parameters.

Returned value

This command returns a string that contains the return code and the name of the host system, for example, 0 MYHOST01. If the host name is longer than 256 bytes, it is truncated. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

See [Appendix B, “Socket call error return codes,”](#) on page 743 for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error number can be returned:

- 2018 ETRYAGAIN

The following REXX socket API error number can be returned:

- 2005 ESUBTASKNOTACTIVE

LE C/C++ equivalent

```
rc = int gethostname(char *name, size_t namelen);
```

Code example

```
/* REXX EZARXR11 */
/*
 * This sample demonstrates the use of the GETHOSTNAME
 * socket command.
 */
src = socket("INITIALIZE","MYSET01");
if perror(src,"INITIALIZE") = 0 then do
  src = socket("GETHOSTNAME");
  Say "Host name is: "WORD(src,2);
end;
src = socket("TERMINATE","MYSET01");
src = perror(src,"TERMINATE");
exit 0;

/* This routine returns -1 if the first word if arg 1 is not zero */
perror: if word(arg(1),1) = 0 then return 0; else
  Say arg(2) "Error : "arg(1);
  return -1;
```

Figure 148. GETHOSTNAME command example

GETNAMEINFO

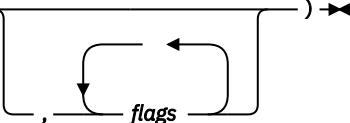
Use the GETNAMEINFO command to translate a socket address to a node name and service location.

This command can be used for both IPv4 or IPv6 socket addresses.

Tip: You can use a resolver trace to determine why a resolver command failed. See [z/OS Communications Server: IP Diagnosis Guide](#).

Format

➡ SOCKET — (— "GETNAMEINFO" — , — *name* —) ➡



Parameters

name

An IPv4 or IPv6 NAME string. If the NAME string is an IPv4-mapped IPv6 address, then the embedded IPv4 address is extracted; then, the lookup is performed on the IPv4 address. If the NAME string is an IPv6 unspecified address, the lookup is not performed and the 1 EAI_NONAME error code is returned.

The format for the *name* parameter depends on the socket type:

AF_INET sockets (IPv4)

name = "domain portid ipaddress"

AF_INET6 sockets (IPv6)

name = "domain portid flowinfo ipaddress scopeid"

where

- The *domain* value is the decimal number 2 for AF_INET and the decimal number 19 for AF_INET6.
- The *portid* value is the port number.
- The *ipaddress* value is the IP address of the remote host. It must be an IPv4 address for AF_INET and an IPv6 address for AF_INET6.
- The *flowinfo* value must be 0.
- The *scopeid* value identifies the interfaces that are applicable for the scope of the address that is specified in the *ipaddress* field. For a link-local IP address, the *scopeid* field can specify a link index, which identifies a set of interfaces. For all other scopes, the *scopeid* field must be set to 0. Setting the *scopeid* field to 0 indicates that any address type and scope can be specified.

If the *scopeid* field is specified and the destination is not a link-local address, the resolver ignores the *scopeid* field.

flags

An optional parameter that specifies the type of information that is returned. Separate multiple flags with spaces. If no flag is issued, the fully qualified host name is returned. The following flags are supported:

NI_NOFQDN

Returns the host name of the fully qualified domain name.

NI_NUMERICHOST

Returns only the numeric form of the host address.

NI_NAMEREQD

Returns an error if the host name cannot be found.

NI_NUMERICSERV

Returns the numeric form of the service.

NI_DGRAM

Indicates that the service query is for a datagram socket. If this flag is not issued, the GETNAMEINFO command assumes that the query is for a stream socket.

NI_NUMERICSCOPE

Returns only the numeric form of the *scopeid* interface index.

Do not specify both the NI_NUMERICHOST and the NI_NAMEREQD flags; otherwise, you get the EAI_FAIL (3) error. See [Appendix F, “GETNAMEINFO flags and returned information examples,” on page 777](#) for examples of returned information when various combinations of flags are used.

Returned value

This command returns a string that contains the return code, the host name, and the service. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

If a link-local IPv6 address is passed as input and the value of the *scopeid* parameter is not 0, then scope information is appended to the host name, using the format *host name%scope information*. For more information about scope information and the GETNAMEINFO command, see [z/OS Communications Server: IPv6 Network and Appl Design Guide](#).

The following string is an example of what is returned by the GETNAMEINFO command:

```
0 BOB01.THEWORLD.COM%23 echo
```

In the example, 0 is the return code, BOB01.THEWORLD.COM is the host name, 23 is the scope ID, and echo denotes the service. The numeric scope information is returned only if the NI_NUMERICSSCOPE flag is issued.

For information about the format of the NAME string, see [“How structures are represented”](#) on page 580. See Appendix B, [“Socket call error return codes,”](#) on page 743 for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 1 EAI_NONAME
- EAI_FAIL
- 5 EAI_FAMILY
- 7 EAI_BADFLAGS

The following REXX socket API error numbers can be returned:

- 2001 EINVALDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE

LE C/C++ equivalent

```
int getnameinfo(const struct sockaddr *sa, socklen_t salen,
               char *host, socklen_t hostlen, char *serv, socklen_t servlen,
               int flags);
```

Code example

```
/* REXX EZARXR13 */
/*
 * This sample demonstrates the use of the GETNAMEINFO
 * socket command.
 */
src = socket("INITIALIZE", "MYSET01");
if perror(src, "INITIALIZE") = 0 then
  Say socket("GETNAMEINFO", "AF_INET6 21 0 2000:197:11:103::1 0");
src = socket("TERMINATE", "MYSET01");
src = perror(src, "TERMINATE");
exit 0;

/* This routine returns -1 if the first word if arg 1 is not zero */
perror: if word(arg(1), 1) = 0 then return 0; else
  Say arg(2) "Error : "arg(1);
  return -1;
```

Figure 149. GETNAMEINFO command example

GETPEERNAME

Use the GETPEERNAME command to return the name of the remote peer that is connected to the socket.

Format

➤ SOCKET — (— "GETPEERNAME" — , — *socketid* —) ➤

Parameters

socketid

The socket descriptor

Returned value

This command returns a string that contains the return code and the name of the remote peer, for example, 0 NAME. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP

error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

For information about the format of the NAME string, see [“How structures are represented”](#) on page 580. See Appendix B, [“Socket call error return codes,”](#) on page 743 for additional information about the numeric error codes that are returned by this command.

Returned value

The following REXX TCP/IP error numbers can be returned:

- 9 EBADF
- 38 ENOTSOCK
- 45 EOPNOTSUPP
- 57 ENOTCONN

The following REXX socket API error numbers can be returned:

- 2001 EINVALIDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE
- 2009 ESOCKETNOTDEFINED
- 2012 EINVALIDNAME

LE C/C++ equivalent

```
int getpeername(int socket, struct sockaddr *name, int *namelen);
```

Code example

```
/* REXX EZARXR14 */
/*
 * This sample demonstrates the use of the GETPEERNAME
 * socket command.
 */
src = socket("INITIALIZE","MYSET01");
if perror(src,"INITIALIZE") = 0 then do
  src = socket("SOCKET","AF_INET","STREAM");
  if perror(src,"SOCKET") = 0 then do
    l_socketid = word(src,2);
    l_RMTName = "AF_INET 7 127.0.0.1";
    src = socket("CONNECT",l_socketid,l_RMTName);
    if perror(src,"CONNECT") = 0 then do
      src = socket("GETPEERNAME",l_socketid);
      if perror(src,"GETPEERNAME") = 0 then do
        l_PeerName = Subword(src,2);
        say "The remote peer is: "l_PeerName;
      end;
    end;
  end;
  src = perror(socket("CLOSE",l_socketid),"CLOSE");
end;
src = perror(socket("TERMINATE","MYSET01"),"TERMINATE");
exit 0;

/* This routine returns -1 if the first word if arg 1 is not zero */
pererror: if word(arg(1),1) = 0 then return 0; else
  Say arg(2) "Error : "arg(1);
  return -1;
```

Figure 150. GETPEERNAME command example

GETPROTOBYNAME

Use the GETPROTOBYNAME command to translate a network protocol name to a protocol number.

Format

➤ SOCKET — (— "GETPROTOBYNAME" — , — *protocolname* —) ➤

Parameters

protocolname

The name of the network protocol

Returned value

This command returns a string that contains the return code and the protocol number, for example, 0 6. The return code can be 0 or the REXX API error number. The return code 0 indicates that the requested socket command was completed successfully.

See [Appendix B, “Socket call error return codes,” on page 743](#) for additional information about the numeric error codes that are returned by this command.

The following REXX socket API error numbers can be returned:

- 2001 EINVALDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE

LE C/C++ equivalent

```
struct protoent *getprotobyname(char name);
```

Code example

```
/* REXX EZARXR15 */
/*
 * This sample demonstrates the use of the GETPROTOBYNAME
 * socket command.
 */
if perror(socket("INITIALIZE","MYSET01"),"INITIALIZE") = 0,
then do
  src = socket("GETPROTOBYNAME","TCP");
  Say "The TCP protocol is assigned the number of",
  WORD(src,2);
end;
src = perror(socket("TERMINATE","MYSET01"),"TERMINATE");
exit 0;

/* This routine returns -1 if the first word if arg 1 is not zero */
perror: if word(arg(1),1) = 0 then return 0; else
  Say arg(2) "Error : "arg(1);
  return -1;
```

Figure 151. GETPROTOBYNAME command example

GETPROTOBYNUMBER

Use the GETPROTOBYNUMBER command to translate a network protocol number to a protocol name.

Format

➤ SOCKET — (— "GETPROTOBYNUMBER" — , — *protocolnumber* —) ➤

Parameters

protocolnumber

The number of the network protocol

Returned value

This command returns a string that contains the return code and the protocol name, for example, 0 TCP. The return code can be 0 or the REXX API error number. The return code 0 indicates that the requested socket command was completed successfully.

See Appendix B, “Socket call error return codes,” on page 743 for additional information about the numeric error codes that are returned by this command.

The following REXX socket API error numbers can be returned:

- 2001 EINVALDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE

LE C/C++ equivalent

```
struct protoent *getprotobynumber(int proto);
```

Code example

```
/* REXX EZARXR16 */
/*
 * This sample demonstrates the use of the GETPROTOBYNUMBER
 * socket command.
 */
if perror(socket("INITIALIZE","MYSET01"),"INITIALIZE") = 0,
then do
  src = socket("GETPROTOBYNUMBER","6");
  Say "The name assigned to protocol 6 is "WORD(src,2);
end;
src = perror(socket("TERMINATE","MYSET01"),"TERMINATE");
exit 0;

/* This routine returns -1 if the first word if arg 1 is not zero */
perror: if word(arg(1),1) = 0 then return 0; else
  Say arg(2) "Error : "arg(1);
  return -1;
```

Figure 152. GETPROTOBYNUMBER command example

GETSERVBYNAME

Use the GETSERVBYNAME command to retrieve a service and port number.

Format

➡ SOCKET — (— "GETSERVBYNAME" — , — *servicename* — { — , — TCP — } — { — , — *protocolname* — } —) ➡

Parameters

servicename

The service name.

protocolname

The name of a network protocol, for example, TCP or UDP. By default, the *protocolname* parameter is set to TCP.

Returned value

This command returns a string containing the return code, service name, the port number that the service is using, and the network protocol, for example, 0 FTP 21 TCP. The return code can be 0 or the REXX API error number. The return code 0 indicates that the requested socket command was completed successfully.

See [Appendix B, “Socket call error return codes,”](#) on page 743 for additional information about the numeric error codes that are returned by this command.

The following REXX socket API error numbers can be returned:

- 2001 EINVALIDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE

LE C/C++ equivalent

```
struct servent *getservbyname(char *name, char *proto);
```

Code example

```
/* REXX EZARXR17 */
/*
 * This sample demonstrates the use of the GETSERVBYNAME
 * socket command.
 */
if perror(socket("INITIALIZE","MYSET01"),"INITIALIZE") = 0,
then do
  src = socket("GETSERVBYNAME","FTP");
  Say "The FTP service is assigned "SUBWORD(src,2);
end;
src = perror(socket("TERMINATE","MYSET01"),"TERMINATE");
exit 0;

/* This routine returns -1 if the first word if arg 1 is not zero */
perror: if word(arg(1),1) = 0 then return 0; else
  Say arg(2) "Error : "arg(1);
  return -1;
```

Figure 153. GETSERVBYNAME command example

GETSERVBYPORT

Use the GETSERVBYPORT command to translate a port number to the name of the service that is using the port.

Format

➡ SOCKET — (— "GETSERVBYPORT" — , — *portid* — { — , — TCP — } — { — , — *protocolname* — } —) — ➡

Parameters

portid

The port number of the service

protocolname

The name of a network protocol, for example, TCP or UDP. By default, the *protocolname* parameter is set to TCP.

Returned value

This command returns a string that contains the return code, service name, the port number that the service is using, and the network protocol, for example, 0 FTP 21 TCP. The return code can be 0 or the REXX API error number. The return code 0 indicates that the requested socket command was completed successfully.

See [Appendix B, “Socket call error return codes,”](#) on page 743 for additional information about the numeric error codes that are returned by this command.

The following REXX socket API error numbers can be returned:

- 2001 EINVALIDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE

LE C/C++ equivalent

```
struct servent *getservbyport(int port, char *proto);
```

Code example

```
/* REXX EZARXR18 */
/*
 * This sample demonstrates the use of the GETSERVBYPOR
 * socket command.
 */
if perror(socket("INITIALIZE","MYSET01"),"INITIALIZE") = 0,
then do
  src = socket("GETSERVBYPOR","21");
  Say "Port 21 is using service "SUBWORD(src,2);
end;
src = perror(socket("TERMINATE","MYSET01"),"TERMINATE");
exit 0;

/* This routine returns -1 if the first word if arg 1 is not zero */
perror: if word(arg(1),1) = 0 then return 0; else
  Say arg(2) "Error : "arg(1);
  return -1;
```

Figure 154. GETSERVBYPOR command example

GETSOCKNAME

Use the GETSOCKNAME command to retrieve the name of a bound socket.

Stream sockets are not assigned a name until after a successful BIND, CONNECT, or ACCEPT command is completed.

Tip: Use this command to discover the port number that is assigned to a socket after the socket has been implicitly bound, for example, after a CONNECT command has been completed.

Format

➤ SOCKET — (— "GETSOCKNAME" — , — *socketid* —) ➤

Parameters

socketid

The socket descriptor

Returned value

This command returns a string that contains the return code and the NAME string of the bound socket, for example, 0 AF_INET6 7 0 2001:197:11:103::1 0. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

For information about the format of the NAME string, see [“How structures are represented” on page 580](#). See [Appendix B, “Socket call error return codes,” on page 743](#) for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 9 EBADF
- 38 ENOTSOCK
- 45 EOPNOTSUPP
- 57 ENOTCONN

The following REXX socket API error numbers can be returned:

- 2001 EINVALIDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE
- 2009 ESOCKETNOTDEFINED
- 2012 EINVALIDNAME

LE C/C++ equivalent

```
int getsockname(int socket, struct sockaddr *name, int *namelen);
```


Code example

```

/* REXX EZARXR19 */
/*
 * This sample demonstrates the use of the GETSOCKNAME
 * socket command. After the remote peer is obtained
 * send is echoed to the remote server.
 *
 * GUIDELINE: It is generally recommended that a program loop around
 *             the RECV command to ensure that all data is read off
 *             the socket. This sample does not follow the guideline.
 */
src = socket("INITIALIZE","MYSET01");
if perror(src,"INITIALIZE") = 0 then do
  src = socket("SOCKET","AF_INET","STREAM");
  if perror(src,"SOCKET") = 0 then do
    l_socketid = WORD(src,2);
    l_RMTname = "AF_INET 7 127.0.0.1";
    src = socket("CONNECT",l_socketid,l_RMTname);
    if perror(src,"CONNECT") = 0 then do
      src = socket("GETSOCKNAME",l_socketid);
      if perror(src,"GETSOCKNAME") = 0 then do
        l_LOCname = SUBWORD(src,2);
        Say "The local socket name is: "l_LOCname;
      end;
      src = socket("SEND",l_socketid,"*****");
      if perror(src,"SEND") = 0 then do
        src = socket("RECV",l_socketid);
        if perror(src,"RECV") = 0 then
          Say "Echoed data: " word(src,3);
        end;
      end;
    end;
  end;
end;
src = perror(socket("TERMINATE","MYSET01"),"TERMINATE");
exit 0;

/* This routine returns -1 if the first word of arg 1 is not zero */
perror: if word(arg(1),1) = 0 then return 0; else
  Say arg(2) "Error : "arg(1);
  return -1;

```

Figure 155. GETSOCKNAME command example

GETSOCKOPT

Use the GETSOCKOPT command to retrieve the active socket options that were set by the SETSOCKOPT command.

You can specify multiple options with this command; however, at least one option is required.

Format

➡ SOCKET — (— "GETSOCKOPT" — , — *socketid* — , — *levelname* — , — *optname* —) ➡



Parameters

socketid

The local socket descriptor.

levelname

The protocol level. The following protocol levels are supported:

IPPROTO_TCP

Retrieve the socket options that are set at the TCP layer

IPPROTO_IP

Retrieve the IPv4 socket options that are set at the IP layer

IPPROTO_IPV6

Retrieve the IPv6 socket options that are set at the IP layer

SOL_SOCKET

Retrieve the socket options that are set at the socket layer

optname

The option to be retrieved. The options that can be retrieved depend on the value of the *levelname* parameter.

The following rules apply:

- Options that begin with SO_ require the SOL_SOCKET protocol level.
- Options that begin with TCP_ require the IPPROTO_TCP protocol level.
- Options that begin with IP_ require the IPPROTO_IP protocol level.
- Options that begin with IPV6_ require the IPPROTO_IPV6 protocol level.

The following values are supported for the *optname* parameter:

IP_MULTICAST_IF

(IPv4-only) Retrieves the IPv4 interface address that is used to send outbound multicast datagrams from the socket application. When you specify this option, the GETSOCKOPT command returns a string that contains the return code and interface address, for example, 0 10.11.103.1.

IP_MULTICAST_LOOP

(IPv4-only) Determines what occurs when datagrams are sent to a group to which the sending host belongs. If this option is enabled, the IP layer loops back a copy of multicast datagrams for local delivery. By default, this option is enabled. When you specify this option, the GETSOCKOPT command returns either 1 (enabled) or 0 (disabled).

IP_MULTICAST_TTL

(IPv4-only) Retrieves the IP time-to-live of outgoing multicast datagrams. By default, this option is set to the binary value '01'x, which means that multicast is available to the local subnet only. When you specify this option, the GETSOCKOPT command returns a string that contains the return code and a value in the range 0-255, for example, 0 227.

IPV6_ADDR_PREFERENCES

(AF_INET6 only) Retrieves the IPv6 address preferences to be used when selecting the source address. The IPV6_ADDR_PREFERENCES flags that are returned can be one or more of the following values:

IPV6_PREFER_SRC_HOME

A home IPv6 address is preferred over a care-of IPv6 address.

IPV6_PREFER_SRC_COA

A care-of IPv6 address is preferred over a home IPv6 address.

IPV6_PREFER_SRC_TMP

A temporary IPv6 address is preferred over a public IPv6 address.

IPV6_PREFER_SRC_PUBLIC

A public IPv6 address is preferred over a temporary IPv6 address.

IPV6_PREFER_SRC_CGA

A cryptographically generated IPv6 address is preferred over a non-cryptographically generated IPv6 address.

IPV6_PREFER_SRC_NONCGA

A non-cryptographically generated IPv6 address is preferred over a cryptographically generated IPv6 address.

IPV6_MULTICAST_HOPS

(IPv6-only) Retrieves the hop limit used for outgoing multicast packets. When you specify this option, the GETSOCKOPT command returns a string that contains the return code and the hop limit, which is a number in the range 0-255.

IPV6_MULTICAST_IF

(IPv6-only) Retrieves the index of the IPv6 interface that is used to send outbound multicast datagrams from the socket application. When you specify this option, the GETSOCKOPT command returns a string that contains the return code and the IPv6 interface index, for example, 0 1523.

IPV6_MULTICAST_LOOP

(IPv6-only) Determines what occurs when datagrams are sent to a group to which the sending host belongs. If this option is enabled, the IP layer loops back a copy of multicast datagrams for local delivery. By default, this option is enabled. When you specify this option, the GETSOCKOPT command returns either 1 (enabled) or 0 (disabled).

IPV6_UNICAST_HOPS

(IPv6-only) Retrieves the hop limit used for outgoing unicast IPv6 packets. When you specify this option, the GETSOCKOPT command returns a string that contains the return code and the hop limit, which is a number in the range 0-255.

IPV6_V6ONLY

(IPv6-only) Determines whether the socket is restricted to sending or receiving IPv6 packets only. By default, a socket is not restricted. When you specify this option, the GETSOCKOPT command returns a string that contains the return code and a number: 1 (enabled) or 0 (disabled).

SO_ASCII

(REXX only) Determines whether all incoming data is translated from ASCII to EBCDIC, and whether all outgoing data is translated from EBCDIC to ASCII. This option returns a string that contains the error code and either ON (enabled) or OFF (disabled). If the option is enabled, the name of the translation table is returned also. The following string is an example of what might be returned: 0 ON MYTRANTB.

The translation tables are searched in the following order:

1. *user_prefix.subtaskid*.TCPXLBIN
2. *user_prefix.userid*.TCPXLBIN
3. *system_prefix*.STANDARD.TCPXLBIN
4. *system_prefix*.RXSOCKET.TCPXLBIN
5. Internal tables

The following descriptions apply:

- The *user_prefix* value is either the user ID or the job name of the REXX program.
- The *system_prefix* value is either TCPIP or the DATASETPREFIX value from the *hlq*.TCPIP.DATA. You can change the *system_prefix* value to match your site convention.
- The *subtaskid* value is the name of the socket set.
- The *userid* value is the user ID under which the REXX EXEC is running.

When the internal tables are used, the data is converted in the following way:

ASCII to EBCDIC		second hex digit of byte of ASCII data															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
first hex digit of byte of ASCII data	0	00	01	02	03	37	2D	2E	2F	16	05	25	0B	0C	0D	0E	0F
	1	10	11	12	13	3C	3D	32	26	18	19	3F	27	1C	1D	1E	1F
	2	40	4F	7F	7B	5B	6C	50	7D	4D	5D	5C	4E	6B	60	4B	61
	3	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	7A	5E	4C	7E	6E	6F
	4	7C	C1	C2	C3	C4	C5	C6	C7	C8	C9	D1	D2	D3	D4	D5	D6
	5	D7	D8	D9	E2	E3	E4	E5	E6	E7	E8	E9	4A	E0	5A	5F	6D
	6	79	81	82	83	84	85	86	87	88	89	91	92	93	94	95	96
	7	97	98	99	A2	A3	A4	A5	A6	A7	A8	A9	C0	6A	D0	A1	07
	8	20	21	22	23	24	15	06	17	28	29	2A	2B	2C	09	0A	1B
	9	30	31	1A	33	34	35	36	08	38	39	3A	3B	04	14	3E	E1
	A	41	42	43	44	45	46	47	48	49	51	52	53	54	55	56	57
	B	58	59	62	63	64	65	66	67	68	69	70	71	72	73	74	75
	C	76	77	78	80	8A	8B	8C	8D	8E	8F	90	9A	9B	9C	9D	9E
	D	9F	A0	AA	AB	AC	AD	AE	AF	B0	B1	B2	B3	B4	B5	B6	B7
	E	B8	B9	BA	BB	BC	BD	BE	BF	CA	CB	CC	CD	CE	CF	DA	DB
	F	DC	DD	DE	DF	EA	EB	EC	ED	EE	EF	FA	FB	FC	FD	FE	FF

Figure 156. ASCII to EBCDIC

EBCDIC to ASCII		second hex digit of byte of EBCDIC data															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
first hex digit of byte of EBCDIC data	0	00	01	02	03	9C	09	86	7F	97	8D	8E	0B	0C	0D	0E	0F
	1	10	11	12	13	9D	85	08	87	18	19	92	8F	1C	1D	1E	1F
	2	80	81	82	83	84	0A	17	1B	88	89	8A	8B	8C	05	06	07
	3	90	91	16	93	94	95	96	04	98	99	9A	9B	14	15	9E	1A
	4	20	A0	A1	A2	A3	A4	A5	A6	A7	A8	5B	2E	3C	28	2B	21
	5	26	A9	AA	AB	AC	AD	AE	AF	B0	B1	5D	24	2A	29	3B	5E
	6	2D	2F	B2	B3	B4	B5	B6	B7	B8	B9	7C	2C	25	5F	3E	3F
	7	BA	BB	BC	BD	BE	BF	C0	C1	C2	60	3A	23	40	27	3D	22
	8	C3	61	62	63	64	65	66	67	68	69	C4	C5	C6	C7	C8	C9
	9	CA	6A	6B	6C	6D	6E	6F	70	71	72	CB	CC	CD	CE	CF	D0
	A	D1	7E	73	74	75	76	77	78	79	7A	D2	D3	D4	D5	D6	D7
	B	D8	D9	DA	DB	DC	DD	DE	DF	E0	E1	E2	E3	E4	E5	E6	E7
	C	7B	41	42	43	44	45	46	47	48	49	E8	E9	EA	EB	EC	ED
	D	7D	4A	4B	4C	4D	4E	4F	50	51	52	EE	EF	F0	F1	F2	F3
	E	5C	9F	53	54	55	56	57	58	59	5A	F4	F5	F6	F7	F8	F9
	F	30	31	32	33	34	35	36	37	38	39	FA	FB	FC	FD	FE	FF

Figure 157. EBCDIC to ASCII

SO_BROADCAST

(REXX only) Determines whether a program can send broadcast messages over the socket to destinations that can receive datagram messages. By default, this option is disabled. When you specify this option, the GETSOCKOPT command returns a string that contains the return code and a number: 1 (enabled) or 0 (disabled).

Restriction: This option has no meaning for stream sockets. It is valid only for datagram sockets.

SO_DEBUG

(REXX only) Determines whether debug information is recorded. By default, this option is disabled. When you specify this option, the GETSOCKOPT command returns a string that contains the return code and a number: 1 (enabled) or 0 (disabled).

Restriction: This option is valid only for stream sockets.

SO_DONTROUTE

Determines whether normal routing determination is bypassed for outgoing packets on the socket. When you specify this option, the GETSOCKOPT command returns a string that contains the return code and a number: 1 (enabled) or 0 (disabled).

Restriction: When a packet is sent, if the local interface cannot be determined from the destination address, the 51 ENETUNREACH error message is returned.

SO_EBCDIC

(REXX only) Determines whether data is translated to and from EBCDIC. This option is ignored by EBCDIC hosts. When you specify this option, the GETSOCKOPT command returns a string that contains the return code and either ON (enabled) or OFF (disabled). If the option is enabled, the name of the translation table is returned also.

Restriction: This option has no effect on the data that is processed by the socket library.

SO_ERROR

Retrieves information about pending errors on the socket or other errors that are not explicitly returned by any socket commands. The error status is cleared after each call. When you specify this option, the GETSOCKOPT command returns a string that contains the return code and the most recent error, for example, 0 36.

SO_KEEPAIVE

Determines if the keep alive mechanism periodically sends a packet on an otherwise idle connection for a stream socket. When enabled, if the remote TCP/IP does not respond to the packet or to retransmissions of the packet, the connection is terminated with the ETIMEDOUT error. By default, this option is disabled. When you specify this option, the GETSOCKOPT command returns a string that contains the return code and a number: 1 (enabled) or 0 (disabled).

Tip: The site administrator can enable the global keep-alive mechanism by specifying the INTERVAL parameter on the TCPCONFIG statement in the TCP/IP stack profile data set, TCPIP.PROFILE.

SO_LINGER

Determines how TCP/IP processes data that has not been transmitted when the CLOSE command is issued for the socket. When this option is enabled and the CLOSE command is issued, the calling program is blocked until either the data is successfully transmitted or the connection times out. When this option is disabled and the CLOSE command is issued, the CLOSE command returns without blocking the caller; then TCP/IP continues to attempt to send data for a specified time, which usually provides sufficient time to complete the data transfer. By default, this option is disabled. When you specify this option, the GETSOCKOPT command returns a string that contains the return code and either 1 (enabled) or 0 (disabled), for example, 0 1.

Restrictions:

- Using the SO_LINGER option does not guarantee that a data transfer will be completed, because TCP/IP waits only for the amount of time that is specified by SETSOCKOPT command.
- This option is valid only for stream sockets.

SO_OOBINLINE

Determines whether out-of-band data is available to the RECV or RECVFROM commands. When this option is enabled, out-of-band data is placed in the normal data input queue as it is received; this data is then available to RECV or RECVFROM commands, even if the OOB flag is not set. When this option is disabled, out-of-band data is placed in the priority data input queue as it is received; this data is then available to RECV or RECVFROM commands only if the OOB flag is set. When you specify this option, the GETSOCKOPT command returns a string that contains the return code and a number: 1 (enabled) or 0 (disabled).

Restriction: This option is valid only for stream sockets.

SO_RCVBUF

Retrieves the size of the data portion of the TCP/IP receive buffer. The size of the receive buffer is protocol specific and is based on the following values:

- (TCP socket) The TCPRCVBufrsize keyword on the TCPCONFIG statement in the PROFILE.TCPIP data set.
- (UDP socket) The UDPRCVBufrsize keyword on the UDPCONFIG statement in the PROFILE.TCPIP data set.
- (Raw socket) The default size of 65535.

When you specify this option, the GETSOCKOPT command returns a string that contains the return code and either the size of the receive buffer or 0 (disabled).

SO_RCVTIMEO

Reports the timeout value for receive-type functions. This option returns a string that contains the number of seconds followed by the number of microseconds. These values specify the length of time to wait for a receive-type function to complete. Returns a value in the range 0 – 2 678 400 (equal to 31 days) for the number of seconds. Returns a value in the range 0 – 1 000 000 (equal to 1 second) for the number of microseconds. If a receive-type function has blocked for this length of time without receiving data, it returns with an errno set to EWOULDBLOCK. The value 0 (the default) indicates that a receive-type function does not time out.

The following receive-type commands are supported:

- READ
- RECV
- RECVFROM

SO_REUSEADDR

Determines whether local addresses are reused. Enabling this option alters the normal algorithm that is used with the BIND command. The normal BIND algorithm permits each Internet address and port combination to be bound only once. If the address and port already have been bound, a subsequent BIND command fails with the 48 EADDRINUSE error message . When this option is enabled, the following situations are supported:

- A server can bind the same port multiple times. Each invocation either must use a different local IP address, or it must use a wildcard address (INADDR_ANY or in6addr_any) only one time for each port.
- A server with active client connections can be restarted and can bind to its port without having to close all of the client connections.
- For datagram sockets, multicasting is supported so that multiple BIND commands can be made to the same class D address and port number.

By default, this option is disabled. When you specify this option, the GETSOCKOPT command returns a string that contains the return code and a number: 1 (enabled) or 0 (disabled).

Tip: If you want multiple servers that bind to INADDR_ANY or IN6ADDR_ANY to listen on the same port number, use the SHAREPORT option on the PORT statement in TCPIP.PROFILE.

SO_SNDBUF

Determines the size of the data portion of the TCP/IP send buffer. The size of the send buffer is protocol specific and is based on the following values:

- (TCP socket) The TCPSENBufrsize keyword on the TCPCONFIG statement in the PROFILE.TCPIP data set.
- (UDP socket) The UDPSENBufrsize keyword on the UDPCONFIG statement in the PROFILE.TCPIP data set.
- (Raw socket) The default size of 65535.

When you specify this option, the GETSOCKOPT command returns a string that contains the return code and the size of the send buffer.

SO_SNDTIMEO

Reports the timeout value for send-type functions. This option returns a string that contains the number of seconds followed by the number of microseconds. These values specify the length of time to wait for a send-type function to complete. Returns a value in the range 0 – 2 678 400 (equal to 31 days) for the number of seconds. Returns a value in the range 0 – 1 000 000 (equal to 1 second) for the number of microseconds. If a send-type function has blocked for this length of time without receiving data, it returns with an errno set to EWOULDBLOCK. The value 0 (the default) indicates that a send-type function does not time out.

The following send-type commands are supported:

- SEND
- SENDTO
- WRITE

SO_TYPE

Retrieves the socket type. When you specify this option, the GETSOCKOPT command returns a string that contains the return code and a number: 1 (SOCK_STREAM), 2 (SOCK_DGRAM), or 3 (SOCK_RAW).

TCP_KEEPAIVE

Determines whether a socket-specific timeout value (in seconds) is used instead of a configuration-specific value, when keep alive timing is active for the socket. When enabled, the socket-specific timeout value remains in effect until either the socket is closed or it is reset by a SETSOCKOPT command. When you specify this option, the GETSOCKOPT command returns a string that contains the return code and either the timeout value or 0 (disabled). For more information about the socket option parameters, see [TCP_KeepAlive socket option in z/OS Communications Server: IP Programmer's Guide and Reference](#).

Tip: The site administrator can enable the global keep-alive mechanism by specifying the INTERVAL parameter on the TCPCONFIG statement in the TCP/IP stack profile data set, TCPIP.PROFILE.

TCP_NODELAY

Determines whether the data that is sent over the socket is subject to the Nagle algorithm (RFC 896). When this option is enabled, TCP waits to send small amounts of data until the acknowledgment for the previous data sent is received. When this option is disabled, TCP sends data when it is presented. When you specify this option, the GETSOCKOPT command returns a string that contains the return code and a number: 1 (enabled) or 0 (disabled).

TCP_MAXSEG

(IPPROTO_TCP protocol only) Retrieves the maximum segment size for a TCP send. When you specify this option, the GETSOCKOPT command returns a string that contains the return code and either the maximum segment size or 0 (disabled).

Returned value

The command returns a string that contains the return code and the option value. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

See [Appendix B, “Socket call error return codes,” on page 743](#) for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 9 EBADF
- 22 ENOTSOCK
- 38 ENOTSOCK
- 42 ENOPROTOOPT
- 45 EOPNOTSUPP

The following REXX socket API error numbers can be returned:

- 2001 EINVALDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE
- 2009 ESOCKETNOTDEFINED
- 2012 EINVALIDNAME

LE C/C++ equivalent

```
int getsockopt(int socket, int level, int option_name, char
               *option_value, int *option_len);
```


Code example

```
/* REXX EZARXR20 */
/*
 * This sample demonstrates the use of the GETSOCKOPT
 * and SETSOCKOPT commands.
 *
 * The program opens a STREAM socket and connects to port 7
 * (echo server) using the loop back address. Before sending
 * data the program issues the SETSOCKOPT command to set the
 * sockets send buffer to 32000 bytes. Data is then sent to
 * and received. After the data is received the GETSOCKOPT
 *
 * GUIDELINE: It is generally recommended that a program loop around
 *             the RECV command to ensure that all data is read off
 *             the socket. This sample does not follow the guideline.
 */
src = socket("INITIALIZE","MYSET01");
if perror(src,"INITIALIZE") \= 0 then signal ENDPROGRAM;
src = socket("SOCKET","AF_INET","STREAM");
if perror(src,"SOCKET") = 0 then do
  l_socketid = WORD(src,2);
  src = socket("GETSOCKOPT",l_socketid,"SOL_SOCKET",,
    "SO_SNDBUF");
  if perror(src,"GETSOCKOPT") = 0 then do
    Say "Current socket send buffer size is",
      word(src,2);
    if word(src,2) < 32000 then do
      Say "Increasing the socket send buffer size",
        "to 32000";
      src = socket("SETSOCKOPT",l_socketid,,
        "SOL_SOCKET","SO_SNDBUF",32000);
      /* *****
       * Data can be sent even if command fails
       * so just post an message if an error occurs.
       * *****/
      src = perror(src,"SETSOCKOPT");
    end;
    l_RMTname = "AF_INET 7 127.0.0.1";
    src = socket("CONNECT",l_socketid,l_RMTname);
    if perror(src,"CONNECT") = 0 then do
      src = socket("SEND",l_socketid,"*****");
      if perror(src,"RECV","SEND") = 0 then do
        src = socket("RECV",l_socketid);
        if perror(src,"RECV") = 0 then
          Say "Echoed data: " word(src,3);
      end;
    end;
    src = perror(socket("CLOSE",l_socketid),"CLOSE");
  end;
end;
ENDPROGRAM:
src = perror(socket("TERMINATE","MYSET01"),"CLOSE");
exit 0;

/* This routine returns -1 if the first word of arg 1 is not zero */
perror: if word(arg(1),1) = 0 then return 0; else
  Say arg(2) "Error : "arg(1);
  return -1;
```

Figure 158. GETSOCKOPT command example

GIVESOCKET

Use the GIVESOCKET command to transfer a socket descriptor to another application that is running on the same host.

The other application can use the TAKESOCKET command to take the socket. If the socket is closed before an application can take it, then the socket is reset. Any stream socket can be given.

Restriction: A socket given by the GIVESOCKET command can be taken only by a process that issues the TAKESOCKET command with the same address family.

Guideline: The application that issues the GIVESOCKET command can use the SELECT command to test for a socket exception condition. The exception condition indicates that the socket has been successfully taken by another application. After the socket has been successfully taken, the application that issued the GIVESOCKET command can safely close the socket. For example, when a server accepts a new connection, the server issues the GIVESOCKET command and then monitors the socket for an exception condition using the SELECT command. The server subtask issues the TAKESOCKET command. After the subtask takes the socket, the server issues the CLOSE command and waits for a new connection request.

Format

➔ SOCKET — (— "GIVESOCKET" — , — *socketid* — , — *clientid* —) ➔

Parameters

socketid

The socket descriptor.

clientid

The client ID of the application. The *clientid* parameter uses the following format:

```
"domain jobname subtaskid"
```

where:

domain

The address family of the socket. The following domains are supported:

- AF_INET or 2
- AF_INET6 or 19

jobname

This optional field specifies the name of the address space that will issue the TAKESOCKET command. If this field is not specified, any address space can take the socket.

Guideline: To prevent possible security exposures, always specify the *jobname* field.

subtaskid

This optional field specifies the name of the active socket set. It can be specified only if the *jobname* field is specified also. If the *subtaskid* field is specified but the value is not the name of the active socket set, then the name of the active socket set is used. If the *subtaskid* field is not specified, by default the name of the active socket set is used.

Consider the following situation: The GETCLIENTID command returns the following string: AF_INET RUNC1 TTLSCXXX. If the application issues the GIVESOCKET command using the *clientid* parameter value of "AF_INET RUNC1 BBBB", BBBB is replaced with TTLSCXXX, because TTLSCXXX is the name of the active socket set.

Returned value

The command returns a string that contains the return code. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

See [Appendix B, "Socket call error return codes," on page 743](#) for additional information about the numeric error codes that are returned by this command.

The following REXX socket API error number can be returned:

- 9 EBADF

The following REXX socket API error numbers can be returned:

- 2001 EINVALDRXSOCKETCALL

- 2005 ESUBTASKNOTACTIVE
- 2009 ESOCKETNOTDEFINED
- 2012 EINVALIDNAME

LE C/C++ equivalent

```
int givesocket(int d, struct clientid *clientid);
```

Code example

See the EZARXS04 REXX sample in the SEZAINST file for an example of using the GIVESOCKET command.

INET6ISSRCADDR

Use the INET6ISSRCADDR command to indicate whether an input IPV6 socket address matches an address that is defined to the stack, which conforms to one or more input IPV6_ADDR_PREFERENCES flags.

A client or server program can use the INET6ISSRCADDR command when it strictly requires the IP addresses that it uses to have one or more specific attributes before allowing network activity. Such a program can perform the following actions:

1. (Optional) Issue the SETSOCKOPT macro with option IPV6_ADDR_PREFERENCES to set source IP address selection preferences for the socket.
2. Issue the BIND2ADDRSEL command to bind a socket to a local IP address. The BIND2ADDRSEL command attempts to assign a local IP address that matches the IPV6_ADDR_PREFERENCES flags that are specified with the SETSOCKOPT command, but does not guarantee that the local IP address that it selects satisfies all preferences.
3. Issue the GETSOCKNAME command to obtain the local IP address that is bound to the socket.
4. Issue the INET6ISSRCADDR command to verify whether the local IP address has the correct attributes.

The command returns a string that contains the return code and the result. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully and indicates one of the following results:

TRUE

The IP address is an address that is known to the TCP/IP stack and satisfies the preference flags.

FALSE

The IP address is an address known to the TCP/IP stack but does not satisfy the preference flags.

You can specify more than one IPV6_ADDR_PREFERENCES flags on this command. All flags must be satisfied for the result to be TRUE. Some IPV6_ADDR_PREFERENCES flags are contradictory. If you specify contradictory flags on a single command invocation, the result is FALSE.

Format

► SOCKET — (— "INET6ISSRCADDR" — , — *name* — , — *flags* —) ►

Parameters

name

The socket address that is to be tested.

name = "domain portid flowinfo ipaddress scopeid"

where:

- The *domain* value must be the decimal number 19 or AF_INET6.
- The *portid* value is not used by the INET6ISSRCADDR command.

Requirement: You must specify a decimal numeral between 0 and 65535 for the *portid* value.

- The *flowinfo* value is not used by the INET6ISSRCADDR command.

Requirement: You must specify a decimal numeral between 0 and 2147483647 for the *flowinfo* value.

- The *ipaddress* value is the IP address to be tested.

Rule: You can specify an IPv6 address or an IPv4-mapped IPv6 address.

- The *scopeid* value identifies the interfaces that are applicable for the scope of the address that is specified in the *ipaddress* field. For a link-local IP address, the *scopeid* field can specify a link index, which identifies a set of interfaces. For all other scopes, the *scopeid* field must be set to 0.

flags

A space-delimited string that contains one or more of the following values:

IPV6_PREFER_SRC_HOME

Tests whether the IP address is a home address. Any valid address that is known to the TCP/IP stack satisfies this flag.

IPV6_PREFER_SRC_COA

Tests whether the IP address is a care-of address. This type of address is not supported and the command always returns FALSE if this flag is specified.

IPV6_PREFER_SRC_PUBLIC

Tests whether the IP address is a public address.

IPV6_PREFER_SRC_TMP

Tests whether the IP address is a temporary address.

IPV6_PREFER_SRC_CGA

Tests whether the IP address is a cryptographically generated address. This type of address is not supported and the command always returns FALSE if this flag is specified.

IPV6_PREFER_SRC_NONCGA

Tests whether the IP address is not cryptographically generated. Any valid address that is known to the TCP/IP stack satisfies this flag.

Any *flags* specification that contains values other than 0 or the options listed above causes an EINVALDRXSOCKETCALL error to be returned.

Tip: Some of these flags are contradictory. For example:

- The flag IPV6_PREFER_SRC_HOME contradicts the flag IPV6_PREFER_SRC_COA.
- The flag IPV6_PREFER_SRC_CGA contradicts the flag IPV6_PREFER_SRC_NONCGA.
- The flag IPV6_PREFER_SRC_TMP contradicts the flags IPV6_PREFER_SRC_PUBLIC.

Result: If you specify contradictory flags on the macro, the result is FALSE.

Returned value

The command returns a string that contains the return code and the result. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully. The result is TRUE if the IP address is an address that is known to the TCP/IP stack and satisfies the preference flags, or FALSE if the IP address is an address that is known to the TCP/IP stack but does not satisfy the preference flags.

See Appendix B, “Socket call error return codes,” on page 743 for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 1 EPERM
- 22 EINVAL
- 47 EAFNOSUPPORT
- 49 EADDRNOTAVAIL

The following REXX socket API error numbers can be returned:

- 2001 EINVALIDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE
- 006 ESOCKETNOTALLOCATED

LE C/C++ equivalent

```
short inet6_is_srcaddr(struct sockaddr_in6 *srcaddr, uint32_t flags);
```

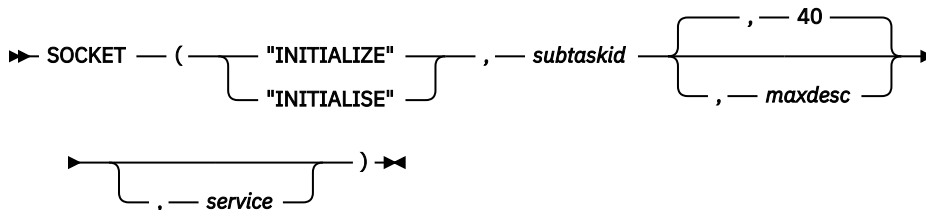
INITIALIZE

Use the INITIALIZE command to preallocate a socket set.

A socket set is a number of preallocated sockets available to a single REXX application. You can define multiple socket sets for one session, but only one socket set can be active at a time. The INITIALIZE command must be issued before any socket services are requested.

Guideline: When multiple socket sets are initialized, the last socket set that is initialized becomes the active socket set. Before closing a socket belonging to a specific socket set, you must ensure that the socket set that owns the socket is the active set. Otherwise, the CLOSE command fails.

Format



Parameters

subtaskid

The name of the socket set. This parameter can be 1-8 characters in length; it cannot contain blanks.

maxdesc

The number of sockets descriptors that can be opened in this socket set. By default, this parameter is set to 40.

service

The name of the TCP/IP service. This is the job name of an active TCP/IP stack. The name of the TCP/IP service must match the name of an active stack.

Result: If service is not specified on the INITIALIZE command, it defaults to the TCPIPJOBNAME that is specified in the Resolver input used for this application. If TCPIPJOBNAME is not specified either, it defaults to TCPIP.

Restrictions:

- In a CINET environment, if the service name does not match the name of an active TCP/IP stack, the INITIALIZE command fails with the 1004 EIBMIUCVERR return code.

- In an INET environment, any service name is accepted. However, the service name is changed to *INET if the service name does not match the name INET that is the TYPE operand specified on the FILESTYPE statement, which is defined in the BPXPARMxx PARMLIB member. The *INET service name indicates to the caller that a UNIX INET environment exists.

Returned value

The command returns a string that contains the return code, the name of the socket set, the number of socket descriptors that can be opened, and the name of the TCP/IP services, for example, 0 MYTASK 40 TCPSVT. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

For information about the format of the NAME string, see [“How structures are represented” on page 580](#). See Appendix B, [“Socket call error return codes,” on page 743](#) for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 9 EBADF
- 22 EINVAL
- 38 ENOTSOCK
- 45 EOPNOTSUPP
- 1004 EIBMIUCVERR

The following REXX socket API error numbers can be returned:

- 2001 EINVALIDRXSOCKETCALL
- 2003 ESUBTASKINVALID
- 2004 ESUBTASKALREADYACTIVE2012 EINVALIDNAME

Tip: If the 2003 ESUBTASKINVALID error code is returned, issue the TERMINATE command and then reissue the INITIALIZE command. If the command continues to fail, verify that the value of *service* parameter is applicable for the system.

LE C/C++ equivalent

This command has no LE C/C++ equivalent.

Code example

The example code described in [“GETADDRINFO” on page 597](#) can be used to experiment with the INITIALIZE command in a CINET environment.

IOCTL

Use the IOCTL command to perform control functions on sockets.

Format

```
➔ SOCKET — ( — "IOCTL" — , — socketid — , — icmd —————— ) ——————
                                     , — ivalue ——————

      ——————
      , — idata —————— ) ➔
```

Parameters

socketid

The socket descriptor.

icmd

The control function. The following commands are supported:

FIONBIO

Enables or disables blocking mode. When the *ivalue* parameter is set to ON, the socket is set to nonblocking mode. When the *ivalue* parameter is set to OFF, the socket is in blocking mode. The command returns a string that contains the return code.

FIONREAD

Requests the number of bytes on the receive queue that are ready for reading. The command returns a string that contains the return code and the number of bytes that are ready for reading.

SIOCATMARK

Determines whether the current location in the input data points to out-of-band data. The command returns a string that contains the return code and YES or NO. YES indicates that there is out-of-band data.

SIOCGIFADDR

Obtains a network interface address. The *ivalue* parameter must specify the name of the network interface, for example, LOGETH13. The network interface name can be 1-16 characters in length. The command returns a string that contains the return code and the network interface address, which has the following format: *interface domain port ipaddress*.

Restriction: This function is valid only for IPv4 interfaces.

SIOCGIFBRDADDR

Obtains the network broadcast address of an interface. The *ivalue* parameter must specify the name of the network interface, for example, LOGETH13. The network interface name can be 1-16 characters in length. The command returns a string that contains the return code and the network broadcast address, which has the following format: *Hinterface domain port ipaddress*.

Restriction: This function is valid only for IPv4 interfaces.

SIOCGIFCONF

Obtains the list of the interfaces that are defined to the system. The *ivalue* parameter must specify the maximum number of interfaces to be returned. The command returns a string that contains the return code and a list of interfaces, which has the following format: *interface domain port ipaddress*

Restriction: Only IPv4 interface information is returned.

Tip: You can parse the information in the following way:

```
Drop st.
l_retcode = socket("IOCTL",l_sockid,"SIOCGIFCONF",10);
parse var l_retcode l_rc l_iflist
i=0;
do until l_iflist = ""
i=i+1;
parse var l_iflist st.i.interface st.i.domain st.i.port,
st.i.address l_iflist
st.i.name = st.i.domain st.i.port st.i.address;
end;
st.0 = i;
```

SIOCGIFDSTADDR

Obtains the network destination address of an interface. The *ivalue* parameter must specify the name of the network interface, for example, LOGETH13. The network interface name can be 1-16 characters in length. The command returns a string that contains the return code and the network destination address, which has the following format: *interface domain port ipaddress*

Restriction: This function is valid only for IPv4 interfaces.

SIOCGIFMTU

Obtains the network MTU (maximum transmission unit) of an interface. The *ivalue* parameter must specify the name of the network interface, for example, LOGETH13. The network interface name can be 1-16 characters in length. The command returns a string that contains the return code, the interface name, and the numeric MTU value.

Restriction: This function is valid only for IPv4 interfaces.

SIOCGIFNAMEINDEX

Obtains a list of all interface names and device indexes that are defined to the system. This list includes loopback addresses, but excludes VIPA addresses. The command returns a string that contains the return code and a list of interfaces, which have the following format: interface_index interface_name.

Tip: You can parse the information in the following way:

```
DROP st.  
l_retcode = socket("IOCTL",1,"SIOCGIFNAMEINDEX");  
i = 0;  
do until l_retcode = ''  
    i=i+1;  
    parse var l_retcode st.i.interfindex st.i.name l_retcode;  
end;
```

SIOCGIFFLAGS

Obtains the network interface flags of an interface. The *ivalue* parameter must specify the link name of the interface, for example, LOGETH13. The command returns a string that contains the return code, the interface name, the flags in four hexadecimal digits, and the symbolic names of the flags.

The following string is an example of what is returned by the SIOCGIFFLAGS function:

```
0 LOOPBACK 0049 IFF_UP IFF_LOOPBACK IFF_RUNNING
```

Restriction: This function is valid only for IPv4 interfaces.

SIOCGIFMETRIC

Obtains the network routing metric of an interface. The *ivalue* parameter must specify the link name of the interface, for example, LOGETH13. The command returns a string that contains the return code, interface name, and routing metric.

Restriction: This function is valid only for IPv4 interfaces.

SIOCGIFNETMASK

Obtains the network mask of an interface. The *ivalue* parameter must specify the interface name, for example, LOGETH13. The command returns a string that contains the return code, interface name, and a socket NAME with the network mask, for example, 0 LOGETH13 AF_INET 0 255.255.240.0.

For more information about the format of the NAME string, see [“How structures are represented”](#) on page 580.

Restriction: This function is valid only for IPv4 interfaces.

SIOCGIPMSFILTER

Obtains a list of the IPv4 source addresses from the stack for a specified interface and multicast group. The *ivalue* parameter must be a space-delimited string that contains an IPv4 multicast address, an IPv4 interface address, and the number of addresses to be returned, for example, "224.224.224.1 176.11.16.103 5". If the number of addresses to be returned is set to 0, all source filters are returned. The command returns a string that contains the return code, filter mode (either MCAST_INCLUDE or MCAST_EXCLUDE), number of IPV4 source addresses that are returned, and the IPv4 source addresses. Depending on the filter mode, source addresses are included or excluded.

The following string is an example of what is returned by the SIOCGIPMSFILTER function:

```
0 MCAST_INCLUDE 2 10.11.103.1 176.11.16.103
```


SIOCGMSFILTER

Obtains a list of the IPv4 or IPv6 source addresses for the specified interface index and multicast group. The *ivalue* parameter must be a space-delimited string that contains an interface index, a socket name for the multicast address, and the number of addresses to be returned, for example, "34 AF_INET 21 224.224.224.1 5". If the number of addresses to be returned is set to 0, all source addresses are returned. The command returns a string that contains the return code, filter mode (either MCAST_INCLUDE or MCAST_EXCLUDE), the number of source addresses that are returned, and a list of the source-socket name groups.

The following string is an example of what is returned by the SIOCGMSFILTER function:

```
0 MCAST_INCLUDE 1 AF_INET 21 176.11.16.103
```

In the example, AF_INET 12 176.11.16.103 is the source-socket name.

For more information about the format of the NAME string, see [“How structures are represented”](#) on page 580 or the BIND or CONNECT commands.

SIOCGPARTNERINFO

Provides an interface for an application to retrieve security information about its partner. The *ivalue* parameter must be a space-delimited string that contains a request type and a timeout value in the range 0 – 60.

The request type is one of the following values:

- PI_REQTYPE_CONNTYPE, CONNTYPE, C, or 0
- PI_REQTYPE_PARTNER_USERID, USERID, U, or 1

The command returns a string that contains the return code and the output data.

The return code is either 0 or -1, with 0 indicating successful completion and -1 indicating that an error occurred.

The output data includes the following information:

- Status

The status indicates the type of information that is returned, which includes connection routing information that can be followed by partner user ID information. If the partner user ID information is not returned, or only the address-space user ID is returned, the appropriate user ID fields are set to 0.

- Connection type (8 hexadecimal digits)
- Length of the returned address-space user ID
- Returned address space user ID
- Length of the returned task-level user ID
- Returned task-level user ID
- Additional error code and the text version of the error (if the return code is -1)

The following string is an example of the output data for a successful command:

```
0 3 0000000E 5 USER1 0 0
```

In this example, 0 is the return code, 3 is the PI_STATUS (indicating that the connection routing information and the partner user ID are returned), 0000000E is the connection routing value (indicating PI_CONNTYPE_SAME_CLUSTER, PI_CONNTYPE_SAME_IMAGE, and PI_CONNTYPE_INTERNAL), 5 is the length of the returned address space user ID, USER1 is the returned address space user ID, and 0 0 indicates that there is no task-level user ID.

If an error occurs, the output data is similar to the following example:

```
-1 1 0000000E 0 0 0 0 ( 73 ETIME Timer expired
```

In this example, -1 is the return code, 1 is the PI_STATUS (indicates that the connection routing information is returned), 0000000E is the connection routing value (indicates PI_CONNTYPE_SAME_CLUSTER, PI_CONNTYPE_SAME_IMAGE, and PI_CONNTYPE_INTERNAL),

0 0 0 0 indicates that no user ID information is returned, 73 is the error code, and ETIME Timer expired is the error description.

For more information about using the SIOCGPARTNERINFO IOCTL, see [z/OS Communications Server: IP Programmer's Guide and Reference](#).

SIOCSAPPLDATA

Associates user-defined data with a socket descriptor. This data can be used to identify socket endpoints for network-management applications or tools such as Netstat or SMF. The *ivalue* parameter is a string that contains the user-defined data; it can be up to 40 bytes in length. The command returns an error code.

Guideline: The content of this field is determined by the application that owns the connection. See application-specific documentation for explanations of the layout, format, and meaning of this field. Typically, the field contains printable EBCDIC characters, although some applications might include binary data.

The application data is displayed by the following items only when the TCP connection has application data associated with it:

- Netstat reports. The information is displayed on the ALL/-A report. If the APPLDATA modifier is used, the information also is displayed on the ALLConn/-a and Conn/-c reports.
- The SMF 119 TCP connection termination record. For more information, see [z/OS Communications Server: IP Programmer's Guide and Reference](#).
- Network management applications. For more information, see [z/OS Communications Server: IP Programmer's Guide and Reference](#).

SIOCSIPMSFILTER

Sets the list of the IPv4 source addresses and the filter mode for an interface and multicast group. The *ivalue* parameter must be a space-delimited string that contains an IPv4 multicast address, an IPv4 interface address, filter mode, the number of IPv4 source addresses to be added, and the list of IPv4 source addresses, for example:

```
224.224.224.1 176.11.16.103 MCAST_INCLUDE 2 48.11.16.103
176.11.16.103
```

The filter mode is one of the following values:

Include

MCAST_INCLUDE, INCLUDE, I, or 0

Exclude

MCAST_EXCLUDE, EXCLUDE, E, or 1

The maximum number of source addresses is 64. If the interface address is 0.0.0.0, then the stack selects the default IPv4-multicast interface address. The command returns a string that contains the return code.

SIOCSMSFILTER

Sets a list of the IPv4 or IPv6 source addresses and the filter mode for an interface and multicast group. The *ivalue* parameter must be a space-delimited string that contains an interface index, the socket address name for the multicast address, the filter mode, the number of source address names, and a list of the source address names, for example:

```
34 AF_INET6 56504 0 FF02::67:69 0 MCAST_INCLUDE 1 2001:10:11:103::1
```

The filter mode is one of the following values:

Include

MCAST_INCLUDE, INCLUDE, I, or 0

Exclude

MCAST_EXCLUDE, EXCLUDE, E, or 1

The maximum number of source addresses is 64. If the interface index is 0, then the stack selects an interface. The command returns a string that contains the return code.

For more information about the format of the NAME string, see [“How structures are represented” on page 580](#).

SIOCSPARTNERINFO

The SIOCSPARTNERINFO IOCTL sets an indicator to retrieve the partner security credentials during connection setup and saves the information. In this way, an application can issue a SIOCGPARTNERINFO IOCTL without suspending the application or at least minimizing the time to retrieve the information. The *ivalue* parameter must be set to PI_REQTYPE_SET_PARTNERDATA, PARTNERDATA, P, or 1. The command returns a string that contains the return code.

For more information about using the SIOCSPARTNERINFO IOCTL, see [z/OS Communications Server: IP Programmer's Guide and Reference](#).

SIOCTTLSCTL

Queries or controls Application Transparent Transport Layer Security (AT-TLS) for a TCP stream socket connection. If the socket is in blocking mode, this function blocks during the initial handshake. If the socket is in nonblocking mode, it returns the 36 EWOULDBLOCK error. See the Application Transparent TLS information in [z/OS Communications Server: IP Programmer's Guide and Reference](#) for more information.

Unless the *ivalue* parameter is set to QUERYONLY, the application must be mapped to an AT-TLS policy and the parameter ApplicationControlled must be set to ON. The *ivalue* parameter can have the following values:

QUERYONLY

Requests security information about the current socket. This request can be issued by any application, regardless of the value of the ApplicationControlled parameter. If the socket is not mapped to an AT-TLS policy and the socket is in a writable state, issuing this command causes AT-TLS to try to locate and assign a policy. The command returns a string that contains status information about the security level of the connection.

QUERYHOST

Queries the partner certificate to validate that the certificate matches the host name. The *idata* parameter must specify the host name. This request returns one of the following values:

0

Host name validation succeeded. The host name in the partner certificate matches the value of the *idata* parameter.

1

Validation status unknown. This value is returned if no partner certificate is present. This can occur for servers if client authentication is not enabled in the policy.

2

Host name validation failed. The host name in the partner certificate did not match the value of the *idata* parameter.

3

Host name validation failed with an unexpected *gsk_validate_hostname* value.

4

Host name validation failed with an unexpected *gsk_decode_certificate* value.

QUERYRULENAME

Queries the TTLSRule name that is mapped to the connection. This request returns the TTLSRule name or *N/A*, if no mapping exists.

QUERYGROUPACTIONNAME

Queries the TTLSGroupAction name that is mapped to the connection. This request returns the TTLSGroupAction name or *N/A*, if no mapping exists.

QUERYENVIRONMENTACTIONNAME

Queries the TLSEnvironmentAction name that is mapped to the connection. This request returns the TLSEnvironmentAction name or *N/A*, if no mapping exists.

QUERYCONNECTIONACTIONNAME

Queries the TLSConnectionAction name that is mapped to the connection. This request returns the TLSConnectionAction name or *N/A*, if no mapping exists.

QUERYSESSIONID

Obtains the session identifier for the SSL session.

QUERYSESSIONTOKEN

Obtains a token for the SSL session. The token represents the AT-TLS environment and session identifier for the secure connection.

INITCONNECTION

Initializes a secure SSL connection using the role that is defined by the Handshake parameter in the mapped policy. When this command is successful, it returns a string that contains status information about the security level of the connection.

INITCONNHSTIMEOUT

Initializes a secure SSL connection using the role that is defined by the Handshake parameter in the mapped policy. The Handshake parameter must be set to Server or ServerWithClientAuth. If the SSL handshake times out before receiving data from the client, SSL is stopped on the connection and the TCP connection remains established. Using this command is equivalent to requesting TTLS_INIT_CONNECTION and TTLS_ALLOW_HSTIMEOUT.

INITCONNECTIONWITHTOKEN

Initializes a secure SSL connection by using the role that is defined by the Handshake parameter in the mapped policy, and sets a token for the SSL session. When this command is issued successfully, it returns a string that contains status information about the security level of the connection.

The *idata* parameter must contain 3 input values delimited by a single space, for example, "0 1 *tokenstring*".

- The first input value determines whether the TCP connection stays alive if the SSL handshake times out. A value of 0 or OFF indicates that the TCP connection does not stay alive. A value of 1 or ON indicates that the TCP connection stays alive.
- The second input value determines whether the application requires that the SSL session ID should be reused. A value of 0 or OFF indicates the session ID is not reused. A value of 1 or ON indicates that the session ID is reused.
- The third input value is the token for the SSL session. It represents the AT-TLS environment and session identifier for the secure connection. The token must be obtained previously by a QUERYSESSIONTOKEN call.

RESETSESSION

For a connection secured with TLSv1.2 or earlier TLS/SSL versions, resets the session ID so that it is not reused by another connection. For a connection secured with TLSv1.3, deletes all session tickets for the session so that they are not reused by another connection. When this command is successful, it returns a string that contains status information about the security level of the connection.

RESETCIPHER

Resets the cipher that is used for the secure session, updating the encryption key. For TLSv1.2 and earlier TLS/SSL versions, a renegotiation is done. If the session ID times out or has been reset, a full handshake is performed. Otherwise, a short handshake is performed.

For TLSv1.3, the encryption key is updated and a Key Update message is sent to the session partner. The message requests that the session partner also update its encryption key.

When this command is successful, it returns a string that contains status information about the security level of the connection.

RESETWRITECIPHER

For a connection secured with TLSv1.3, resets the write cipher used for the secure session, updating the encryption key. The encryption key is updated and a Key Update message is sent to the session partner. The message does not request that the session partner update its encryption key. When this command is successful, it returns a string that contains status information about the security level of the connection.

SENDESSIONTICKET

For a connection secured with TLSv1.3, requests that the server generate and send a session ticket to the client for use resuming the session later. When this command is successful, it returns a string that contains status information about the security level of the connection.

STOPCONNECTION

Stops SSL security on the connection. The TCP connection remains established. Future sends and receives are not encrypted.

If an error occurs, the SIOCTLCTL function returns an error code. Unless otherwise specified in the *ivalue* parameter, if the command is completed, the SIOCTLCTL function returns a string that contains the following information:

- Return code
- Policy status
- Connection status
- Security type
- SSL protocol
- Negotiated cipher that can be presented in 2 bytes
- Client user ID if available
- FIPS status
- Four-byte negotiated cipher
- Four-byte negotiated TLSv1.3 key share

For example, the following string is an example of what is returned by the SIOCTLCTL function:

```
0 5 2 2 0300 05 userid 01 0005 *NA*
```

In the example, 0 is the return code, 5 is the policy status, 2 is the connection status, the second 2 is the security type, 0300 is the SSL protocol, the second 05 is the negotiated cipher, *userid* is the user ID that is associated with the client's certificate in the SAF database, 01 is the FIPS 140 status, and 0005 is the four-byte negotiated cipher, and *NA* indicates that the key share is not applicable.

The policy status is one of the following values:

- 1** AT-TLS function is off. The TCP/IP address space is not enabled to support AT-TLS.
- 2** No AT-TLS policy is defined for the connection.
- 3** A policy is defined for the connection, but AT-TLS is not enabled for the connection.
- 4** A policy is defined for the connection, and AT-TLS is enabled for the connection.
- 5** A policy is defined for the connection, and both AT-TLS and the ApplicationControlled parameter are enabled for this connection.

The connection status is one of the following values:

- 1** The connection is not secure.

2 The connection handshake is in progress.

3 The connection is secure.

The security type is one of the following values:

0 Unknown. The connection is not secure.

1 Client.

2 Server.

3 Server with client authentication and authentication type set to PASSTHRU.

4 Server with client authentication and authentication type set to FULL.

5 Server with client authentication and authentication type set to REQUIRED.

6 Server with client authentication and authentication type set to SAFCheck.

The SSL protocol is one of the following values:

0000 Unknown. The connection is not secure.

0200 SSL Version 2.

0300 SSL Version 3.

0301 TLS Version 1.0

0302 TLS Version 1.1

0303 TLS Version 1.2

0304 TLS Version 1.3

The value FF indicates that the cipher has not been negotiated. SSL Version 2 cipher suites are 1 character. SSL Version 3 and TLS ciphers are 2 or 4 characters. The value 4X indicates that the cipher cannot be represented in 2 bytes. The 2-byte cipher must be obtained from the 4-byte cipher field. See the TTLS cipherParms statement description in [z/OS Communications Server: IP Configuration Reference](#) for additional cipher values.

The client user ID value is either the user ID that is associated with the client's certificate in the SAF database, or the character string **NONE** that indicates that the client ID is unknown.

The FIPS 140 status is one of the following values:

00 No FIPS 140 support

01 FIPS 140 On support

02 FIPS 140 Level1 support

03

FIPS 140 Level2 support

04

FIPS 140 Level3 support

If the cipher is not negotiated, the 4-byte cipher value is *NA*; otherwise, the 4-byte value of the negotiated cipher is returned. See the [TTLSCipherParms statement description in z/OS Communications Server: IP Configuration Reference](#) for additional cipher values.

When the negotiated protocol is TLSv1.3 (0304), the 4-byte value is the negotiated key share. See the [ClientKeyShareGroups parameter in the TTLSSignatureParms statement in z/OS Communications Server: IP Configuration Reference](#) for key share values. Otherwise, the field is *NA*.

Rule: When using AT-TLS application control, you must ensure that no outstanding data resides in the socket receive buffers for the application. If data exists when AT-TLS is enabled, the negotiation will fail. To flush the receive buffers, you can issue a nonblocking RECV command on the socket before you issue the IOCTL command.

Tips:

- For TCP/IP socket error information concerning this command, see the Application Transparent TLS information in [z/OS Communications Server: IP Programmer's Guide and Reference](#).
- For information about SSL error numbers, see [z/OS Cryptographic Services System SSL Programming](#).

ivalue

Additional information that is needed to run the requested command. The value of the *ivalue* parameter can be either input or output; it is independent of the *icmd* parameter.

idata

Additional information that is needed to run the requested command. The value of the *idata* parameter can be either input or output; it is independent of the *icmd* and *ivalue* parameter.

Returned value

The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

See [Appendix B, "Socket call error return codes," on page 743](#) for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 9 EBADF
- 6 ENXIO
- 22 EINVAL
- 41 EPROTOTYPE
- 45 EOPNOTSUPP
- 54 ECONNRESET

The following REXX socket API error numbers can be returned:

- 2001 EINVALIDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE
- 2009 ESOCKETNOTDEFINED
- 2023 EICWTNOINPUTDATA
- 2024 EICWTINVALIDTIMEOUT
- 2025 EICWTINVALIDREUSE

- 2026 EICWTINVALIDTOKEN

LE C/C++ equivalent

This command has no LE C/C++ equivalent.

Code example

See the EZARXS05 REXX sample in the SBLSCLI0 file for an example of using the IOCTL command.

LISTEN

Use the LISTEN command to determine whether a socket is ready to accept client connection requests.

The LISTEN command applies only to stream sockets. The LISTEN command performs two actions:

1. The command completes the bind process, if it has not already been done explicitly.
2. Creates a connection request queue (the backlog queue). New connections received by the stack for this socket are placed on the backlog queue. The application then can issue the ACCEPT command to process these pending connections. If the backlog queue is full, the stack rejects new connection requests.

Guideline: Applications that issue the LISTEN command should bind to a well-known port using INADDR_ANY or IN6ADDR_ANY. If the LISTEN command completes the bind for IPv4, the socket is bound to AF_INET 0 INADDR_ANY. If the LISTEN command completes the bind for IPv6, the socket is bound to AF_INET 0 0 IN6ADDR_ANY 0.

Format

```
➔ SOCKET — ( — "LISTEN" — , — socketid — { , — 10 — } — { , — backlog — } — ) ➔
```

Parameters

socketid

The socket descriptor

backlog

The number of pending connection requests. This parameter can have a value in the range 0 to the maximum number that is specified by the SOMAXCONN parameter in the TCPIP profile. If the *backlog* parameter is set to 0, 1, or 2, then the backlog queue uses the default value of 2. If the *backlog* parameter is not specified, by default it is set to 10.

Returned value

The command returns a string that contains the return code. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

See [Appendix B, “Socket call error return codes,” on page 743](#) for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 9 EBADF
- 22 EINVAL,
- 38 ENOTSOCK,
- 45 EOPNOTSUPP,

The following REXX socket API error numbers can be returned:

- 2001 EINVALDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE
- 2006 ESOCKETNOTALLOCATED
- 2009 ESOCKETNOTDEFINED
- 2012 EINVALIDNAME

LE C/C++ equivalent

This command has no LE C/C++ equivalent.

Code example

See [“ACCEPT” on page 586](#) for an example of how to use the LISTEN command.

READ

Use the READ command to read data on the specified socket. The maximum amount of data to be read is specified by the *maxlength* parameter. If the socket is in blocking mode and data is not available on the socket, the command blocks until data arrives.

Consider the following additional information:

- If the socket is a stream socket and the length of the data returned is 0, the remote peer has closed its side of the connection.
- If the socket is a connected datagram socket, the command returns data up to the length specified by the *maxlength* parameter. The remainder of the datagram is discarded. To ensure that the entire datagram is received, set the *maxlength* parameter to 65535 or greater.

Guidelines:

- For stream sockets, data is processed as streams of information with no boundaries separating the data. The application provides record management. Applications should place the command in a loop until all the data has been received.
- For nonblocking sockets, use the SELECT command to determine whether there is data to be read on the socket.

Tip: If the SO_ASCII socket option is enabled, then the data received is translated from EBCDIC to ASCII.

Format

➔ SOCKET — (— "READ" — , — *socketid* — { — , — 10 000 — } — { — , — *maxlength* — } —) ➔

Parameters

socketid

The socket descriptor.

maxlength

The maximum amount of data (in bytes) to be returned. The *maxlength* parameter can be a number in the range 0-100 000. By default, this parameter is set to 10 000.

Returned value

The command returns a string that contains the return code, the maximum length of the data returned, and the data, for example, 0 19 This is sample data. The return code can be 0, a REXX socket

API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully. The data length 0 indicates that the connection was closed by the remote peer.

See [Appendix B, “Socket call error return codes,”](#) on page 743 for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 4 EITNR
- 5 EIO
- 9 EBADF
- 22 EINVAL
- 35 EWOULDBLOCK
- 38 ENOTSOCK
- 45 EOPNOTSUPP
- 54 ECONNRESET
- 57 ENOTCONN
- 60 ETIMEDOUT

The following REXX socket API error numbers can be returned:

- 2001 EINVALIDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE
- 2009 ESOCKETNOTDEFINED

LE C/C++ equivalent

```
ssize_t read(int fs, void *buf, size_t N);
```

Code example

```
/* REXX EZARXR21 */
/*
 * This sample demonstrates the use of the READ and RECV
 * socket commands.
 *
 * To use the READ command, set the variable g_RECVCMD equal to "READ"
 * to use the RECV command, set the variable g_RECVCMD equal to "RECV"
 *
 * The program creates a listening socket and then goes into a
 * loop and blocks on the accept command. When a new connection is
 * ACCEPTED the program will issue the READ or RECV command until
 * the connection is terminated.
 *
 * If the data received is the string "DONE", then the
 * program will close the accepted socket and wait for a new
 * connection request.
 */
g_RECVCMD = "READ"
src = socket("INITIALIZE", "MYSET01", 10);
if perror(src, "INITIALIZE") = 0 then do
  src = socket("SOCKET", "AF_INET6", "STREAM");
  if perror(src, "SOCKET") = 0 then do
    parse var src . l_sockid
    l_name6 = "AF_INET6 54004 0 ::0 0";
    src = socket("BIND", l_sockid, l_name6);
    if perror(src, "BIND") = 0 then do
      src = socket("LISTEN", l_sockid);
      if perror(src, "LISTEN") = 0 then do
        say "Listening on socket " l_sockid;
        do forever
          src = socket("ACCEPT", l_sockid);
          if perror(src, "ACCEPT") = 0 then do
```

```

parse var src . l_newsockid . ;
l_datalen = -1;
l_Done = "FALSE";
l_totallen = 0;
l_packet = "";
/* *****
* Loop around RECV|READ command until all data has
* has been received and the client closes the
* connection.
* *****/
do until l_datalen = 0 | l_done = "TRUE"
  src = socket(g_RECVCMD,l_newsockid,512);
  if perror(src,g_RECVCMD) = 0 then do
    parse var src l_retcode l_datalen l_data
    if l_datalen > 0 then do
      l_totallen = l_totallen + l_datalen;
      if l_packet = "" then do
        l_packet = l_data;
        if l_packet = "DONE" then
          l_done = "TRUE";
      end;
    else l_packet = l_packet||l_data;
  end;
end;

else do
/* *****
* A data length of zero indicates the
* connection has been closed by the
* remote side
* *****/
  Say "Connection has been closed",
    "received "l_totallen" bytes";
  l_done = "TRUE";
end;
end;
else do
  l_done = "TRUE";
end;
end; /* DO READ */
src = socket("CLOSE",l_newsockid);
src = perror(src,"CLOSE");
end; /* ACCEPT */
end; /* DO FOREVER */
end;
end;
end; /* INITIALIZE */
src = perror(socket("TERMINATE","MYSET01"),"TERMINATE");
exit 0;

/* This routine returns -1 if the first word if arg 1 is not zero */
perror: if word(arg(1),1) = 0 then return 0; else
  Say arg(2) "Error : "arg(1);
  return -1;

```

Figure 159. READ command example

RECV

Use the RECV command to receive data on a specified socket. The RECV command can be issued only against connected sockets.

Consider the following additional information:

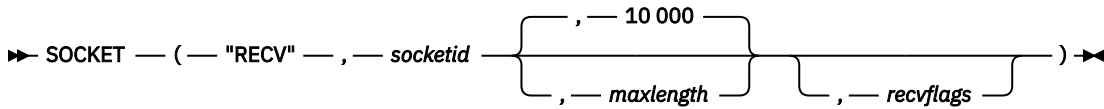
- If the socket is in blocking mode and data is not available, the command blocks until data arrives. If the socket is in nonblocking mode and data is not available, the command returns the 35 EWOULDBLOCK error code.
- If the socket is a stream socket and the length of the data returned is 0, the remote peer has closed its side of the connection.
- If the socket is a connected datagram socket, the command returns data up to the length specified by the *maxlength* parameter. The remainder of the datagram is discarded. To ensure that the entire datagram is received, set the *maxlength* parameter to 65535 or greater.

Guidelines:

- For stream sockets, data is processed as streams of information with no boundaries separating the data. The application provides record management. Applications should place the command in a loop until all the data has been received.
- For nonblocking sockets, use the SELECT command to determine whether there is data to be read on the socket.

Tip: If the SO_ASCII socket option is enabled, then the data received is translated from EBCDIC to ASCII.

Format



Parameters

socketid

The socket descriptor.

maxlength

The maximum amount of data (in bytes) to be returned. The *maxlength* parameter can be a number in the range 0-100 000. By default, this parameter is set to 10 000.

recvflags

An optional parameter. Specifies the following receive flags:

MSG_OOB, OOB

Receive out-of-band data (stream sockets only). Even if the OOB flag is not set, out-of-band data can be read if the SO_OOBLIN option is set for the socket.

MSG_PEEK, PEEK

Peek at the data, but do not destroy data. If the peek flag is set, the next receive operation will read the same data.

MSG_WAITALL, WAITALL

Requests that the function block until the full amount of data requested can be returned (stream sockets only). The function may return a smaller amount of data if the connection is terminated, an error is pending, or SO_RCVTIMEO is set and the timer expired for the socket.

Returned value

The command returns a string that contains the return code, the maximum length of the data returned, and the data, for example, 0 19 This is sample data. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

See [Appendix B, "Socket call error return codes," on page 743](#) for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 4 EITNR
- 5 EIO
- 9 EBADF
- 22 EINVAL
- 35 EWOULDBLOCK
- 38 ENOTSOCK
- 45 EOPNOTSUPP
- 54 ECONNRESET

- 57 ENOTCONN
- 60 ETIMEDOUT

The following REXX socket API error numbers can be returned:

- 2001 EINVALDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE
- 2009 ESOCKETNOTDEFINED

LE C/C++ equivalent

```
int recv(int socket, char *buffer, int length, int flags);
```

Code example

See [“ACCEPT” on page 586](#) or [“READ” on page 645](#). To use the RECV command, substitute the command RECV for the command READ.

RECVFROM

Use the RECVFROM command to receive data on the specified socket.

If the number of bytes is less than the number of bytes requested, the command returns the number of bytes that are available. If the socket is in blocking mode and data is not available on the socket, the command blocks until data arrives. When the socket is in nonblocking mode and data is not available, the command returns the 35 EWOULDBLOCK return code .

Consider the following additional information:

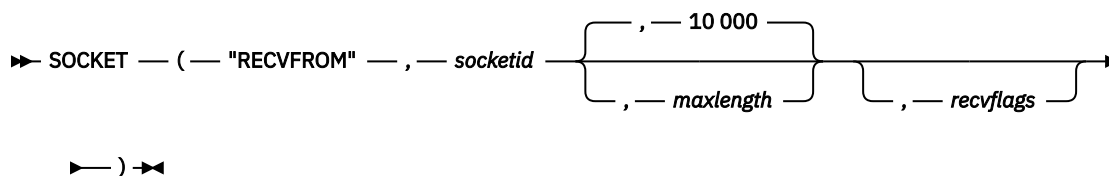
- If the socket is a stream socket and the length of the data returned is 0, the remote peer has closed its side of the connection.
- If the socket is a datagram socket, the command returns data up to the length specified by the *maxlength* parameter. The remainder of the datagram is discarded. If the socket is a datagram socket and the amount of data returned is 0, a datagram packet was received with no data.

Guidelines:

- Use the RECV command for stream and connected UDP sockets. For stream sockets, data is processed as streams of information with no boundaries separating the data. Applications should place the RECVFROM command in a loop until all the data has been received.
- If the socket is a datagram socket, the RECVFROM command returns the name of the remote partner. If the socket is a stream socket, use the command GETPEERNAME to determine the name of the remote partner.
- Use the SELECT command to determine whether there is data to be read on the socket.

Tip: If the SO_ASCII socket option is enabled, then the data received is translated from EBCDIC to ASCII.

Format



Parameters

socketid

The socket descriptor.

maxlength

The maximum amount of data (in bytes) to be returned. The *maxlength* parameter can be a number in the range 0-100 000. By default, this parameter is set to 10 000.

recvflags

An optional parameter. Specifies the following receive flags:

MSG_OOB, OOB

Receive out-of-band data (stream sockets only). Even if the OOB flag is not set, out-of-band data can be read if the SO_OOBINLINE option is set for the socket.

MSG_PEEK, PEEK

Peek at the data, but do not destroy data. If the peek flag is set, the next receive operation will read the same data.

MSG_WAITALL, WAITALL

Requests that the function block until the full amount of data requested can be returned (stream sockets only). The function may return a smaller amount of data if the connection is terminated, an error is pending, or SO_RCVTIMEO is set and the timer expired for the socket.

Returned value

The command returns a string that contains the return code, a NAME string, the maximum length of the data returned, and the data. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

The following list are examples of what is returned by the RECVFROM command.

IPv4 socket

```
0 AF_INET 54004 10.1.2.3 19 This is sample data
```

IPv6 socket

```
0 AF_INET6 54004 0 2001:10:1:2::3 0 19 This is sample data
```

In the examples, 0 is the return code, AF_INET 54004 10.11.103.1 or AF_INET6 54004 0 2001:10:1:2::3 0 is the socket name of the remote partner, 19 is the length of the data received, and This is sample data is the data that was received on the socket.

For information about the format of the NAME string, see [“How structures are represented” on page 580](#). See Appendix B, [“Socket call error return codes,” on page 743](#) for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 4 EITNR
- 5 EIO
- 9 EBADF
- 22 EINVAL
- 35 EWOULDBLOCK
- 38 ENOTSOCK
- 45 EOPNOTSUPP
- 54 ECONNRESET
- 57 ENOTCONN
- 60 ETIMEDOUT

The following REXX socket API error numbers can be returned:

- 2001 EINVALDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE
- 2009 ESOCKETNOTDEFINED

LE C/C++ equivalent

```
int recvfrom(int socket, char *buffer, int length, int flags,
struct sockaddr *address, int *address_length);
```

Code example

```
/* REXX EZARXR21 */
/*
 * This sample demonstrates the use of the READ and RECV
 * socket commands.
 *
 * To use the READ command, set the variable g_RECVCMD equal to "READ"
 * to use the RECV command, set the variable g_RECVCMD equal to "RECV"
 *
 * The program creates a listening socket and then goes into a
 * loop and blocks on the accept command. When a new connection is
 * ACCEPTED the program will issue the READ or RECV command until
 * the connection is terminated.
 *
 * If the data received is the string "DONE", then the
 * program will close the accepted socket and wait for a new
 * connection request.
 */
g_RECVCMD = "READ"
src = socket("INITIALIZE", "MYSET01", 10);
if perror(src, "INITIALIZE") = 0 then do
  src = socket("SOCKET", "AF_INET6", "STREAM");
  if perror(src, "SOCKET") = 0 then do
    parse var src . l_sockid
    l_name6 = "AF_INET6 54004 0 ::0 0";
    src = socket("BIND", l_sockid, l_name6);
    if perror(src, "BIND") = 0 then do
      src = socket("LISTEN", l_sockid);
      if perror(src, "LISTEN") = 0 then do
        say "Listening on socket " l_sockid;
        do forever
          src = socket("ACCEPT", l_sockid);
          if perror(src, "ACCEPT") = 0 then do
            parse var src . l_newsockid . ;
            l_datalen = -1;
            l_Done = "FALSE";
            l_totallen = 0;
            l_packet = "";
            /* *****
             * Loop around RECV|READ command until all data has
             * has been received and the client closes the
             * connection.
             * *****/
            do until l_datalen = 0 | l_Done = "TRUE"
              src = socket(g_RECVCMD, l_newsockid, 512);
              if perror(src, g_RECVCMD) = 0 then do
                parse var src l_retcode l_datalen l_data
                if l_datalen > 0 then do
                  l_totallen = l_totallen + l_datalen;
                  if l_packet = "" then do
                    l_packet = l_data;
                    if l_packet = "DONE" then
                      l_Done = "TRUE";
                end;
                else l_packet = l_packet||l_data;
              end;
            else do
              /* *****
               * A data length of zero indicates the
               * connection has been closed by the
               * remote side
               * *****/
              Say "Connection has been closed",
```

```

        "received "l_totallen" bytes";
        l_done = "TRUE";
    end;
end;
else do
    l_done = "TRUE";
end;
end; /* DO READ */
src = socket("CLOSE",l_newsockid);
src = perror(src,"CLOSE");
end; /* ACCEPT */
end; /* DO FOREVER */
end;
end;
end; /* INITIALIZE */
src = perror(socket("TERMINATE","MYSET01"),"TERMINATE");
exit 0;

/* This routine returns -1 if the first word if arg 1 is not zero */
perror: if word(arg(1),1) = 0 then return 0; else
    Say arg(2) "Error : "arg(1);
    return -1;

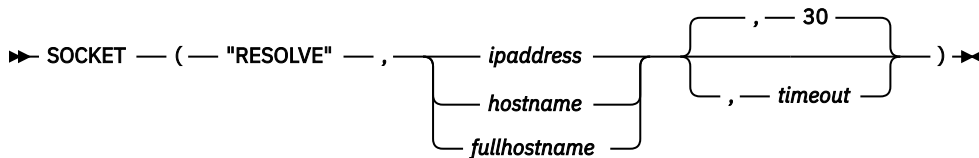
```

Figure 160. READ command example

RESOLVE

Use the RESOLVE command to resolve a host name or an IP address.

Format



Parameters

ipaddress

The IPv4 address of the system in dotted decimal format.

hostname

The host name, for example, BOB01. The maximum length is 255 characters.

Fullhostname

The fully qualified host name, for example, BOB01.THEWORLD.COM. The maximum length is 255 characters.

timeout

The resolver timeout value. By default, this is set to 30 seconds.

Returned value

The command returns a string that contains the return code, the IP address, and the host name. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

The following string is an example of what is returned by the RESOLVE command:

```
0 10.201.202.1 BOB01.THEWORLD.COM
```

In the example, 0 is the return code, 10.201.202.1 is the host IP address, and BOB01.THEWORLD.COM is the host name.

See [Appendix B, "Socket call error return codes,"](#) on page 743 for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error number can be returned:

- 22 EINVAL

The following REXX socket API error numbers can be returned:

- 2001 EINVALIDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE
- 2012 EINVALIDNAME
- 2016 EHOSTNOTFOUND

LE C/C++ equivalent

This command has no LE C/C++ equivalent.

Code example

```
/* REXX EZARXR23 */
/*
 * This sample demonstrates the use of the RESOLVE
 * socket command.
 */
src = socket("INITIALIZE","MYSET01");
src = socket("RESOLVE","your.hostname.here");
if word(src,1) = 0 then do
  Say "The host name is "WORD(src,3)
  Say "The IP Address is "WORD(src,2);
end;
src = socket("TERMINATE","MYSET01");
exit 0;
```

Figure 161. RESOLVE command example

SELECT

Use the SELECT command to monitor groups of sockets to determine when one or more of the sockets is ready for a read operation, is ready for a write operation, or has an exception pending.

Guidelines:

- A close on the other side of a socket connection is reported not as an exception but as a read event, which returns 0 bytes of data.
- When the CONNECT command is called with a socket descriptor in nonblocking mode, set up completion is reported as a write event on the socket.
- The SELECT command returns an exception pending when either a connection is reset or when a TAKESOCKET command is completed for a socket that was previously given using the GIVESOCKET command.

Format

➡ SOCKET — (— "SELECT" — , — *fdset* — , — *timeout* —) —>

Parameters

fdset

Specifies the set of socket descriptors to be monitored for activity. The *fdset* parameter is a string in the following format:

```
"READ" rdlist "WRITE" wrlist "EXCEPTION" exlist
```

where:

- The *rdlist* value is a space-delimited list of sockets to be monitored for reading.
- The *wrlist* value is a space-delimited list of sockets to be monitored for writing.
- The *exlist* value is a space-delimited list of sockets to be monitored for exceptions.

To specify that all sockets are monitored, set the value of the *rdlist*, *wrlist*, or *exlist* string to an asterisk (*).

For example, if you want to monitor the sockets with files descriptors 1, 2, and 3 to determine when the socket has data to be read, issue the following command:

```
SOCKET("SELECT","READ 1 2 3 WRITE EXCEPTION",120);
```

This code example also sets a *timeout* parameter of 120 seconds.

To specify that no sockets are monitored, issue the following command:

```
SOCKET("SELECT","READ WRITE EXCEPTION",);
```

timeout

A positive integer that indicates the maximum length of time (in seconds) that the SELECT command will monitor the sockets. If no *timeout* parameter is specified, the sockets are monitored indefinitely.

Returned value

The command returns a string that contains the return code, the number of ready sockets, and information about the ready sockets. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

The following string is an example of what is returned by the SELECT command:

```
0 2 READ 1 3 WRITE EXCEPTION
```

In this example, 0 is the return code, 2 is the number of ready sockets, and 1 and 3 are the sockets that ready for READ operations. There are no sockets ready for WRITE operations or sockets with pending exceptions.

See [Appendix B, “Socket call error return codes,” on page 743](#) for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 9 EBADF
- 22 EINVAL
- 38 ENOTSOCK
- 45 EOPNOTSUPP

The following REXX socket API error numbers can be returned:

- 2001 EINVALIDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE
- ESOCKETNOTDEFINED
- 2012 EINVALIDNAME

LE C/C++ equivalent

This command has no LE C/C++ equivalent.

Code example

```
/* REXX EZARXR24 */
/*
 * This is an example of a function command that can be used to
 * to determine if a connection is available. This procedure
 * would be called before calling the ACCEPT command.
 *
 * Two arguments are passed. The p_listensocketlist is a
 * list of 1 or more space-delimited socket descriptors to be
 * monitored. The p_timeout value is how long the SELECT
 * command should wait before returning.
 *
 * The monitored sockets can be BLOCKING or NON-BLOCKING
 * sockets.
 *
 * Example usage:
 *
 * INITIALIZE a socket set.
 * open a SOCKET descriptor.
 * BIND the socket to a well known port.
 * set the socket to passive mode using the LISTEN command.
 * LOOP:
 * Call the IsConnectionAvailable function
 * If a connection is available, ACCEPT the connection
 * and continue processing. Otherwise, do some other work,
 * and loop.
 *
 * Function Usage:
 *   l_retcode = IsConnectionAvailable(l_sockid,60);
 *
 * The function will return one of the following:
 * - The list of socket descriptors that have
 *   connections pending.
 * - The string "TIMEOUT". This indicates the select command
 *   timed out before any connections arrived.
 * - the string "ERROR". This indicates an ERROR occurred
 *   when the SELECT command was issued.
 */
IsConnectionAvailable: PROCEDURE
Parse arg p_listensocketlist, p_timeout
l_fdset = "READ "p_listensocketlist" WRITE EXCEPTION";
l_retvalue = SOCKET("SELECT",l_fdset,p_timeout);
parse var l_retvalue l_retcode l_numSockets;
if l_retcode = 0 then do
  if l_numsocket > 0 then do
    parse value l_retvalue with 'READ' l_sockidrdlist 'WRITE' .;
    parse value l_retvalue with 'WRITE' l_sockidwrlist 'EXCEPTION' .
    parse value l_retvalue with 'EXCEPTION' l_sockidexlist;
    l_retcode = l_sockidrdlist;
  end
  else do
    Say "Select command timed out";
    l_retcode = "TIMEOUT"
  end;
end;
else do
  l_retcode = "ERROR";
end;
return l_retcode;
```

Figure 162. SELECT command example

Tip: See the IOCTL command for another example on how to use SELECT command.

SEND

Use the SEND command to send an outgoing message on the connected socket.

When the socket is a TCP socket, the following conditions apply:

- If the socket is in blocking mode and if the total amount of data to be sent cannot be processed by the stack when the command is issued, the command blocks until the data can be sent.

- If the socket is in nonblocking mode and if the total amount of data to be written cannot be processed by the stack when the command is issued, the command returns the number of bytes that were successfully written. If none of the data can be written, the command returns the value -1 and the 35 EWOULDBLOCK error message .

When the socket is a connected UDP socket, the SEND command either is completed or failed. A connected UDP socket does not return the 35 EWOULDBLOCK error message .

Guideline: Place the SEND command in a loop to ensure that all the data is written. For a TCP socket, a partial write operation might occur regardless of whether the socket is in blocking or nonblocking mode. A partial write operation occurs when the stack copies some but not all of the application data:

- If a partial write operation occurs on a socket in blocking mode, the blocking socket is interrupted. The return value contains the number of bytes written, and the return code contains the reason for the interruption. In such cases, consider ending the connection.
- If a partial write operation occurs on a socket in nonblocking mode, the return value indicates the number of bytes that were successfully sent. If this number is less than the number of bytes specified on the SEND command, repeat the SEND command until all data is written. The blocking condition can last for a long time, so use other strategies to ensure that the application does not remain in a busy loop sending data.

Tips:

- Use the SELECT command to determine whether a socket is ready to send additional data. To do so, test the socket for a WRITE event.
- If the SO_ASCII socket option is enabled, then the data received is translated from EBCDIC to ASCII.

Format

```
➤ SOCKET — ( — "SEND" — , — socketid — , — data — , — sendflags — ) ➤
```

Parameters

socketid

The socket descriptor.

data

The data to be sent.

sendflags

Optional flags that specify how the data is sent. The following flags are supported:

MSG_OOB

Sends the data out-of-band. Out-of-band data is supported only for stream sockets created in the AF_INET domain. The values OOB or OUT_OF_BAND are supported also.

MSG_DONTRoute

Specifies that the data is routed by the calling program. The value DONTRoute is supported also.

Returned value

The command returns a string that contains the return code and the amount of data sent, for example, 0 19. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

See [Appendix B, “Socket call error return codes,” on page 743](#) for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 4 EINTR

- 9 EBADF
- 5 EIO
- 22 EINVAL
- 32 EPIPE
- 35 EWOULDBLOCK
- 38 ENOTSOCK
- 40 EMSGSIZE
- 45 EOPNOTSUPP
- 54 ECONNRESET
- 57 ENOTCONN

The following REXX socket API error numbers can be returned:

- 2001 EINVALDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE
- 2009 ESOCKETNOTDEFINED

LE C/C++ equivalent

```
int send(int socket, char *buffer, int length, int flags);
```

Code example

See the EZARXS03 REXX sample in the SBLSCLI0 file for an example of using the SEND command.

SENDTO

Use the SENDTO command to send an outgoing message on a socket descriptor. This command differs from the SEND command in that it includes the destination address as a parameter.

This command is used primarily to send data using connectionless protocols such as UDP or RAW. Use the SENDTO command to send datagrams on a UDP socket regardless of whether the socket is connected.

Guidelines:

- For TCP or connected UDP sockets, use the SEND command.
- If the SO_ASCII socket option is enabled, then the data received is translated from EBCDIC to ASCII.

Format

```
➤ SOCKET — ( — "SENDTO" — , — socketid — , — data — , — sendflags — , — name — ➤
➤ ) ➤
```

Parameters

socketid

The socket descriptor.

data

The data to be sent.

sendflags

Optional flags that specify how the data is sent. The following flag is supported:

MSG_DONTRROUTE

Do not route the data. Routing is handled by the calling program. The DONTRROUTE value is supported also.

name

The socket name of the remote host to which the data is sent.

The format for the *name* parameter depends on the socket type:

AF_INET sockets (IPv4)

name = "*domain portid ipaddress*"

AF_INET6 sockets (IPv6)

name = "*domain portid flowinfo ipaddress scopeid*"

where

- The *domain* value is the decimal number 2 for AF_INET and the decimal number 19 for AF_INET6.
- The *portid* value is the port number.
- The *ipaddress* value is the IP address of the remote host. It must be an IPv4 address for AF_INET and an IPv6 address for AF_INET6.
- The *flowinfo* value must be 0.
- The *scopeid* value identifies the interfaces that are applicable for the scope of the address that is specified in the *ipaddress* field. For a link-local IP address, the *scopeid* field can specify a link index, which identifies a set of interfaces. For all other scopes, the *scopeid* field must be set to 0. Setting the *scopeid* field to 0 indicates that any address type and scope can be specified.

Returned value

The command returns a string that contains the return code and an integer that specifies the amount of data that was sent, for example, 0 192. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

See [Appendix B, "Socket call error return codes," on page 743](#) for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 4 EINTR
- 9 EBADF
- 5 EIO
- 22 EINVAL
- 32 EPIPE
- 35 EWOULDBLOCK
- 38 ENOTSOCK
- 40 EMSGSIZE45 EOPNOTSUPP
- 54 ECONNRESET
- 57 ENOTCONN

The following REXX socket API error numbers can be returned:

- 2001 EINVALIDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE
- 2009 ESOCKETNOTDEFINED

LE C/C++ equivalent

```
int sendto(int socket, char *buffer, int length, int flags, struct sockaddr
*address, int address_len);
```

Code example

See [“RECVFROM” on page 649](#) for an example of using the SENDTO command.

SETSOCKOPT

Use the SETSOCKOPT command to set socket options.

Format

```
► SOCKET — ( — "SETSOCKOPT" — , — socketid — , — levelname — , — optname — , ►
    ► optvalue — ) ►
```

Parameters

socketid

The socket descriptor.

levelname

The protocol level. The following protocol levels are supported:

IPPROTO_TCP

Set socket options at the TCP layer

IPPROTO_IP

Set IPv4 socket options at the IP layer

IPPROTO_IPV6

Set IPv6 socket options at the IP layer

SOL_SOCKET

Set socket options at the socket layer

optname

The option or options. The following rules apply:

- Options that begin with SO_ require the SOL_SOCKET protocol level.
- Options that begin with TCP_ require the IPPROTO_TCP protocol level.
- Options that begin with IP_ require the IPPROTO_IP protocol level.
- Options that begin with IPV6_ require the IPPROTO_IPV6 protocol level.
- Options that begin with MCAST require the IPPROTO_IP or IPPROTO_IPV6 protocol level.

The following values are supported for the *optname* parameter:

IP_ADD_MEMBERSHIP

(IPv4-only) Enables an application to join a multicast group on a specific interface. Applications in a multicast group can receive multicast datagrams. An application can join multiple multicast groups on the same interface or the same multicast group on multiple interfaces, but only one interface address can be specified with a single command. The *optvalue* parameter must be a string that contains the multicast address followed by the interface address on which the application wants to receive multicast datagrams, for example, "224.224.224.1 10.11.13.4". This command returns the return code or error number.

Tip: Use the IOCTL command with the SIOCGIFADDR option to determine the interface address.

IP_ADD_SOURCE_MEMBERSHIP

(IPv4-only) Enables an application to join a multicast group on a specific interface and a specific source address. The *optvalue* parameter must be a string that contains the multicast address, the interface address, and the source address, for example, "224.224.224.1 10.11.16.103 10.11.107.1". The source address represents a filter; the application receives multicast packets only if the source address matches the source address filter for the multicast group.

This command returns the return code or error number.

Restrictions:

- Only one interface address can be specified with a single call.
- The stack supports up to 64 source address filters for each multicast-group interface pair. If the number of filters exceeds the maximum, ENOBUFS is returned.
- You can specify only a single source address with each call. If you want to join a multicast group and receive data from two different source addresses, then issue the SETSOCKOPT command twice.

Guideline: Applications that want to receive multicast datagrams need to join multicast groups. Use this option when the application wants to receive multicast packets on a specific group from one or more senders

IP_BLOCK_SOURCE

(IPv4-only) Enables an application to block multicast packets that are sent from a specific address. The application must have previously joined the multicast group. The *optvalue* parameter must be a string that contains the multicast address, source address, and interface address, for example, "224.224.224.1 10.11.16.103 10.11.107.1". This option returns 0 if it is successfully completed; otherwise, it returns the error number.

IP_DROP_MEMBERSHIP

(IPv4-only) Enables an application to exit a multicast group. If source filtering is enabled, all source filters are deleted. The *optvalue* parameter must be a string that contains the multicast address and the interface address, for example, "224.224.224.1 10.11.13.4". This option returns 0 if it is successfully completed; otherwise, it returns the error number.

IP_DROP_SOURCE_MEMBERSHIP

(IPv4-only) Enables an application to leave a multicast-source multicast group. The application will no longer receive multicast packets from the group. The *optvalue* parameter must be a string that contains the multicast address, source address, and interface address, for example, "224.224.224.1 10.11.13.4 10.11.107.1". This option returns 0 if it is successfully completed; otherwise, it returns the error number.

IP_MULTICAST_IF

(IPv4-only) Sets the IPv4 interface address that is used to send outbound multicast datagrams. Multicast datagrams can be sent only on one interface at a time. The *optvalue* parameter is the IP address of the interface. This option returns 0 if it is successfully completed; otherwise, it returns the error number.

IP_MULTICAST_LOOP

(IPv4-only) Controls whether a multicast datagram is looped back on the outgoing interface by the IP layer for local delivery when datagrams are sent to a group to which the sending host belongs. By default, loopback is enabled. The *optvalue* parameter must be one of the following values: 0 (disabled) or 1 (enabled). This option returns 0 if it is successfully completed; otherwise, it returns the error number.

IP_MULTICAST_TTL

(IPv4-only) Sets the IP time-to-live of outgoing multicast datagrams. By default, this is set to 1; multicast is available only to the local subnet. The *optvalue* parameter must be an integer in the range 1 - 255.

IP_UNBLOCK_SOURCE

(IPv4-only) Enables an application to unblock a previously blocked source address for an IPv4 multicast group. Only one interface address can be specified with a single call. The *optvalue*

parameter must be a string that contains the multicast address, source address, and interface address, for example, "224.224.224.1 10.11.103.1 10.11.107.1". This option returns 0 if it is successfully completed; otherwise, it returns the error number.

IPV6_ADDR_PREFERENCES

(AF_INET6 only) Sets the IPv6 address preferences to be used when selecting the source address. The following are the valid IPV6_ADDR_PREFERENCES flags:

IPV6_PREFER_SRC_HOME

A home IPv6 address is preferred over a care-of IPv6 address.

IPV6_PREFER_SRC_COA

A care-of IPv6 address is preferred over a home IPv6 address.

IPV6_PREFER_SRC_TMP

A temporary IPv6 address is preferred over a public IPv6 address.

IPV6_PREFER_SRC_PUBLIC

A public IPv6 address is preferred over a temporary IPv6 address.

IPV6_PREFER_SRC_CGA

A cryptographically generated IPv6 address is preferred over a non-cryptographically generated IPv6 address.

IPV6_PREFER_SRC_NONCGA

A non-cryptographically generated IPv6 address is preferred over a cryptographically generated IPv6 address.

You can specify a single flag or multiple flags that are separated by blanks.

Results:

- Combining contradictory flags, such as IPV6_PREFER_SRC_CGA and IPV6_PREFER_SRC_NONCGA, results in error code 2020 (EINVALCOMBINATION).
- The stack could assign a source IP address that does not conform to one or more of the IPV6_ADDR_PREFERENCES flags that you have set.

IPV6_JOIN_GROUP

(IPv6-only) Enables an application to join a multicast group on a specific interface. Only applications that want to receive multicast datagrams need to join multicast groups. An application can join multiple multicast groups on the same interface, or it can join the same multicast group on multiple interfaces. The *optvalue* parameter must be a string that contains the multicast address and the index of the interface on which the application wants to receive multicast datagrams, for example, "FF02:225:9:10::11 3". If the interface index is set to 0, the stack will choose the local address. This option returns 0 if it is successfully completed; otherwise, it returns the error number.

Guideline: Use the SIOCGIFNAMEINDEX function of the IOCTL command to determine the index number for an interface.

Restriction: Only one interface address can be specified in a single call. A multicast address can be associated with a real interface only.

IPV6_LEAVE_GROUP

(IPv6-only) Enables an application to leave a multicast group. The *optvalue* parameter must be a string that contains the multicast address and the interface address, for example, "FF02:225:9:10::11 3". The *optvalue* parameter must match the original IPV6_JOIN_GROUP parameters; for example, if the interface index specified for the IPV6_JOIN_GROUP was 0, then 0 also must be specified as the interface index for IPV6_LEAVE_GROUP command. This option returns 0 if it is successfully completed; otherwise, it returns the error number.

IPV6_MULTICAST_HOPS

(IPv6-only) Sets the hop limit that is used for outgoing multicast packets. The *optvalue* parameter is optional; if it is not issued, the hop limit is set to 1.

The *optvalue* parameter can have the following values:

-1

The default value for the stack is used.

0-255

The hop limit.

This option returns 0 if it is successfully completed; otherwise, it returns the error number.

Restriction: To set the hop limit value to be greater than the TCP/IP default value, a REXX-application user ID must have superuser authority.

IPV6_MULTICAST_IF

(IPv6-only) Sets the index of the IPv6 interface that is used to send outbound multicast datagrams from the socket application. The *optvalue* parameter must specify the interface index number, for example, 34. This option returns 0 if it is successfully completed; otherwise, it returns the error number.

IPV6_MULTICAST_LOOP

(IPv6-only) Controls whether a multicast datagram is looped back on the outgoing interface by the IP layer for local delivery when datagrams are sent to a group to which the sending host belongs. By default, multicast datagrams are looped back. The *optvalue* parameter must be one of the following values: 0 (disabled) or 1 (enabled). This option returns 0 if it is successfully completed; otherwise, it returns the error number.

IPV6_UNICAST_HOPS

(IPv6-only) Sets the hop limit that is used for outgoing unicast IPv6 packets. The *optvalue* parameter is optional; if it is not issued, the hop limit is set to 1.

The *optvalue* parameter can have the following values:

-1

The default value for the stack is used.

0-255

The hop limit.

This option returns 0 if it is successfully completed; otherwise, it returns the error number.

Restriction: An application must be APF authorized to set the hop limit value to be greater than the TCP/IP default value. This option is not valid when used in CICS applications. CICS applications cannot run as APF authorized.

IPV6_V6ONLY

(IPv6-only) Restricts a socket to sending and receiving IPv6 packets only. By default, a socket is not restricted. The *optvalue* parameter must be one of the following values: 0 (disabled) or 1 (enabled). This option returns 0 if it is successfully completed; otherwise, it returns the error number.

MCAST_BLOCK_SOURCE

Enables an application to block multicast packets from a specific source address. The multicast group must have been previously joined. The *optvalue* parameter must be a string that contains the interface index, the multicast address, and the source address. Specify the multicast address and source address using the NAME string. The following string is an example of what might be coded for the *optvalue* parameter:

```
"45 AF_INET6 54666 0 FF02::32:1 0 AF_INET6 0 0 2001:10:11:107::1 0"
```

For more information about the format of the NAME string, see [“How structures are represented” on page 580](#). This option returns 0 if it is successfully completed; otherwise, it returns the error number.

Tip: This option is valid for both IPv4 and IPv6.

MCAST_JOIN_GROUP

Enables an application to join a multicast group on a specific interface. Only applications that want to receive multicast datagrams need to join multicast groups. The *optvalue* parameter must be

a string that contains the interface index and a multicast address. Specify the multicast address using the socket address name format. The following string is an example of what you can code for the *optvalue* parameter:

```
"45 AF_INET 1234 224.224.224.1"
```

For more information about the format of the NAME string, see [“How structures are represented” on page 580](#). This option returns 0 if it is successfully completed; otherwise, it returns the error number.

Tip: This option is valid for both IPv4 and IPv6.

MCAST_JOIN_SOURCE_GROUP

Enables an application to join a source multicast group on a specific interface and source address. Only applications that want to receive multicast datagrams need to join source multicast groups. The *optvalue* parameter must be a string that contains the interface index, the multicast address, and the source address. Specify the multicast address and source address the using socket address name format. The following string is an example of what you can code for the *optvalue* parameter:

```
"45 AF_INET6 1234 0 FF02::123:1 0 AF_INET6 0 0 2001:10:11:107::1 0"
```

For more information about the format of the NAME string, see [“How structures are represented” on page 580](#). This option returns 0 if it is successfully completed; otherwise, it returns the error number.

Tip: This option is valid for both IPv4 and IPv6.

MCAST_LEAVE_GROUP

Enables an application to leave a multicast group or to leave all source multicast groups. The *optvalue* parameter must be a string that contains the interface index and the multicast address; specify the multicast address using the socket address name format. The following string is an example of what you can code for the *optvalue* parameter:

```
"45 AF_INET6 1234 0 FF02::123:1 0"
```

For more information about the format of the NAME string, see [“How structures are represented” on page 580](#). This option returns 0 if it is successfully completed; otherwise, it returns the error number.

Tip: This option is valid for both IPv4 and IPv6.

MCAST_LEAVE_SOURCE_GROUP

Enables an application to leave a source multicast group. The *optvalue* parameter must be a string that contains the interface index, the multicast address, and the source address. Specify the multicast address and source address using the socket address name format. The following string (all on one line) is an example of what you can code for the *optvalue* parameter:

```
"45 AF_INET6 1234 0 FF02::123:1 0 AF_INET6 1234 0  
2001:10:11:103::1 0"
```

For more information about the format of the NAME string, see [“How structures are represented” on page 580](#). This option returns 0 if it is successfully completed; otherwise, it returns the error number.

Tip: This option is valid for both IPv4 and IPv6.

MCAST_UNBLOCK_SOURCE

Enables an application to unblock multicast packets that are sent from a specific address. The multicast group must have been previously blocked. The *optvalue* parameter must be a string that contains the interface index, the multicast address, and the source address. Specify the multicast

address and source address using the socket address name format. The following string is an example of what you can code for the

```
optvalue
```

parameter:

```
"45 AF_INET6 1234 0 FF02::123:1 0 AF_INET6 1234 0 2001:10:11:103::1 0"
```

For more information about the format of the NAME string, see [“How structures are represented” on page 580](#). This option returns 0 if it is successfully completed; otherwise, it returns the error number.

Tip: This option is valid for both IPv4 and IPv6.

Restriction: Only one source address can be specified in a call.

SO_ASCII

(REXX only) Enables all incoming data to be translated from ASCII to EBCDIC, and all outgoing data to be translated from EBCDIC to ASCII. The *optvalue* parameter must be one of the following values: 0 (disabled) or 1 (enabled). This option returns a string that contains the error code and either ON (enabled) or OFF (disabled). If the option is enabled, the name of the translation table is returned also. The following string is an example of what might be returned: 0 ON MYTRANTB.

The translation tables are searched in the following order:

1. *user_prefix.subtaskid*.TCPXLBIN
2. *user_prefix.userid*.TCPXLBIN
3. *system_prefix*.STANDARD.TCPXLBIN
4. *system_prefix*.RXSOCKET.TCPXLBIN
5. Internal tables

The following descriptions apply:

- The *user_prefix* value is either the user ID or the job name of the REXX program.
- The *system_prefix* value is either TCPIP or the DATASETPREFIX value from the *hlq*.TCPIP.DATA. You can change the *system_prefix* value to match your site convention.
- The *subtaskid* value is the name of the socket set.
- The *userid* value is the user ID under which the REXX EXEC is running.

When the internal tables are used, the data is converted in the following way:

ASCII to EBCDIC		second hex digit of byte of ASCII data															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
first hex digit of byte of ASCII data	0	00	01	02	03	37	2D	2E	2F	16	05	25	0B	0C	0D	0E	0F
	1	10	11	12	13	3C	3D	32	26	18	19	3F	27	1C	1D	1E	1F
	2	40	4F	7F	7B	5B	6C	50	7D	4D	5D	5C	4E	6B	60	4B	61
	3	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	7A	5E	4C	7E	6E	6F
	4	7C	C1	C2	C3	C4	C5	C6	C7	C8	C9	D1	D2	D3	D4	D5	D6
	5	D7	D8	D9	E2	E3	E4	E5	E6	E7	E8	E9	4A	E0	5A	5F	6D
	6	79	81	82	83	84	85	86	87	88	89	91	92	93	94	95	96
	7	97	98	99	A2	A3	A4	A5	A6	A7	A8	A9	C0	6A	D0	A1	07
	8	20	21	22	23	24	15	06	17	28	29	2A	2B	2C	09	0A	1B
	9	30	31	1A	33	34	35	36	08	38	39	3A	3B	04	14	3E	E1
	A	41	42	43	44	45	46	47	48	49	51	52	53	54	55	56	57
	B	58	59	62	63	64	65	66	67	68	69	70	71	72	73	74	75
	C	76	77	78	80	8A	8B	8C	8D	8E	8F	90	9A	9B	9C	9D	9E
	D	9F	A0	AA	AB	AC	AD	AE	AF	B0	B1	B2	B3	B4	B5	B6	B7
	E	B8	B9	BA	BB	BC	BD	BE	BF	CA	CB	CC	CD	CE	CF	DA	DB
	F	DC	DD	DE	DF	EA	EB	EC	ED	EE	EF	FA	FB	FC	FD	FE	FF

Figure 163. ASCII to EBCDIC

EBCDIC to ASCII		second hex digit of byte of EBCDIC data															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
first hex digit of byte of EBCDIC data	0	00	01	02	03	9C	09	86	7F	97	8D	8E	0B	0C	0D	0E	0F
	1	10	11	12	13	9D	85	08	87	18	19	92	8F	1C	1D	1E	1F
	2	80	81	82	83	84	0A	17	1B	88	89	8A	8B	8C	05	06	07
	3	90	91	16	93	94	95	96	04	98	99	9A	9B	14	15	9E	1A
	4	20	A0	A1	A2	A3	A4	A5	A6	A7	A8	5B	2E	3C	28	2B	21
	5	26	A9	AA	AB	AC	AD	AE	AF	B0	B1	5D	24	2A	29	3B	5E
	6	2D	2F	B2	B3	B4	B5	B6	B7	B8	B9	7C	2C	25	5F	3E	3F
	7	BA	BB	BC	BD	BE	BF	C0	C1	C2	60	3A	23	40	27	3D	22
	8	C3	61	62	63	64	65	66	67	68	69	C4	C5	C6	C7	C8	C9
	9	CA	6A	6B	6C	6D	6E	6F	70	71	72	CB	CC	CD	CE	CF	D0
	A	D1	7E	73	74	75	76	77	78	79	7A	D2	D3	D4	D5	D6	D7
	B	D8	D9	DA	DB	DC	DD	DE	DF	E0	E1	E2	E3	E4	E5	E6	E7
	C	7B	41	42	43	44	45	46	47	48	49	E8	E9	EA	EB	EC	ED
	D	7D	4A	4B	4C	4D	4E	4F	50	51	52	EE	EF	F0	F1	F2	F3
	E	5C	9F	53	54	55	56	57	58	59	5A	F4	F5	F6	F7	F8	F9
	F	30	31	32	33	34	35	36	37	38	39	FA	FB	FC	FD	FE	FF

Figure 164. EBCDIC to ASCII

SO_BROADCAST

Enables a program to send broadcast messages over the socket to destinations that can receive datagram messages. By default, this option is disabled. The *optvalue* parameter must be one of the following values: 0 (disabled) or 1 (enabled). This option returns 0 if it is successfully completed; otherwise, it returns the error number.

Restriction: This option is not valid for stream sockets.

SO_DEBUG

(REXX only) Control whether debugging information is recorded. By default, this option is disabled. The *optvalue* parameter must be either ON (enabled) or OFF (disabled). This option returns 0 if it is successfully completed; otherwise, it returns the error number. This option is valid only for stream sockets.

SO_DONTROUTE

Bypasses normal routing algorithms for outgoing packets on the socket. If the local interface cannot be determined, when a packet is sent using one of the SEND commands, then the 51 ENETUNREACH error number is returned. This option returns either 1 (enabled) or 0 (disabled).

SO_EBCDIC

(REXX only) Enables data to be translated to and from EBCDIC. This option is ignored by EBCDIC hosts. The *optvalue* parameter must be either ON (enabled) or OFF (disabled). This option returns a string that contains the error code and either ON (enabled) or OFF (disabled). If the option is enabled, the name of the translation table is returned also. The following string is an example of what might be returned:

```
0 ON MYTRANB
```

SO_KEEPAIVE

Sets the keep alive mechanism to periodically send a packet on an otherwise idle connection for a stream socket. By default, this option is disabled. When enabled, if the remote TCP/IP does not respond to the packet or to retransmissions of the packet, then the connection is terminated with the ETIMEDOUT error. The *optvalue* parameter must be one of the following values: 0 (disabled) or 1 (enabled). This option returns 0 if it is successfully completed; otherwise, it returns the error number.

Tip: You also can enable the keep alive mechanism by modifying the TCPIP.PROFILE file.

SO_LINGER

Specifies how TCP/IP processes data that has not been transmitted when the CLOSE command is issued for the socket. When this option is enabled and the CLOSE command is issued, the calling program is blocked until either the data is successfully transmitted or the connection times out. When this option is disabled and the CLOSE command is issued, the CLOSE command returns without blocking the caller; then, TCP/IP continues to attempt to send data for a specified time, which usually provides sufficient time to complete the data transfer. By default, this option is disabled.

The *optvalue* parameter is a string in the following format:

```
linger = "onoff lingertime"
```

where

- The *onoff* value is either 0 (disabled) or 1 (enabled).
- The *lingertime* value is the number of seconds that TCP/IP tries to send data after the CLOSE command is issued.

This option returns 0 if it is successfully completed; otherwise, it returns the error number.

Restrictions:

- Using the SO_LINGER option does not guarantee that a data transfer will be completed, because TCP/IP waits only the amount of time that is specified.
- This option is valid only for stream sockets.

Guidelines:

- Avoid setting a linger time of 0. If you set the linger time to 0, the connection stops rather than closing in an orderly manner. This results in a RESET segment being sent to the connection partner. If the aborting socket is in nonblocking mode, the CLOSE command is processed as though no linger option is set.
- Enable the SO_LINGER option only when necessary.

SO_OOBINLINE

Controls whether out-of-band data is received. When this option is enabled, out-of-band data is placed in the normal data input queue as it is received; this data is then available to RECV or RECVFROM commands, even if the OOB flag is not set. When this option is disabled, out-of-band data is placed in the priority data input queue as it is received; this data is then available to RECV or RECVFROM commands only if the OOB flag is set. By default, this option is disabled. The *optvalue* parameter must be one of the following values: 0 (disabled) or 1 (enabled). This option returns 0 if it is successfully completed; otherwise, it returns the error number.

Restriction: This option is valid only for stream sockets.

SO_RCVBUF

Controls the size of the data portion of the TCP/IP receive buffer. The size of the receive buffer is protocol specific and is based on the following values:

- (TCP socket) The TCPRCVBufsize keyword on the TCPCONFIG statement in the PROFILE.TCPIP data set.
- (UDP socket) The UDPRCVBufsize keyword on the UDPCONFIG statement in the PROFILE.TCPIP data set.
- (Raw socket) The default size 65535.

The *optvalue* parameter must be either 0 (disabled) or a positive integer that specifies the size of the TCP/IP receive buffer. If you disable this option, the default system setting is used. This option returns 0 if it is successfully completed; otherwise, it returns the error number.

SO_RCVTIMEO

Sets the maximum length of time that a receive-type function can wait before it completes. You can specify the number of seconds and microseconds that indicate the length of time to wait for a receive-type function to complete. If a receive-type function has blocked for this much time without receiving data, it returns with the errno EWOULDBLOCK. The value 0 (the default) indicates that a receive-type function does not time out. The *optvalue* parameter must be a string that contains the number of seconds followed by the number of microseconds. The GETSOCKOPT command returns the return code or error number. Specify a value in the range 0 – 2 678 400 (equal to 31 days) for the number of seconds. Specify a value in the range 0 – 1 000 000 (equal to 1 second) for the number of microseconds. Although you can specify the number of microseconds, the internal TCP/IP timers that are used to implement this function have a granularity of approximately 100 milliseconds. The following receive type commands are supported:

- READ
- RECV
- RECVFROM

SO_REUSEADDR

Controls whether local addresses are reused. Enabling this option alters the normal algorithm that is used with the BIND command. The normal BIND algorithm permits each Internet address and port combination to be bound only once. If the address and port already have been bound, a subsequent BIND command fails with the 48 EADDRINUSE error. When this option is enabled, the following situations are supported:

- A server can bind the same port multiple times, if each invocation uses a different local IP address and the wildcard address INADDR_ANY is used only one time for each port.

- A server with active client connections can be restarted and can bind to its port without having to close all of the client connections.
- For datagram sockets, multicasting is supported so that multiple BIND commands can be made to the same class D address and port number.

By default, this option is disabled. The *optvalue* parameter must be one of the following values: 0 (disabled) or 1 (enabled). This option returns 0 if it is successfully completed; otherwise, it returns the error number.

Tip: If you want to permit multiple servers to bind to INADDR_ANY or IN6ADDR_ANY and listen on the same port, use the SHAREPORT option on the PORT statement in TCPIP.PROFILE.

SO_SNDBUF

Controls the size of the data portion of the TCP/IP send buffer. The size of the send buffer is protocol specific and is based on the following values:

- (TCP socket) The TCPSENDBufsize keyword on the TCPCONFIG statement in the PROFILE.TCPIP data set.
- (UDP socket) The UDPSENDBufsize keyword on the UDPCONFIG statement in the PROFILE.TCPIP data set.
- (Raw socket) The default size 65535.

The *optvalue* parameter must be either 0 (disabled) or a positive integer that specifies the size of the TCP/IP send buffer. If you disable this option, the default system setting is used. This option returns 0 if it is successfully completed; otherwise, it returns the error number.

SO_SNDTIMEO

Sets the maximum length of time that a send-type function can remain blocked before it completes. You can specify the number of seconds and microseconds that indicate the length of time to wait for a send-type function to complete. If a send-type function has blocked for this length of time, the function returns with a partial count or with an errno set to EWOULDBLOCK if no data is sent. The value 0 (the default) indicates that a send type function does not time out. The *optvalue* parameter must be a string that contains the number of seconds followed by the number of microseconds. The GETSOCKOPT command returns the return code or error number. Specify a value in the range 0 – 2 678 400 (equal to 31 days) for the number of seconds. Specify a value in the range 0 – 1 000 000 (equal to 1 second) for the number of microseconds. While the number of microseconds can be specified, the internal TCP/IP timers that are used to implement this function have a granularity of approximately 100 milliseconds. The following send-type commands are supported:

- SEND
- SENDTO
- WRITE

TCP_KEEPALIVE

Specifies whether a socket-specific timeout value (in seconds) is used instead of a configuration-specific value, when keep alive timing is active for the socket. When enabled, the socket-specific timeout value remains in effect until either the socket is closed or it is reset by a SETSOCKOPT command. The *optvalue* parameter must be either 0 (disabled) or the keep alive value. This option returns a string that contains the return code and the keep alive value. If the option is disabled, the keep alive value is 0. For more information about the socket option parameters, see [TCP_KeepAlive socket option in z/OS Communications Server: IP Programmer's Guide and Reference](#).

Tip: The site administrator can enable the global keep-alive mechanism by specifying the INTERVAL parameter on the TCPCONFIG statement in the TCP/IP stack profile data set, TCPIP.PROFILE.

TCP_NODELAY

Specifies whether the data that is sent over the socket is subject to the Nagle algorithm (RFC 896). When this option is enabled, TCP waits to send small amounts of data until the acknowledgment

for the previous data sent is received. When this option is disabled, TCP sends data when it is presented. The *optvalue* parameter must be one of the following values: 0 (disabled) or 1 (enabled). This option returns 0 if it is successfully completed; otherwise, it returns the error number.

optvalue

Additional information that is needed to run the requested command.

Returned value

The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

See [Appendix B, “Socket call error return codes,” on page 743](#) for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 9 EBADF
- 22 EINVAL
- 38 ENOTSOCK
- 42 ENOPROTOOPT
- 45 EOPNOTSUPP
- 60 ETIMEDOUT

The following REXX socket API error numbers can be returned:

- 2001 EINVALIDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE
- 2009 ESOCKETNOTDEFINED
- 2012 EINVALIDNAME

LE C/C++ equivalent

```
int setsockopt(int socket, int level, int option_name, char *option_value,
               int *option_length);
```

Code example

See the EZARXS01 REXX sample in the SEZAINST file for an example of using the SETSOCKOPT command.

SHUTDOWN

Use the SHUTDOWN command to shut down all or part of a duplex connection.

Format

➤ SOCKET — (— "SHUTDOWN" — , — *socketid* — { — , — BOTH — } — { — , — how — } —) ➤

Parameters

socketid

The socket descriptor of the connected socket.

how

Specifies which operations are to be ended. The following parameters are supported:

BOTH

Ends further send and receive operation on the socket. By default, this is the value of the *how* parameter. The value 2 is supported also.

SEND

Ends further send operations on the socket. The following values are supported also: 1, TO, SENDING, WRITE, WRITING.

READ

Ends further receive operations on the socket. The following values are supported also: 0, FROM, READING, RECEIVE, RECEIVING.

Returned value

The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

See Appendix B, “Socket call error return codes,” on page 743 for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 9 EBADF
- 22 EINVAL
- 38 ENOTSOCK
- 45 EOPNOTSUPP

The following REXX socket API error numbers can be returned:

- 2001 EINVALDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE
- 2009 ESOCKETNOTDEFINED
- 2012 EINVALIDNAME

LE C/C++ equivalent

```
long shutdown(int *s, int how);
```

Code example

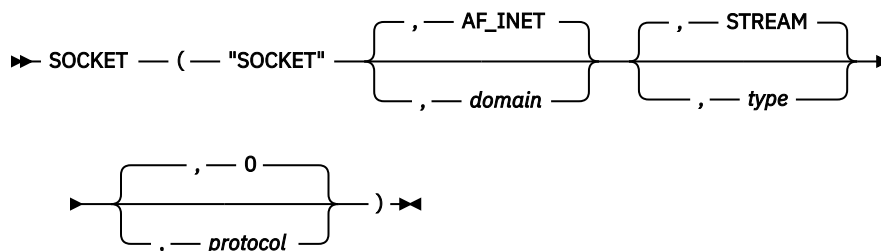
See the EZARXS02 REXX sample in the SEZAINST file for an example of using the SHUTDOWN command.

SOCKET

Use the SOCKET command to open a socket descriptor in the active socket set.

Restriction: If the socket type is SOCK_RAW or RAW, the user ID associated with the REXX socket application must have z/OS UNIX System Services superuser authority. The user ID must have the UID value 0 or have read access to the BPX.SUPERUSER security profile. An application can attempt to obtain superuser authority by issuing the z/OS UNIX System Services SYSCALLS command: address syscall 'SETEUID 0'. If this command fails, the user ID does not have the authorization needed to run the program; contact your security administrator.

Format



Parameters

domain

The address family of the socket. The supported families are AF_INET (2) and AF_INET6 (19). By default, the *domain* parameter is set to AF_INET.

type

An optional parameter that specifies the type of socket to be created. By default, this parameter is set to STREAM. The following values are supported:

- STREAM or SOCK_STREAM
- DATAGRAM or SOCK_DGRAM
- RAW or SOCK_RAW

protocol

An optional parameter that specifies the protocol that is requested. By default, the value to which this parameter is set depends on the *type* parameter. The default protocol for stream sockets is TCP. The default protocol for datagram sockets is UDP. There is no default for RAW sockets. To enable the stack to select the applicable protocol, set the *protocol* parameter to 0.

The following protocols are supported:

Stream sockets

IPPROTO_TCP or TCP

Datagram sockets

IPPROTO_UDP or UDP

RAW sockets

- IPPROTO_IP or IP
- IPPROTO_IPV6 or IPV6
- IPPROTO_ICMP or ICMP
- IPPROTO_ICMPV6 or ICMPV6
- IPPROTO_RAW or RAW

Returned value

The command returns a string that contains the return code and the new socket descriptor, for example, 0 6. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

See [Appendix B, "Socket call error return codes,"](#) on page 743 for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 9 EBADF
- 22 EINVAL

- 38 ENOTSOCK
- 45 EOPNOTSUPP
- 139 EPERM

The following REXX socket API error numbers can be returned:

- 2001 EINVALDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE
- 2007 EMAXSOCKETSREACHED

LE C/C++ equivalent

```
int socket(int *domain, int type, int protocol);
```

Code example

```
/* REXX EZARXR26 */
/*
 * This sample demonstrates the use of the SOCKET
 * socket command.
 *
 * HINT: See other socket command descriptions for
 * additional examples.
 */
src=SOCKET("INITIALIZE","MYSET01",20);
src=SOCKET("SOCKET","AF_INET","STREAM")
parse var src l_retcode l_socketid
if l_retcode = 0 then do
  Say "Socket Created Successfully. Socket descriptor is" l_socketid;
  src=SOCKET("CLOSE",l_socketid);
  parse var src l_retcode .
  if l_retcode = 0 then
    say "Socket "l_socketid" closed successfully";
  else do
    say "Close of socket "l_socketid" failed.";
    say src;
  end;
end;
else do
  Say "Socket not created."
  Say "... "src;
end;
x=SOCKET("TERMINATE","MYSET01");
exit;
```

Figure 165. SOCKET command example

SOCKETSET

Use the SOCKETSET command to retrieve the name of the active socket set. If you specify the name of a socket set as a parameter, then that socket set becomes the active socket set.

Format

```
➡ SOCKET — ( — "SOCKETSET" ————— ) ➡
               | , — subtaskid |
```

Parameters

subtaskid

The name of a socket set that was created with the INITIALIZE command

Returned value

The command returns a string that contains the return code and the name of the active socket set, for example, 0 MYSET. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

See [Appendix B, “Socket call error return codes,” on page 743](#) for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error number can be returned:

- None

The following REXX socket API error number can be returned:

- 2005 ESUBTASKNOTACTIVE

LE C/C++ equivalent

This command has no LE C/C++ equivalent.

Code example

```
/* REXX EZARXR27 */
/*
 * This sample demonstrates the use of the SOCKETSET
 * socket command.
 */
src=socket("INITIALIZE", "MYSET");
src=Socket("SOCKETSET");
Say src;
src=socket("TERMINATE", "MYSET");
exit 0;
```

Figure 166. SOCKETSET command example

SOCKETSETLIST

Use the SOCKETSETLIST command to list the names of all available socket sets that are currently defined by the application.

Rule: All sockets sets are created with the INITIALIZE command.

Format

➤ SOCKET — (— "SOCKETSETLIST" —) ➤

Parameters

This command has no parameters.

Returned value

If socket sets are defined, this command returns a string that contains 0 and names of the socket sets that are available to the application, for example, 0 subtask1 subtask2. If no socket sets are defined, 0 is returned.

LE C/C++ equivalent

This command has no LE C/C++ equivalent.

Code example

```

/* REXX EZARXR28 */
/*
 * This program demonstrates the use of the SOCKETLIST
 * socket command. The program will initialize two socket
 * sets and then issue the SOCKETLIST command to obtain
 * some information.
 */
src1=SOCKET("INITIALIZE","MYSET01",10);
Say "RC of INITIALIZATION of MYSET01 = "src1"
src2=SOCKET("INITIALIZE","MYSET02",29);
Say "RC of INITIALIZATION of MYSET02 = "src2"
src=SOCKET("SOCKETSETLIST");
parse var src l_retcode l_socketsets;
Say "Socket sets available are: "l_socketsets;
src=SOCKET("TERMINATE","MYSET01");
src=SOCKET("TERMINATE","MYSET02");
EXIT 0;

```

Figure 167. SOCKETSETLIST command example

SOCKETSETSTATUS

Use the `SOCKETSETSTATUS` command to list information about a socket set.

Format

```

➤ SOCKET — ( — "SOCKETSETSTATUS" — , — subtaskid ) ➤

```

Parameters

subtaskid

An optional parameter that specifies the name of the socket set. If this parameter is not specified, then the `SOCKETSETSTATUS` command returns the status of the active socket set.

Returned value

The command returns a string that contains the return code, the name of the socket set, the state of the socket set, the number of sockets that are available for use, and the number of sockets that currently are being used. The return code can be 0 or the REXX API error number. The return code 0 indicates that the requested socket command was completed successfully.

The state of a socket set is one of the following values:

CONNECTED

Indicates that the socket set has been initialized with the INITIALIZE command

SEVERED

Indicates that the socket set has been initialized using the INITIALIZE command, but a problem exists with the socket set

FREE

Indicates that the socket set has not been initialized

The following string is an example of what is returned by the SOCKETSETSTATUS command:

```
0 MYSET Connected Free 15 Used 1
```

In the example, 0 is the return code, MYSET is the name of the socket set, CONNECTED is the status of the socket set, 15 is the number of the free sockets, and 1 is the number of sockets that are in use.

See [Appendix B, “Socket call error return codes,” on page 743](#) for additional information about the numeric error codes that are returned by this command.

The following REXX socket API error number can be returned::

- 2005 ESUBTASKNOTACTIVE

LE C/C++ equivalent

This command has no LE C/C++ equivalent.

Code example

```
/* REXX EZARXR29 */
/*
 * This sample demonstrates the use of the SOCKETSETSTATUS
 * socket command. Error checking is not performed for the
 * socket commands as the intent is to show what the command
 * will return when multiple INITIALIZE commands are
 * issued.
 */
x1=socket("INITIALIZE","MYTEST",15);
x2=socket("SOCKETSET");
x3=socket("SOCKETSETSTATUS");
x4=socket("SOCKET","AF_INET");
parse var x4 l_retcode l_x4_socketid;
if l_retcode = 0 then do
  x5=socket("SOCKETSETSTATUS");
  Say "FIRST INITIALIZE command:"
  Say "  INITIALIZE      = "x1;
  Say "  SOCKETSET       = "x2;
  Say "  SOCKETSETSTATUS  = "x3;
  Say "  SOCKET          = "l_retcode l_x4_socketid;
  Say "  SOCKETSETSTATUS = "x5;
  Say "*** END";
  y1=socket("INITIALIZE","REXXSET",15);
  y2=socket("SOCKETSET");
  y3=socket("SOCKETSETSTATUS");
  y4=socket("SOCKETSETSTATUS","MYTEST");
  y5=socket("SOCKET","AF_INET");
  parse var y5 l_retcode l_y5_socketid;
  if l_retcode = 0 then do
    y6=socket("SOCKETSETSTATUS");
    Say "SECOND INITIALIZE command:"
    Say "  INITIALIZE      = "y1;
    Say "  SOCKETSET       = "y2;
    Say "  SOCKETSETSTATUS  = "y3;
    Say "  SOCKETSETSTATUS  = "y4;
    Say "  SOCKET          = "l_retcode l_y5_socketid;
    Say "  SOCKETSETSTATUS = "y6;
    Say "*** END";
    rc2=socket("CLOSE",l_y5_socketid);
    say "rc2 = "rc2;
    xx=socket("SOCKETSETSTATUS","MYTEST");
  end;
  rc1=socket("CLOSE",l_x4_socketid);
  Say "rc1 = "rc1;
end;
x=socket("TERMINATE","MYTEST");
x=socket("TERMINATE","REXXSET");
exit;
```

Figure 168. SOCKETSETSTATUS command example

TAKESOCKET

Use the TAKESOCKET command to take a socket descriptor that is passed from another program using the GIVESOCKET command. A socket descriptor can be taken by an application only when the socket is in the same address family.

Guidelines: An application that issues the TAKESOCKET command needs to know both the client ID of the application that issued the GIVESOCKET command and the socket descriptor that was passed. REXX provides several techniques that can be used to pass this information to the application that issues the TAKESOCKET command:

- When the application that issues the GIVESOCKET command also will be the application to issue the TAKESOCKET command, the client ID and socket descriptor can be passed between the routines that

are responsible for the two commands using standard REXX programming techniques. In this situation, consider setting the socket to nonblocking mode to permit additional socket processing to occur as needed. Use the SELECT command to determine when a socket is ready or when an exception occurred. Use this technique primarily during application development. If the socket descriptor is never given to a different application or subtask, avoid using the GIVESOCKET or TAKESOCKET commands.

- When the application that will take the socket descriptor is running in a different address space, consider using the following methods to pass the information:
 - Pass the *clientid* value through the program startup parameters or by using an external input stream
 - Use the z/OS UNIX System Services SYSCALLS interface and named pipes

Format

➤ SOCKET — (— "TAKESOCKET" — , — *clientid* — , — *socketid* —) ➤

Parameters

clientid

Identifies the application that issued the GIVESOCKET command. The *clientid* parameter has the following format:

```
clientid = "domain jobname subtaskid"
```

All three fields are required:

domain

The address family of the socket. The supported families are AF_INET (2) and AF_INET6 (19).

jobname

The job name of the application that issued the GIVESOCKET command.

subtaskid

The name of the socket set used by the application that issued the GIVESOCKET command.

socketid

The socket descriptor that was given by the application that issued the GIVESOCKET command.

Returned value

The command returns a string that contains the return code and the socket descriptor, for example, 0 1. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

Tip: The 13 EACCESS return code indicates that the application that issued the TAKESOCKET command is not authorized to take the socket descriptor. The *jobname* field of the *clientid* parameter for the GIVESOCKET command must match the *jobname* field of the *clientid* parameter for the TAKESOCKET command.

See [Appendix B, "Socket call error return codes,"](#) on page 743 for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 9 EBADF
- 13 EACCESS
- 22 EINVAL
- 38 ENOTSOCK
- 45 EOPNOTSUPP

The following REXX socket API error numbers can be returned:

- 2001 EINVALDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE
- 2009 ESOCKETNOTDEFINED
- 2012 EINVALNAME

LE C/C++ equivalent

This command has no LE C/C++ equivalent.

Code example

See the EZARXS06 REXX sample in the SEZAINST file for an example of using the TAKESOCKET command.

TERMINATE

Use the TERMINATE command to close all sockets in the specified socket set and to release the socket set.

A socket set is a number of preallocated sockets available to a single application. You can define multiple socket sets for one session, but only one socket set can be active at a time. When the active socket set is released, the next socket set in the stack becomes the active socket set.

Format

```
➤ SOCKET — ( — "TERMINATE" — , — subtaskid — ) ➤
```

Parameters

subtaskid

An optional parameter that specifies the name of the socket set. If this parameter is not specified, then the active socket set is released.

Returned value

The command returns a string that contains the return code and the name of the socket set, for example, 0 MYTASKID. The return code can be 0 or the REXX API error number. The return code 0 indicates that the requested socket command was completed successfully.

See [Appendix B, “Socket call error return codes,” on page 743](#) for additional information about the numeric error codes that are returned by this command.

The following REXX socket API error numbers can be returned:

- 2001 EINVALDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE

LE C/C++ equivalent

This command has no LE C/C++ equivalent.

Code example

```
/* REXX EZARXR30 */
/*
 * This sample demonstrates the use of the TERMINATE
 * socket command.
 */
src = socket("INITIALIZE","MYSET01",10);
if word(src,1) = 0 then say "INITIALIZE SUCCESSFUL";
src = socket("TERMINATE", "MYSET01");
say "TERMINATE = " src;
exit 0;
```

Figure 169. TERMINATE command example

VERSION

Use the VERSION command to retrieve the name, version number, and version date of the REXX socket library.

Format

➡ SOCKET — (— "VERSION" —) ➡

Parameters

This command has no parameters.

Returned value

The command returns a string that contains the return code, version name, version number, and version date, for example, 0 REXX/SOCKETS z/OS V1R9 April 20, 2006. A return code of 0 indicates that the requested socket command was completed successfully.

LE C/C++ equivalent

This command has no LE C/C++ equivalent.

Code example

```
/* REXX EZARXR31 */
/*
 * This sample demonstrates the use of the VERSION
 * socket command.
 */
src = socket("VERSION");
say "Version = " src;
exit 0;
```

Figure 170. VERSION command example

WRITE

Use the WRITE command to send an outgoing message on the connected socket. The WRITE command is similar to the SEND command, except that the WRITE command does not support the control flags that are available with the SEND command.

When the socket is a TCP socket, the following conditions apply:

- If the socket is in blocking mode and the total amount of data to be sent cannot be processed by the stack when the command is issued, then the command blocks until the data can be sent.

- If the socket is in nonblocking mode and the total amount of data to be written cannot be processed by the stack when the command is issued, then the command returns the number of bytes that were successfully written. If none of the data can be written, the command returns the value -1 and the 35 EWOULDBLOCK error code.

When the socket is a connected UDP socket, the WRITE command either is completed or fails. A connected UDP socket does not return the 35 EWOULDBLOCK error code.

Restriction: The WRITE command does not support send flags.

Guidelines: Place the WRITE command in a loop to ensure that all the data is written. For a TCP socket, a partial write operation might occur regardless of whether the socket is in blocking or nonblocking mode. A partial write operation occurs when the stack copies some but not all of the application data:

- If a partial write operation occurs on a socket in blocking mode, the blocking socket is interrupted. The return value contains the number of bytes written, and the return code contains the reason for the interruption. In such cases, consider ending the connection.
- If a partial write operation occurs on a socket in nonblocking mode, the return value indicates the number of bytes that were successfully sent. If this is less than the number of bytes specified on the WRITE command, repeat the WRITE operation until all data is written. The blocking condition might last for a long time, so consider other strategies to ensure that the application does not remain in a busy loop sending data.

Tips:

- Use the SELECT command to determine whether a socket is ready to send additional data. To do so, test the socket for a WRITE event.
- If the SO_ASCII socket option is enabled, then the data received is translated from EBCDIC to ASCII.

Format

➤ SOCKET — (— "WRITE" — , — *socketid* — , — *data* —) ➤

Parameters

socketid

The socket descriptor

data

The string to be sent

Returned value

The command returns a string that contains the return code and the length of the data string, for example, 0 19. The return code can be 0, a REXX socket API error number, or the REXX TCP/IP error number that is set by the socket command. The return code 0 indicates that the requested socket command was completed successfully.

See [Appendix B, "Socket call error return codes," on page 743](#) for additional information about the numeric error codes that are returned by this command.

The following REXX TCP/IP error numbers can be returned:

- 4 EINTR
- 9 EBAADF
- 5 EIO
- 22 EINVAL
- 32 EPIPE
- 35 EWOULDBLOCK
- 38 ENOTSOCK

- 40 EMSGSIZE
- 45 EOPNOTSUPP
- 54 ECONNRESET
- 57 ENOTCONN

The following REXX socket API error numbers can be returned:

- 2001 EINVALIDRXSOCKETCALL
- 2005 ESUBTASKNOTACTIVE
- 2009 ESOCKETNOTDEFINED

LE C/C++ equivalent

```
ssize_t send(int socket, const void *buffer, size_t length, int flags);
```

Code example

See the SEND command. Substitute the command WRITE for the command SEND.

REXX socket sample programs

This section provides information about the sample programs that show how to use the REXX socket API. These programs are provided as is.

Overview of REXX sample programs

The sample programs and the jobs that you can use to run them are located in the SEZAINST file.

The following information applies to the batch jobs:

- The batch job REXXAPI runs standalone socket EXECs and TCP/IP clients.
- The batch job REXXAPIS runs TCP/IP servers.
- The batch job REXXAPIT runs the subtask that is required to test the REXAPI04 program.

Table 21. REXX socket sample programs				
Name	REXX EXEC	MVS batch job	APIs demonstrated	Notes
EZARXS01	REXAPI01	REXXAPI	INITIALIZE SOCKET GETSOCKOPT SETSOCKOPT CONNECT GETSOCKNAME SEND RECV TERMINATE	Client, requires server REXAPI05

Table 21. REXX socket sample programs (continued)

Name	REXX EXEC	MVS batch job	APIs demonstrated	Notes
EZARXS02	REXAPI02	REXXAPI	INITIALIZE SOCKET CONNECT GETSOCKNAME SEND RECV SHUTDOWN TERMINATE	Client, requires server REXAPI05
EZARXS03	REXAPI03	REXXAPI	INITIALIZE SOCKET CONNECT IOCTL SEND RECV CLOSE TERMINATE	Client, uses server REXAPI05 or REXAPI04 demonstrating AT-TLS
EZARXS04	REXAPI04	REXXAPIS	INITIALIZE SOCKET BIND LISTEN ACCEPT GETCLIENTID GIVESOCKET SELECT CLOSE	Server, not enabled for AT-TLS
EZARXS05	REXAPI05	REXXAPIS	INITIALIZE SOCKET BIND LISTEN ACCEPT IOCTL SEND RECV CLOSE TERMINATE	Server, supports AT-TLS
EZARXS06	REXAPI06	REXXAPIT	INITIALIZE TAKESOCKET GETPEERNAME RECV SEND CLOSE TERMINATE	Child server subtask

Table 21. REXX socket sample programs (continued)

Name	REXX EXEC	MVS batch job	APIs demonstrated	Notes
EZARXRSC	RSCLIENT	REXXAPI	INITIALIZE GETHOSTID SOCKET GETHOSTNAME CONNECT WRITE READ TERMINATE SOCKETSETSTATUS	IPv4 client
EZARXR6C	R6CLIENT	REXXAPI	INITIALIZE GETHOSTID SOCKET GETHOSTNAME GETADDRINFO CONNECT WRITE READ TERMINATE SOCKETSETSTATUS	IPv6 client
EZARXRSS	RSSERVER	REXXAPIS	INITIALIZE GETHOSTID SOCKET BIND LISTEN IOCTL SELECT ACCEPT RECV CLOSE SEND TERMINATE SETSOCKETSTATUS	IPv4 server
EZARXR6S	R6SERVER	REXXAPIS	INITIALIZE GETHOSTNAME GETADDRINFO SOCKET BIND LISTEN IOCTL SELECT ACCEPT GETNAMEINFO RECV CLOSE SEND TERMINATE SETSOCKETSTATUS	IPv6 server

Table 21. REXX socket sample programs (continued)				
Name	REXX EXEC	MVS batch job	APIs demonstrated	Notes
EZARXJ01	n/a	REXXAPI	MVS sample job control	MVS job control for REXXAPI
EZARXJ02	n/a	REXXAPIS	MVS sample job control	MVS job control for REXXAPIS
EZARXJ03	n/a	REXXAPIT	MVS sample job control	MVS job control for REXXAPIT

The REXX-EXEC RSCLIENT sample program for IPv4

The client sample program is a REXX socket program that shows you how to use the commands that are provided by the REXX sockets API. The program connects to the server sample program and receives data, which is displayed on the screen. It uses sockets in blocking mode.

After parsing and testing the input parameters, the RSCLIENT EXEC program obtains a socket set using the INITIALIZE command and a socket using the SOCKET command. The program then connects to the server and writes the user ID, the node ID, and the number of lines requested on the connection to the server. It reads data in a loop; the data is displayed on the screen until the data length is 0, which indicates that the server has closed the connection. If an error occurs, the client program lists the return code, determines the status of the socket set, and ends the socket set.

The server adds the EBCDIC new-line character to the end of each record, and the client uses this character to determine the start of a new record. If the connection is abnormally closed, the records that were partially received are not displayed.

The REXX-EXEC RSSERVER sample program for IPv4

The server sample program shows an example of how to use sockets in nonblocking mode. The program waits for connect requests from client programs, accepts the requests, and then sends the data. The sample can handle multiple client requests in parallel processing.

The server program sets up a socket to accept connection requests from clients and then waits in a loop for events reported by the SELECT command. If a socket event occurs, it is processed. A read event can occur on the original socket for accepting connection requests and on sockets for accepted socket requests. A write event can occur only on sockets for accepted socket requests.

A read event on the original socket for connection requests means that a connection request from a client occurred. Read events on other sockets indicate that there is either data to receive or that the client has closed the socket. Write events indicate that the server can send more data. The server program sends only one line of data in response to a write event.

The server program keeps a list of sockets to which it wants to write. It keeps this list to avoid unwanted socket events. The TCP/IP protocol is not designed for one single-threaded program communicating on many different sockets, but for multithread applications where one thread processes only the events that originate from a single socket.

The REXX-EXEC R6CLIENT sample program for IPv6

The client sample program is a REXX socket program that shows you how to use the commands that are provided by the REXX sockets API. The program connects to the server sample program and receives data, which is displayed on the screen. It uses sockets in blocking mode.

After parsing and testing the input parameters, the R6CLIENT EXEC sample program obtains a socket set using the INITIALIZE command and a socket using the SOCKET command. The program then connects to the server and writes the user ID, the node ID, and the number of lines requested on the connection to the server. It reads data in a loop; the data is displayed on the screen until the data length is 0, which

indicates that the server has closed the connection. If an error occurs, the client program lists the return code, determines the status of the socket set, and ends the socket set.

The server adds the EBCDIC new-line character to the end of each record, and the client uses this character to determine the start of a new record. If the connection is abnormally closed, the partially received records are not displayed.

The REXX-EXEC R6SERVER sample program for IPv6

The server sample program shows an example of how to use sockets in nonblocking mode. The program waits for connect requests from client programs, accepts the requests, and then sends data. The sample can handle multiple client requests in parallel processing.

The server program sets up a socket to accept connection requests from clients and waits in a loop for events that are reported by the SELECT command. If a socket event occurs, it is processed. A read event can occur on the original socket for accepting connection requests and on sockets for accepted socket requests. A write event can occur only on sockets for accepted socket requests.

A read event on the original socket for connection requests means that a connection request from a client occurred. Read events on other sockets indicate that there is either data to receive or that the client has closed the socket. Write events indicate that the server can send more data. The server program sends only one line of data in response to a write event.

The server program keeps a list of sockets to which it wants to write. It keeps this list to avoid unwanted socket events. The protocol is not designed for one single-threaded program communicating on many different sockets, but for multithread applications where one thread processes events only from a single socket.

AT-TLS security definitions for REXX samples

When Application Transparent Transport Layer Security (AT-TLS) is enabled on the TCP/IP stack, you must define the REXX sample applications to the Policy Agent.

The user must supply the necessary key rings. For information about creating certificates and key rings, see the TLS/SSL security information in *z/OS Communications Server: IP Configuration Guide*. These AT-TLS security definitions assume that the REXXAPI and REXXAPIS sample jobs are being used.

Running the REXX sample programs

This topic describes how to run the REXX sample programs.

Procedure

To run the REXX sample programs, complete the following steps:

1. Uncomment the MVS job control EXEC card.
2. Run the REXX EXEC.

Testing the GIVESOCKET and TAKESOCKET commands

This topic describes how to test the GIVESOCKET and TAKESOCKET commands using the sample programs.

Procedure

To test the GIVESOCKET and TAKESOCKET commands, complete the following steps:

1. Uncomment REXAPI04 from the job REXXAPIS and submit the sample job control to the JES reader.
2. Submit job REXXAPI using REXAPI03.
3. Submit REXXAPIT.

REXXAPIT assumes that the socket being passed is 3 and that the client ID is AF_INET6 REXAPI TCPSVT.

Chapter 15. Pascal application programming interface

This information describes the Pascal language for IPv4 socket application program interface (API) that is provided with TCP/IP. This interface allows programmers to write application programs that use the TCP, UDP, and IP layers of the TCP/IP protocol suite. Topics include:

- Software requirements
- Data structures
- Using procedure calls
- Pascal return codes
- Procedure calls
- Sample Pascal program

To use the Pascal language API, you should have experience in Pascal language programming and be familiar with the principles of internetwork communication.

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

Note: The Pascal API is not enhanced for IPv6 support.

Steps for Pascal language API procedure calls

This topic describes the steps of writing the Pascal program.

Before you begin

To use the Pascal language API, you should have experience in Pascal language programming and be familiar with the principles of internetwork communication.

Procedure

Perform the following steps to write the Pascal program.

1. Start TCP/UDP/IP service (BeginTcpIp).

-
2. Specify the set of notifications that TCP/UDP/IP can send you (Handle).

-
3. Establish a connection (TcpOpen, UdpOpen, RawIpOpen, and TcpWaitOpen).

Note: If using TcpOpen, communication must wait for the appropriate notification of connection.

-
4. Transfer a data buffer to or from the TCP/IP address space (TcpSend, TcpFSend, TcpWaitSend, TcpReceive, TcpFReceive, TcpWaitReceive, UdpSend, UdpNReceive, RawIpSend, UdpReceive, and RawIpReceive).

Notes:

- a. TcpWaitReceive and TcpWaitSend are synchronous calls.
- b. TcpFSend and TcpSend are the asynchronous ways of sending data on a TCP connection. Both procedures return to your program immediately. TcpSend does not wait under any circumstance.

- c. TcpSend and TcpFSend differ in how they handle the situation when TCP/IP address space has insufficient buffer space to accept the data being sent.
- d. In the case of insufficient buffer space, TCP/IP responds to TcpSend with the return code NObufferSPACE. This return code is sent back to the application. It is the application's responsibility to wait for BUFFERspaceAVAILABLE notification and resend the data.
- e. In the case of TcpFSend with insufficient buffer space, the PASCAL API blocks until buffer space becomes available or an error is detected. This is the only condition under which TcpFSend blocks.

5. Check the status returned from TCP/IP in the form of notifications (GetNextNote).

6. Repeat the data transfer operations (Steps 4 and 5) until the data is exhausted.

7. Terminate the connection (TcpClose, UdpClose, and RawIpClose).

Note: If you are using TcpClose, you must wait for the connection to terminate.

8. Terminate the communication service (EndTcpIp).

Results

You know you are done when control is returned to you. Control is returned, in most instances, after the initiation of your request. When appropriate, some procedures have alternative wait versions that return only after request completion.

Example

A sample program is supplied with TCP/IP. See [“Sample Pascal program” on page 728](#), for a listing of the sample program.

Pascal language API software requirements

To develop programs in Pascal that interface directly to the TCP, UDP, and IP protocol boundaries, you must have the IBM VS Pascal Compiler and Library (5668-767).

Pascal API header files

The following list shows the headers used by Pascal applications:

- cmclien
- cmcomm
- cminter
- cmresglb

Pascal language API compatibility considerations

Unless noted in [z/OS Communications Server: New Function Summary](#), an application program that is compiled and link edited on a release of z/OS Communications Server IP can be used on higher level releases. That is, the API is upward compatible.

Application programs that are compiled and link edited on a release of z/OS Communications Server IP cannot be used on older releases. That is, the API is not downward compatible.

Pascal language API data structures

Programs containing Pascal language API calls must include the appropriate data structures. The data structures are declared in CMCOMM and CMCLIEN. To include these data sets in your program source, enter:

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

Additional include statements are required in programs that use certain calls. The following list shows the members 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 SEZACMTX data set. Include this data set in your SYSLIB concatenation when you are creating a load module to link an application program. You must specify SEZACMTX before the Pascal libraries when linking TCP/IP programs.

Connection state

ConnectionState is the current state of the connection. See [Figure 171 on page 689](#) for the Pascal declaration of the ConnectionStateType data type. ConnectionStateType is used in StatusInfoType and NotificationInfoType. It defines the client program view of the state of a TCP connection, in a form more readily usable than the formal TCP connection state defined by RFC 793. See [Table 22 on page 690](#) for the mapping between TCP states and ConnectionStateType.

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

Figure 171. 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 22 on page 690 lists the TCP connection states.

Table 22. 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, otherwise CONNECTIONclosing.
CLOSE-WAIT	If there is incoming data that the client program has not received, then OPEN, otherwise SENDINGonly.
ESTABLISHED	OPEN
FIN-WAIT-1, FIN-WAIT-2	RECEIVINGonly
LISTEN	LISTENING
SYN-SENT, SYN-RECEIVED	TRYINGtoOPEN

Connection information record

The connection information record is used as a parameter in several of the procedure calls. It enables you and the TCP/IP program to exchange information about the connection. The Pascal declaration is shown in [Figure 172 on page 690](#).

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

Figure 172. Pascal declaration of connection information record

Connection

A number identifying the connection that is described. This connection number is different from the connection number displayed by the NETSTAT command.

OpenAttemptTimeout

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.

BytesToRead

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

UnackedBytes

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

The current state of the connection. `ConnectionStateType` defines the client program view of the state of a TCP connection, in a form more readily usable than the formal TCP connection state defined by RFC 793.

LocalSocket

The local internet address and local port. Together, these form one end of a connection. The foreign socket forms the other end. See [Figure 173 on page 691](#) for the Pascal declaration of the `SocketType` record.

ForeignSocket

The foreign, or remote, internet address and its associated port. These form one end of a connection. The local socket forms the other end. [Figure 173 on page 691](#) shows the Pascal declaration of a socket type.

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

Figure 173. Pascal declaration of socket type

Address

The internet address.

Port

The port.

Notification record

The notification record is used to provide event information. You receive this information by using the `GetNextNote` call. If it is a variant record, the number of fields depends on the type of notification. See [Figure 174 on page 692](#) for the Pascal declaration of this record.

```

NotificationInfoType =
  record
    Connection: ConnectionType;
    Protocol: ProtocolType;
    case NotificationTag: NotificationEnumType of
      BUFFERspaceAVAILABLE:
        (
          AmountOfSpaceInBytes: integer
        );
      CONNECTIONstateCHANGED:
        (
          NewState: ConnectionStateType;
          Reason: CallReturnCodeType
        );
      DATAdelivered:
        (
          BytesDelivered: integer;
          LastUrgentByte: integer;
          PushFlag: Boolean
        );
    );

```

```

FSENDresponse:
  (
    SendTurnCode: CallReturnCodeType;
    SendRequestErr: Boolean;
  );
PINGresponse:
  (
    PingTurnCode: CallReturnCodeType;
    ElapsedTime: TimeStampType
  );
RAWIPpacketsDELIVERED:
  (
    RawIpDataLength: integer;
    RawIpFullLength: integer;
  );
RAWIPspaceAVAILABLE:
  (
    RawIpSpaceInBytes: integer;
  );
SMSGreceived: ();
TIMERexpired:
  (
    Datum: integer;
    AssociatedTimer: TimerPointerType
  );
UDPDatagramDELIVERED:
  (
    DataLength: integer;
    ForeignSocket: SocketType;
    FullLength: integer
  );
UDPDatagramSPACEavailable: ();
URGENTpending:
  (
    BytesToRead: integer;
    UrgentSpan: integer
  );
USERdefinedNOTIFICATION:
  (
    UserData: UserNotificationDataType
  );
end;

```

Figure 174. Notification record

Connection

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

The type of notification being sent. A set of fields depends 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.

AmountOfSpaceInBytes

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

CONNECTIONstateCHANGED

Indicates that a TCP connection has changed state.

NewState

The new state for this connection.

Reason

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

Notes:

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

- For active open:
 - OPEN
 - RECEIVINGonly or SENDINGonly
 - CONNECTIONclosing
 - NONEXISTENT
- For passive open:
 - 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. Reason codes giving the reason for a connection changing to NONEXISTENT are:

- OK
- UNREACHABLEnetwork
- TIMEOUTopen
- OPENrejected
- REMOTEreset
- WRONGsecORprc
- FATALerror
- TCPipSHUTDOWN

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

Number of bytes of data delivered to you.

LastUrgentByte

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

PushFlag

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

FSENDresponse

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

SendTurnCode

The status of the send operation.

PINGresponse

Notification given when a PINGresponse is received.

PingTurnCode

The status of the PING operation.

ElapsedTime

The time elapsed between the sending of a request and the reception of a response. This field is valid only if PingTurnCode has a value of OK.

RAWIPpacketsDELIVERED

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

RawIpDataLength

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

RawIpFullLength

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

RAWIPspaceAVAILABLE

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

RawIpSpaceInBytes

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

RESOURCESavailable

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

MSGreceived

Notification given when one or more special messages (Smsgs) arrive. The GetSmsg call is used to retrieve queued Smsgs.

TIMERexpired

Notification given when a timer set through SetTimer expires.

Datum

The data specified when SetTimer was called.

AssociatedTimer

The address of the timer that expired.

UDPdatagramDELIVERED

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

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 your buffer was too small to contain the entire datagram.

DataLength

Length of the data delivered to your buffer.

ForeignSocket

The source of the datagram.

FullLength

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

UDPdatagramSPACEavailable

Notification given when buffer space becomes available for a datagram for which UdpSend previously returned NObufferSPACE because of insufficient resources.

URGENTpending

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

BytesToRead

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

UrgentSpan

Number of bytes that are not delivered 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. Connection and protocol fields also are set from the values supplied to AddUserNote.

File specification record

The file specification record is used to fully specify a data set. The Pascal declaration is shown in [Figure 175 on page 695](#).

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

Figure 175. Pascal declaration of file specification record

Pascal language API: using procedure calls

Your program uses procedure calls to initiate communication with the TCP/IP address space. Most of these procedure calls return with a code, which indicates success or the type of failure incurred by the call. See [Table 23 on page 697](#) for an explanation of the return codes.

Before invoking any of the other interface procedures, use `BeginTcpIp` to start the TCP/UDP/IP service. When 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 TCP/IP address space notifies you of asynchronous events. Also, some notifications are generated in your address space by the TCP interface. Notifications can be received only after `BeginTcIp`.

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. See [Figure 174 on page 692](#) for more information.

TCP initialization procedures

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

TCP termination procedure

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

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 the `BeginTcpIp` initialization routine before you can begin using TCP communication procedures.

PING interface

The Ping interface lets a client send an ICMP echo request to a foreign host. You must call the `BeginTcpIp` initialization routine before you can begin using the PING Interface.

Monitor procedures

The `MonQuery` monitor procedure provides a mechanism for querying the TCP/IP address space.

Any program using this monitor procedure must include `CMCOMM` and `CMCLIEN`.

UDP communication procedures

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

Raw IP interface

The Raw IP interface lets a client program send and receive arbitrary IP datagrams on any IP Internet protocol except TCP and UDP. Only one client can use any given protocol at one time. Only clients that are APF-authorized can use the Raw IP interface.

Timer routines

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

Host lookup routines

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

Assembler calls

`AddUserNote` is provided and can be called directly from an assembler language interrupt handler.

Other routines

This group includes the following procedures.

- `GetSmsg`
- `ReadXlateTable`
- `SayCalRe`
- `SayConSt`
- `SayIntAd`
- `SayIntNum`
- `SayNotEn`
- `SayPorTy`
- `SayProTy`

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 712](#)) to convert the `ReturnCode` parameter to a printable form.

The `SayCalRe` function converts a return value into a descriptive message. For example, if `SayCalRe` is invoked with the return value `BADlengthARGUMENT`, it returns the message `invalid length specified`. See [Table 23 on page 697](#) for a description of Pascal return codes and their equivalent message text from `SayCalRe`.

Most return values 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.

Table 23. Pascal language return codes

Return Value	Return Code	Message Text
OK	0	OK.
ABNORMALcondition	-1	Abnormal condition because of CSA storage shortage.
ALREADYclosing	-2	Connection is already closing.
BADlengthARGUMENT	-3	Length specified that is not valid.
CANNOTsendDATA	-4	Cannot send data.

Table 23. Pascal language return codes (continued)

Return Value	Return Code	Message Text
CLIENTrestart	-5	Client reinitialized TCP/IP service.
CONNECTIONalreadyEXISTS	-7	Connection exists.
ERRORinPROFILE	-8	Error in profile data set. Details are in PROFILE.TCPERROR or the //SYSERROR DD file.
FATALerror	-9	Fatal error; not valid user parameter (storage key).
HASnoPASSWORD	-10	No password is in the RACF directory.
INCORRECTpassword	-11	TCPIP is not authorized to access the data set.
INVALIDrequest	-12	Request not valid.
INVALIDuserID	-13	User ID not valid.
INVALIDvirtualADDRESS	-14	Virtual address not valid.
LOCALportNOTavailable	-16	The requested local port is not available.
NObufferSPACE	-19	No more space for data currently available. This applies to this connection only; space might still be available for other connections.
NONlocalADDRESS	-21	The internet address is not local to this host.
NOoutstandingNOTIFICATIONS	-22	No outstanding notifications.
NOsuchCONNECTION	-23	No such connection.
NOtcpIPservice	-24	No TCP/IP service is available.
NOTyetBEGUN	-25	TCP/IP service not yet begun.
NOTyetOPEN	-26	The connection is not yet open.
OPENrejected	-27	Foreign host rejected the open attempt.
PARAMlocalADDRESS	-28	TcpOpen error: local address not valid.
PARAMstate	-29	TcpOpen error: initial state not valid.
PARAMtimeout	-30	Timeout parameter not valid.
PARAMunspecADDRESS	-31	TcpOpen error: unspecified foreign address in active open.
PARAMunspecPORT	-32	TcpOpen error: unspecified foreign port in active open.
PROFILEnotFOUND	-33	TCPIP cannot read PROFILE data set.
RECEIVEstillPENDING	-34	Receive is still pending on this connection.
REMOTEclose	-35	Foreign host unexpectedly closed the connection.
REMOtereset	-36	Foreign host abended the connection.
SOFTWAREerror	-37	Software error in TCP/IP.
TCPipSHUTDOWN	-38	TCP/IP is being shut down.

Table 23. Pascal language return codes (continued)

Return Value	Return Code	Message Text
TIMEOUTopen	-40	Foreign host did not respond within OPEN timeout.
TOOmanyOPENS	-41	Too many open connections exist.
UNAUTHORIZEDuser	-43	You are not authorized to issue this command.
UNIMPLEMENTEDrequest	-45	TCP/IP request not implemented.
UNREACHABLEnetwork	-47	Destination network cannot be reached.
UNSPECIFIEDconnection	-48	Connection not specified.
VIRTUALmemoryTOOsmall	-49	Client address space has too little storage.
WRONGsecORprc	-50	Foreign host disagreed on security or precedence.
ZEROresources	-56	TCP cannot handle more connections now.
UDPlocalADDRESS	-57	Local address for UDP not correct.
UDPunspecADDRESS	-59	Address was not specified; specification is necessary.
UDPunspecPORT	-60	Port was unspecified; specification is necessary.
FSENDstillPENDING	-62	FSend still pending on this connection.
ERRORopeningORreadingFILE	-80	Error opening or reading data set.
FILEformatINVALID	-81	File format is not valid.
SAYCALRE*	-130	Unknown TCP return code.

* Return codes that are not valid (out of the range -128 to 0) return Unknown TCP return codes when translated using SAYCALRE.

Pascal language API procedure calls

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

AddUserNote

This procedure can be called from assembler language code to add a USERdefinedNOTIFICATION notification to the note queue and cause the initiation of GetNextNote if it is waiting for a notification. [Figure 176 on page 700](#) shows a sample calling sequence.

	LA	R13,PASCSAVE	
	LA	R1,PASCPARM	
	L	R15,=V(ADDUSERN)	
	BALR	R14,R15	
		.	
PASCSAVE	DS	18F	Register save area
ENV	DC	F'0'	Zero initially. It is filled with an environment address. Pass it unchanged in subsequent calls to ADDUSERN.
DATA1	DS	H	Data for Connection field of notification.
DATA2	DS	C	Data for Protocol field of notification.
DATA3	DS	XL40	Data for UserData field of notification.
RC	DS	F	AddUserNote stores return code here.
PASCPARM	DC	A(ENV)	
	DC	A(DATA1)	
	DC	A(DATA2)	
	DC	A(DATA3)	
	DC	A(RC)	

Figure 176. Sample calling sequence

Parameter Description

ReturnCode (RC)

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

- OK
- NObufferSPACE

BeginTcpIp

Use BeginTcpIp to inform the TCP/IP address space that you want to start using its services as show in [Figure 177 on page 700](#).

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

Figure 177. BeginTcpIp example

Parameter Description

ReturnCode

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

- OK
- ABNORMALcondition
- FATALerror
- NOTcpIPservice
- TCPipSHUTDOWN
- VIRTUALmemoryTOOsmall

For a description of the Pascal return codes, see [Table 23 on page 697](#).

ClearTimer

This procedure resets the timer to prevent it timing out as shown in [Figure 178 on page 701](#).


```

procedure ClearTimer
(
    T: TimerPointerType
);
external;

```

Figure 178. ClearTimer example

Parameter Description

T
A timer pointer, as returned by a previous CreateTimer call.

CreateTimer

This procedure allocates a timer. The timer is not set in any way. See [“SetTimer” on page 714](#) to activate the timer. [Figure 179 on page 701](#) shows an example.

```

procedure CreateTimer
(
    var T: TimerPointerType
);
external;

```

Figure 179. Create timer example

Parameter Description

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

DestroyTimer

This procedure deallocates (frees) a timer you created. [Figure 180 on page 701](#) shows an example.

```

procedure DestroyTimer
(
    var T: TimerPointerType
);
external;

```

Figure 180. Destroy timer example

Parameter Description

T
A timer pointer, as returned by a previous CreateTimer call.

EndTcpIp

Use EndTcpIp when you have finished with the TCP/IP services. The procedure shown in [Figure 181 on page 701](#) releases ports and protocols in use that are not permanently reserved. It causes TCP to clean up the data structures it has associated with your commands.

```

procedure EndTcpIp;
external;

```

Figure 181. EndTcpIp example

GetHostNumber

The GetHostNumber procedure resolves a host name into an internet address. This is shown in [Figure 182 on page 702](#).

GetHostNumber uses a table lookup to convert the name of a host (alphanumeric name or dotted decimal number) to an internet address, and returns this address in 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 1 decimal integer for each of the 4 bytes, separated by dots. For example, 14.0.0.7 for X'0E000007'. See [z/OS Communications Server: IP Configuration Reference](#) for information about how to create host lookup tables.

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

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

Figure 182. GetHostNumber example

Parameter Description

Name

The name or dotted decimal number to be converted. The maximum name length is 128 characters.

HostNumber

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

GetHostResol

The GetHostResol procedure converts a host name into an internet address by using a name server. [Figure 183 on page 702](#) shows an example.

GetHostResol passes the query to the remote name server through the resolver. The name server converts the name of a host (alphanumeric name or dotted decimal number) 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 1 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;
```

Figure 183. GetHostResol example

Parameter Description

Name

The name or dotted decimal number to be converted. The maximum length is 255 characters.

HostNumber

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

GetHostString

The GetHostString procedure call uses a table lookup to convert an internet address dotted decimal format 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. An example is shown in [Figure 184 on page 703](#).

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

Figure 184. GetHostString example

Parameter Description

Address

The address to be converted. The address must be in dotted decimal format.

Name

Set to the corresponding host, gateway, or network name, or to null string if a match is not found. The maximum length is 24 characters.

GetIdentity

This procedure returns the following information:

- The user ID of the MVS user
- The host machine name
- The network domain name
- The user ID of the TCP/IP address space

The host machine name and domain name are extracted from the HostName and DomainOrigin statements, respectively, in TCPIP.DATA. If a HostName statement is not specified, then the default host machine name is the name specified by the TCP/IP installer during installation (the name from the line containing the definition, VMCF,MVPXSSI,nodename, in the IEFSSNxx member of PARMLIB). The TCP/IP address space user ID is extracted from the TcpipUserid/TcpipJobname statement in TCPIP.DATA; if the statement is not specified, the default is TCPIP. See [z/OS Communications Server: IP Configuration Reference](#) for information about TCPIP.DATA search order.

[Figure 185 on page 703](#) shows the GetIdentity procedure.

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

Figure 185. GetIdentity example

Parameter Description

UserId

The user ID of the TSO user or the job name of a batch job that has invoked GetIdentity.

HostName

The host machine name.

DomainName

The network domain name.

TcpIpServiceName

The user ID of the TCP/IP address space.

Result

Indicates success or failure of the call.

GetNextNote

Use this procedure to retrieve notifications from the queue. This procedure returns the next notification queued for you. [Figure 186 on page 704](#) shows an example of the GetNextNote procedure.

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

Figure 186. GetNextNote example

Parameter**Description****Note**

Next notification is stored here when ReturnCode is OK.

ShouldWait

Set ShouldWait to TRUE if you want GetNextNote to wait until a notification becomes available. Set ShouldWait to FALSE if you want GetNextNote to return immediately. When ShouldWait is set to FALSE, ReturnCode is set to NOoutstandingNOTIFICATIONS if notification is not currently queued.

ReturnCode

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

- OK
- NOoutstandingNOTIFICATIONS
- NOTyetBEGUN

For a description of Pascal return codes, see [Table 23 on page 697](#).

GetSmsg

Your program should call this procedure after receiving an SMSGreceived notification. Each call to GetSmsg retrieves one queued Smsg. Your program should exhaust all queued Smsgs by calling GetSmsg repeatedly until the Success field returns with a value of FALSE. After a value of FALSE is returned, do not call GetSmsg again until you receive another SMSGreceived notification. [Figure 187 on page 704](#) shows an example of the GetSmsg procedure.

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

Figure 187. GetSmsg example

Parameter**Description****Smsg**

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

Success

If Smsg returned TRUE; otherwise FALSE.

Handle

Use the Handle procedure to specify that you want to receive notifications in the given set as shown in [Figure 188 on page 705](#). You must always use it after calling the BeginTcpIp procedure and before accessing the TCP/IP services. This Pascal set of notifications can contain any of the NotificationEnumType values shown in [Figure 174 on page 692](#).

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

Figure 188. Handle example

Parameter

Description

Notifications

The set of notification types to be handled.

ReturnCode

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

- OK
- NOTyetBEGUN

For a description of Pascal return codes, see [Table 23 on page 697](#).

IsLocalAddress

This procedure queries the TCP/IP address space to determine whether the HostAddress is one of the addresses recognized for this host. If the address is local, it returns OK. If the address is not local, it returns NONlocalADDRESS. [Figure 189 on page 705](#) shows an example.

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

Figure 189. IsLocalAddress example

Parameter

Description

HostAddress

The host address to be tested.

ReturnCode

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

- OK
- NONlocalADDRESS
- TCPIPshutdown
- FATALerror
- SOFTWAREerror

For a description of Pascal return codes, see [Table 23 on page 697](#).

IsLocalHost

This procedure returns the correct host class for Name, which can be a host name or a dotted decimal address. [Figure 190 on page 706](#) shows an example of the IsLocalHost procedure.

The host classes are:

HOSTlocal

An internet address for the local host

HOSTloopback

One of the dummy internet addresses used to designate various levels of loopback testing

HOSTremote

A known host name for some remote host

HOSTunknown

An unknown host name (or other error)

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

Figure 190. IsLocalHost example

Parameter

Description

Name

The host name. The maximum name length is 255 characters.

Class

The host class

MonQuery

The MonQuery procedure is used to obtain status information or to request TCP/IP to perform certain actions.

Restriction: When you use this function, the total number of IPv4 IP addresses that can be configured to the TCP/IP stack is limited to 255 IP addresses. This limitation of 255 IP addresses applies to all IPv4 IP addresses, including loopback and dynamic VIPA addresses.

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

Figure 191. MonQuery example

Parameter

Description

Buffer

The address of the buffer to receive data.

BufSize

The size of the buffer.

ReturnCode

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

- OK
- FATALerror
- NOTyetBEGUN
- TCPIPshutdown
- UNIMPLEMENTEDrequest
- UNAUTHORIZEDuser
- SOFTWAREerror

Length

The length of the data returned in the buffer.

QueryRecord

Your program sets up a QueryRecord to specify the type of status information to be retrieved. The MonQueryRecordType is shown in [Figure 192 on page 707](#).

```
MonQueryRecordType =
  record
    case QueryType: MonQueryType of
      QUERYhomeONLY: ();
    end; { MonQueryRecordType }
```

Figure 192. Monitor query record

The only QueryType values available for customer use is:

QUERYhomeONLY

Used to obtain a list of the home Internet addresses (up to 255) recognized by TCP/IP. 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 the InternetAddressType to get the number of the home addresses that are returned.

For a description of Pascal return codes, see [Table 23 on page 697](#).

PingRequest

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

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

Figure 193. PingRequest example

Parameter

Description

ForeignAddress

The address of the foreign host to receive an ICMP echo request.

Length

Indicates the length of the ICMP packet, excluding the IP header. The range of values for this field is 8 - 65507 bytes.

Timeout

The amount of time to wait for a response, in seconds.

ReturnCode

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

- OK
- ABNORMALcondition
- BADlengthARGUMENT
- CONNECTIONalreadyEXISTS
- VIRTUALmemoryTOOsmall
- NOTyetBEGUN
- TIMEOUTopen
- PARAMtimeout
- SOFTWAREerror
- TCPipSHUTDOWN
- UNAUTHORIZEDuser

For a description of Pascal return codes, see [Table 23 on page 697](#).

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

RawIpClose

This procedure tells the TCP/IP address space that the client does not handle the protocol any longer. Any queued incoming packets are discarded. [Figure 194 on page 708](#) shows an example of the RawIpClose procedure.

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

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

Figure 194. RawIpClose example

Parameter Description

ProtocolNo

The number of the Internet protocol.

ReturnCode

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

- OK
- NOSuchCONNECTION
- NOTyetBEGUN
- SOFTWAREerror
- TCPipSHUTDOWN
- UNAUTHORIZEDuser

For a description of Pascal return codes, see [Table 23 on page 697](#).

RawIpOpen

This procedure tells the TCP/IP address space that the client wants to send and receive packets of the specified protocol. [Figure 195 on page 709](#) shows an example.

Do not 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 address space, you receive the LOCALportNOTavailable return code.

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

Figure 195. *RawIpOpen* example

Parameter	Description
-----------	-------------

ProtocolNo	The number of the Internet protocol.
-------------------	--------------------------------------

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

- OK
- LOCALportNOTavailable
- NObufferSPACE
- NOTyetBEGUN
- SOFTWAREerror
- TCPipSHUTDOWN
- UNAUTHORIZEDuser

For a description of Pascal return codes, see [Table 23 on page 697](#).

Note: You can open the ICMP protocol, but your program receives only those ICMP packets not interpreted by the TCP/IP address space.

RawIpReceive

Use the procedure shown in Figure 196 on page 709 to specify a buffer to receive Raw IP datagrams 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;
```

Figure 196. *RawIpReceive* example

Parameter	Description
-----------	-------------

ProtocolNo	The number of the Internet protocol.
-------------------	--------------------------------------

Buffer	The address of your buffer.
---------------	-----------------------------

BufferLength	The length of your buffer. If you specify a length greater than 65535 bytes, only the first 65535 bytes are used.
---------------------	---

ReturnCode

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

- OK
- NOSuchCONNECTION
- NOTyetBEGUN
- SOFTWAREerror
- TCPIPshutdown
- UNAUTHORIZEDuser
- INVALIDvirtualADDRESS

For a description of Pascal return codes, see [Table 23 on page 697](#).

RawIpSend

This procedure shown in this example sends IP datagrams of the given protocol number. The entire packet, including the IP header, must be in the buffer. The TCP/IP address space uses the total length field of the IP header to determine where each packet ends. Subsequent packets begin at the next doubleword (eight-byte) boundary within the buffer.

The packets in your buffer are transmitted unchanged 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 fewer than 40 bytes, or greater than 65535 bytes.
- NumPackets is 0.
- All packets do not fit into DataLength.

A ReturnCode value of NObufferSPACE indicates that the data is rejected, because TCP/IP 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;
```

Figure 197. RawIpSend example

Parameter Description

ProtocolNo

The number of the Internet protocol.

Buffer

The address of your buffer containing packets to send.

DataLength

The total length of data in your buffer.

NumPackets

The number of packets in your buffer.

ReturnCode

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

- OK
- BADlengthARGUMENT
- NObufferSPACE
- NOSuchCONNECTION
- NOTyetBEGUN
- SOFTWAREerror
- TCPipSHUTDOWN
- UNAUTHORIZEDuser
- INVALIDvirtualADDRESS

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

For a description of Pascal return codes, see [Table 23 on page 697](#).

ReadXlateTable

The procedure shown in [Figure 198 on page 711](#) reads the binary translation table data set specified by TableName, and fills in the AtoETable and EtoATable translation tables.

```

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

```

Figure 198. ReadXlateTable example

Parameter	Description
-----------	-------------

TableName

The name of the translate table. ReadXlateTable tries to read *user_id*.TableName.TCPXLBIN. If that data set exists but it has an incorrect format, ReadXlateTable returns with a ReturnCode FILEformatINVALID. If *user_id*.TableName.TCPXLBIN does not exist, ReadXlateTable tries to read *hlq*.TableName.TCPXLBIN. ReturnCode reflects the status of reading that data set.

AtoETable

Filled with ASCII-to-EBCDIC table if return code is OK.

EtoATable

Filled with EBCDIC-to-ASCII table if return code is OK.

TranslateTableSpec

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

ReturnCode

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

- OK
- ERRORopeningORreadingFILE

- FILEformatINVALID

SayCalRe

This function returns a printable string describing the return code passed in CallReturn. [Figure 199 on page 712](#) shows an example.

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

Figure 199. SayCalRe example

Parameter Description

CallReturn

The return code to be described.

SayConSt

This function returns a printable string describing the connection state passed in State. For example, if SayConSt is invoked with the type identifier RECEIVINGonly, it returns the message Receiving only. [Figure 200 on page 712](#) shows an example of this procedure.

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

Figure 200. SayConSt example

Parameter Description

State

The connection state to be described.

SayIntAd

This function converts the Internet Protocol address specified by InternetAddress to a printable string. If the address can be resolved to a name by use of local host tables, the name is returned. The address to name resolution depends on how the resolver is configured and if any local host tables exist. See [z/OS Communications Server: IP Configuration Guide](#) for information about configuring the resolver and how local host tables can be used. If the address cannot be resolved to a name, the dotted decimal format of the address is returned. [Figure 201 on page 712](#) shows an example of this procedure.

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

Figure 201. SayIntAd example

Parameter Description

InternetAddress

The internet address to be converted.

SayIntNum

This function converts the internet address specified by `InternetAddress` to a printable string, in dotted decimal form as shown in [Figure 202 on page 713](#).

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

Figure 202. SayIntNum example

Parameter Description

InternetAddress

The internet address to be converted.

SayNotEn

This function returns a printable string describing the notification enumeration type passed in `Notification`. For example, if `SayNotEn` is invoked with the type identifier `FSENDreponse`, it returns the message "Fsend response".

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

Figure 203. SayNotEn example

Parameter Description

Notification

The notification enumeration type to be described.

SayPorTy

This function returns a printable string describing the port number passed in `Port`, if it is a well-known port number such as port number 23, the Telnet port. Otherwise, the EBCDIC representation of the number is returned. [Figure 204 on page 713](#) shows an example of this function.

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

Figure 204. SayPorTy example

Parameter Description

Port

The port number to be described.

SayProTy

This function converts the protocol type specified by Protocol to a printable string, if it is a well-known protocol number, such as 6 (TCP). Otherwise, the EBCDIC representation of the number is returned. [Figure 205 on page 714](#) shows an example of this function.

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

Figure 205. SayProTy example

Parameter	Description
-----------	-------------

Protocol	The number of the protocol to be described.
-----------------	---

SetTimer

The procedure shown in [Figure 206 on page 714](#) 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;
```

Figure 206. SetTimer example

Parameter	Description
-----------	-------------

T	A timer pointer, as returned by a previous <code>CreateTimer</code> call.
----------	---

AmountOfTime	The time interval in seconds.
---------------------	-------------------------------

Data	An integer value to be returned with the <code>TIMERexpired</code> notification.
-------------	--

TcpAbort

Use the procedure shown in [Figure 207 on page 714](#) to shut down a specific connection immediately. Data sent by your application on the abended connection might be lost. TCP sends a reset packet to notify the foreign host that you have abended 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;
```

Figure 207. TcpAbort example

Parameter
Description

Connection

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

ReturnCode

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

- OK
- ABNORMALcondition
- FATALerror
- NOsuchCONNECTION
- NOTyetBEGUN
- TCPipSHUTDOWN
- SOFTWAREerror
- REMOTEreset

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

For a description of Pascal return codes, see [Table 23 on page 697](#).

TcpClose

Use the procedure shown in [Figure 208 on page 715](#) to begin the TCP one-way closing sequence. During this closing sequence, you, the local client, cannot send any more data. Data might 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;
```

Figure 208. TcpClose example

Parameter
Description

Connection

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

ReturnCode

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

- OK
- ABNORMALcondition
- ALREADYclosing
- NOsuchCONNECTION
- NOTyetBEGUN
- TCPipSHUTDOWN
- SOFTWAREerror
- REMOTEreset

For a description of Pascal return codes, see [Table 23 on page 697](#).

Notes:

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

TcpFReceive, TcpReceive, and TcpWaitReceive

The examples in this material illustrate 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. The return code OK means that the request has been accepted. When received data has been placed in your buffer, your program receives a DATAdelivered notification.

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 0, the parameter indicates an error.

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

Figure 209. TcpFReceive example

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

Figure 210. TcpReceive example

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

Figure 211. TcpWaitReceive example

Parameter Description

Connection

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

Buffer

The address of the buffer to contain the received data.

BytesToRead

The size of the buffer. TCP/IP 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.

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

BytesRead

Set when TcpWaitReceive returns. If it is greater than 0, it indicates the number of bytes received into your buffer. If it is less than or equal to 0, it indicates an error. Possible BytesRead values are:

- OK+
- ABNORMALcondition
- FATALerror
- TIMEOUTopen+
- UNREACHABLEnetwork+
- BADlengthARGUMENT
- NOSuchCONNECTION
- NOTyetBEGUN
- NOTyetOPEN
- OPENrejected+
- RECEIVEstillPENDING
- REMOTEreset+
- TCPipSHUTDOWN+
- REMOTEclose

ReturnCode

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

- OK
- ABNORMALcondition
- BADlengthARGUMENT
- FATALerror
- NOSuchCONNECTION
- NOTyetBEGUN
- NOTyetOPEN
- RECEIVEstillPENDING
- REMOTEclose
- TCPipSHUTDOWN
- INVALIDvirtualADDRESS
- SOFTWAREerror

For a description of Pascal return codes, see [Table 23 on page 697](#).

(TcpWaitReceive):

1. For BytesRead OK, the function was initiated, but the connection is no longer receiving for an unspecified reason. Your program does not have to issue TcpClose, but the connection is not completely terminated until a NONEXISTENT notification is received for the connection.
2. For BytesRead REMOTEclose, the foreign host has closed the connection. Your program should respond with TcpClose.

3. If you receive any of the codes marked with (+), the function was initiated but the connection has now been terminated (see “2” on page 693). 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 multiple connection 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

The examples in this material illustrate `TcpFSend`, `TcpSend`, and `TcpWaitSend`.

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

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

In the case of insufficient buffer space, when space becomes available a `BUFFERspaceAVAILABLE` notification is received.

Your program can issue successive `TcpWaitSend` calls. Buffer shortage conditions are handled transparently. Errors at this point are most likely unable to recover or are caused by a terminated connection.

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;
```

Figure 212. TcpFSend example

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

Figure 213. TcpSend example

```

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

```

Figure 214. *TcpWaitSend* example

Parameter Description

Connection

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

Buffer

The address of the buffer containing the data to send.

BufferLength

The size of the buffer.

PushFlag

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

UrgentFlag

Is set to mark the data as *urgent*. The semantics of urgent data depends on your application.

Note: Use urgent data with caution. If the foreign application follows the Telnet-style use of urgent data, it might flush all urgent data, until a special character sequence is encountered.

ReturnCode

Indicates success or failure of call:

- OK
- ABNORMALcondition
- BADlengthARGUMENT
- CANNOTsendDATA
- FATALerror
- NObufferSPACE (*TcpSend* and *TcpFSend*)
- NOSuchCONNECTION
- NOTyetBEGUN
- NOTyetOPEN
- TCPipSHUTDOWN
- INVALIDvirtualADDRESS
- SOFTWAREerror
- REMOTEreset

For a description of Pascal return codes, see [Table 23 on page 697](#).

Notes:

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

3. The PushFlag is used to expedite 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 use 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 put data into packets 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 multiple connection servers.

TcpNameChange

Use the procedure shown in [Figure 215 on page 720](#) if the address space running the TCP/IP program is not named TCPIP and is not the same as specified in the TcpipUserid statement of the TCPIP.DATA data set. (See [z/OS Communications Server: IP Configuration Reference](#).)

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

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

Figure 215. TcpNameChange example

Parameter

Description

NewNameOfTcp

The name of the address space running TCP/IP.

TcpOpen and TcpWaitOpen

The examples in this material illustrate TcpOpen and TcpWaitOpen.

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

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

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

Figure 216. TcpOpen example

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

Figure 217. TcpWaitOpen example

Parameter**Description****ConnectionInfo**

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 (TCP/IP uses the home address corresponding to the network interface used to route to the foreign address) and a port of UNSPECIFIEDport (TCP/IP assigns a port number, in the range of 1000 - 65535). 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: You usually specify a predetermined port number, known by another program, which can do an active open to connect to your program. Alternatively, you can use UNSPECIFIEDport to let TCP/IP assign a port number, obtain the port number through TcpStatus, and transmit it to the other program through an existing TCP connection or manually. You generally specify an address of UNSPECIFIEDaddress, so that the active open to your port succeeds, regardless of the home address to which it was sent.

ForeignSocket

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

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

ReturnCode

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

- OK
- ABNORMALcondition
- FATALerror
- LOCALportNOTavailable
- NObufferSPACE
- NOSuchCONNECTION
- NOTyetBEGUN
- OPENrejected (TcpWaitOpen Only)

- PARAMlocalADDRESS
- PARAMstate
- PARAMtimeout
- PARAMunspecADDRESS
- PARAMunspecPORT
- REMOTEreset (TcpWaitOpen Only)
- SOFTWAREerror
- TCPIPshutdown
- TIMEOUTopen (TcpWaitOpen Only)
- TOOMANYOPENS
- UNAUTHORIZEDuser (TcpWaitOpen Only)
- UNREACHABLEnetwork (TcpWaitOpen Only)
- ZEROresources

For a description of Pascal return codes, see [Table 23 on page 697](#).

TcpOption

Use the procedure shown in [Figure 218 on page 722](#) to set an option for a TCP connection.

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

Figure 218. TcpOption example

Parameter Description

Connection

The connection number, as returned by TcpOpen or TcpWaitOpen in the Connection field of the StatusInforType record.

OptionName

The code for the option.

Name Description

OPTIONtcpKEEPALIVE

If OptionValue is nonzero, then the keep-alive mechanism is activated for connection. If OptionValue is 0, then the keep-alive mechanism is deactivated for the connection. When activated, the keep-alive mechanism periodically sends a packet on an otherwise idle connection. If the remote TCP does not respond to the packet or to retransmissions of the packet, then the connection state is changed to NONEXISTENT, with reason TIMEOUT connection.

OptionValue

The value for the option.

ReturnCode

Indicates success or failure of call.

Possible return values are:

- OK

- NOsuchCONNECTION
- NOTyetBEGUN
- TCPIPshutdown
- INVALIDrequest
- SOFTWAREerror
- REMOTEreset

For a description of Pascal return codes, see [Table 23 on page 697](#).

TcpStatus

Use `TcpStatus` to obtain the current status of a TCP connection. Your program sets the `Connection` field of the `ConnectionInfo` record to the number of the connection whose status you want. [Figure 219 on page 723](#) shows an example of `TcpStatus`.

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

Figure 219. TcpStatus example

Parameter

Description

ConnectionInfo

If `ReturnCode` is OK, the following fields are returned.

OpenAttemptTimeout

If the connection is in the process of being opened (including a passive open), this field is set to the number of seconds remaining before the open is terminated if it has not completed. Otherwise, it is set to `WAITforever`.

BytesToRead

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

UnackedBytes

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

The current connection state.

LocalSocket

The local socket, consisting of a local address and a local port.

ForeignSocket

The foreign socket, consisting of a foreign address and a foreign port.

ReturnCode

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

- OK
- NOsuchCONNECTION
- NOTyetBEGUN
- TCPIPshutdown
- REMOTEreset
- SOFTWAREerror

For a description of Pascal return codes, see [Table 23 on page 697](#).

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.

UdpClose

The procedure shown in [Figure 220 on page 724](#) closes the UDP socket specified in the `ConnIndex` field. All incoming datagrams on this connection are discarded.

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

Figure 220. UdpClose example

Parameter Description

ConnIndex

The `ConnIndex` value returned from `UdpOpen`.

ReturnCode

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

- OK
- NOSuchCONNECTION
- NOTyetBEGUN
- TCPIPshutdown
- SOFTWAREerror

For a description of Pascal return codes, see [Table 23 on page 697](#).

UdpNReceive

The procedure shown in [Figure 221 on page 724](#) notifies the TCP/IP address space that you are willing to receive UDP datagram data. This call returns immediately. The data buffer is not valid until you receive a `UDPdatagramDELIVERED` notification.

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

Figure 221. UdpNReceive example

Parameter Description

ConnIndex

The `ConnIndex` value returned from `UdpOpen`.

BufferAddress

The address of your buffer that is filled with a UDP datagram.

BufferLength

The length of your buffer. If you specify a length larger than 65507 bytes, only the first 65507 bytes are used.

ReturnCode

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

- OK
- ABNORMALcondition
- FATALerror
- NOsuchCONNECTION
- NOTyetBEGUN
- RECEIVestillPENDING
- TCPIPshutdown
- SOFTWAREerror
- BADlengthARGUMENT
- INVALIDvirtualADDRESS

For a description of Pascal return codes, see [Table 23 on page 697](#).

UdpOpen

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

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. [Figure 222 on page 725](#) shows an example.

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

Figure 222. UdpOpen example

Parameter Description

LocalSocket

The local socket (address and port pair).

ConnIndex

The ConnIndex value returned from UdpOpen.

ReturnCode

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

- OK
- ABNORMALcondition
- FATALerror
- LOCALportNOTavailable
- NObufferSPACE
- NOTyetBEGUN
- SOFTWAREerror
- TCPIPshutdown
- UDPLocalADDRESS

- TOOManyOPENS
- UNAUTHORIZEDuser

For a description of Pascal return codes, see [Table 23 on page 697](#).

UdpReceive

The procedure shown in [Figure 223 on page 726](#) notifies the TCP/IP address space that you are willing to receive UDP datagram data.

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

If you use UdpReceive, TCP/IP can put a datagram as large as 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 TCP/IP address space in this call. It only tells TCP/IP 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;

```

Figure 223. UdpReceive example

Parameter Description

ConnIndex

The ConnIndex value returned from UdpOpen.

DatagramAddress

The address of your buffer that is filled with a UDP datagram.

ReturnCode

Indicates success or failure of a call:

- OK
- ABNORMALcondition
- FATALerror
- NOsuchCONNECTION
- NOTyetBEGUN
- SOFTWAREerror
- TCPipSHUTDOWN
- INVALIDvirtualADDRESS

For a description of Pascal return codes, see [Table 23 on page 697](#).

UdpSend

The procedure shown in [Figure 224 on page 727](#) 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 TCP/IP.

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;

```

Figure 224. *UdpSend* example

Parameter Description

ConnIndex

The ConnIndex value returned from UdpOpen.

ForeignSocket

The foreign socket (address and port) to which the datagram is to be sent.

BufferAddress

The address of your buffer containing the UDP datagram to be sent, excluding UDP header.

Length

The length of the datagram to be sent, excluding UDP header. Maximum is 65507 bytes.

ReturnCode

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

- OK
- BADlengthARGUMENT
- NObufferSPACE
- NOSuchCONNECTION
- NOTyetBEGUN
- SOFTWAREError
- TCPIPshutdown
- UDPUnspecADDRESS
- UDPUnspecPORT
- INVALIDvirtualADDRESS

For a description of Pascal return codes, see [Table 23 on page 697](#).

Unhandle

Use the procedure shown in [Figure 225 on page 727](#) when you no longer want to receive notifications in the given set.

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

```

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

```

Figure 225. *Unhandle* example

Parameter Description

Notifications

The set of notifications that you no longer want to receive.

ReturnCode

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

- OK
- NOTyetBEGUN
- INVALIDrequest

For a description of Pascal return codes, see [Table 23 on page 697](#).

Sample Pascal program

This information contains an example of a Pascal application program. The source code can be found in the SEZAINST data set.

Building the sample Pascal API module

This topic describes the steps of building a sample Pascal API module.

Procedure

The following steps describe how to build the Pascal API module:

1. Compile the sample Pascal program.
2. Link-edit the object code module to form an executable module sample.

Running the sample module

This topic describes the steps of running a sample Pascal API module.

Procedure

The following steps describe how to run the sample module:

1. Run the Pascal API sample program with the Receive option (shown in [Figure 226 on page 728](#)).

Run PSAMPLE to start the sample program on the TSO command line. The following example is a typical response:

```
ENTER TSO COMMAND, CLIST, OR REXX EXEC BELOW:

===> psample

Transfer Mode: (Send or Receive) receive

Host Name or Internet Address: mvs1

mvs1
Transfer rate 483884. Bytes/sec.
Transfer rate 442064. Bytes/sec.
Transfer rate 478802. Bytes/sec.
Transfer rate 549568. Bytes/sec.
Transfer rate 635116. Bytes/sec.
Program terminated successfully.
***
```

Figure 226. Sample Pascal API with receive option

2. Run the Pascal API sample program with the Send option on a second TSO ID (shown in [Figure 227 on page 729](#)).

Run PSAMPLE on the TSO command line to start the sample program. The following example is a typical response:

ENTER TSO COMMAND, CLIST, OR REXX EXEC BELOW:

==> psample

Transfer Mode: (Send or Receive) **send**

Host Name or Internet Address: **mvs1**

mvs1
Transfer rate 516540. Bytes/sec.
Transfer rate 487030. Bytes/sec.
Transfer rate 427816. Bytes/sec.
Transfer rate 566186. Bytes/sec.
Transfer rate 612128. Bytes/sec.
Program terminated successfully.

Figure 227. Sample Pascal API with send option

Sample Pascal application program

The following example shows a Pascal application program.

```
%UHEADER 5647-A01 (C) IBM CORP 1991, 2002. &SYSPARM EZABB01S PSAMPLE
{
  TCP/IP for MVS
  SMP/E Distribution Name: EZABB01V (for PSAMPLE source in SEZAINST)
                        EZABB01S (for PSAMPLE module in SEZAMOD1)

  Licensed Materials - Property of IBM
  This product contains "Restricted Materials of IBM"
  5694-A01 (C) Copyright IBM Corp. 1991, 2002
  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 - */
/* CFD List: */
/* */
/* $L1=D45MDEYE HTCP320 960205 KAA: RAS DCR - Module Eyecatchers */
/* $L2=D109 HTCP340 971024 KDJ: OS/390 Copyright */
/* $A1=PQ11420 HTCP340 971210 SLHUANG: Remove FRECEIVEerror and */
/* replace SendTurnCode */
/* $N1=PMV24171 CSV1R4 011128 SLHUANG: Ignore Bufferspaceavailable */
/* notification */
/* */
/* End CFD List: */
}

*****
/* Memory-to-memory Data Transfer Rate Measurement */
/* */
/* Pseudocode: Establish access to TCP/IP Services */
/* Prompt user for operation parameters */
/* Open a connection (Sender:active, Receiver:passive) */
/* 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 = 999; { constant on both sides }
CLOCKunitsPERthousandth = '3E8000'x;

static
Buffer      : packed array (.1..BUFFERlength.) of char;
BufferAddress : Address31Type;
ConnectionInfo : StatusInfoType;
Count       : integer;
DataRate    : real;
Difference   : TimeStampType;
HostAddress  : InternetAddressType;
IbmSeconds   : integer;
Ignored      : integer;
Line        : string(80);
Note        : NotificationInfoType;
PushFlag     : boolean; { for TcpFSend }
RealRate     : real;
ReturnCode   : integer;
SendFlag     : boolean; { are we sending or receiving }
StartingTime : TimeStampType;
Thousandths  : integer;
TotalBytes   : integer;
UrgentFlag   : boolean; { for TcpFSend }

var RoundRealRate : integer;

{*****}
{* Print message, release resources and reset environment *}
{*****}
procedure Restore ( const Message: string;
                   const ReturnCode: integer );
%UHEADER
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,
            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);
        GetHostResol (Trim(Ltrim(Line)), HostAddress);
        if HostAddress = NOhost then begin

```

```

    Restore ('GetHostResol failed. ', OK);
    return;
end;

{ Open a TCP connection: active for Send and passive for Receive }
{ - Connection value will be returned by TcpIp }
{ - initialize IBM reserved fields: Security, Compartment }
{ and Precedence }
{ for Active open - set Connection State to TRYINGtoOPEN }
{ - must initialize foreign socket }
{ for Passive open - set ConnectionState to LISTENING }
{ - may leave foreign socket uninitialized to }
{ accept any open attempt }
with ConnectionInfo do begin
    Connection      := UNSPECIFIEDconnection;
    OpenAttemptTimeout := WAITforever;
    Security         := DEFAULTsecurity;
    Compartment      := DEFAULTcompartment;
    Precedence       := DEFAULTprecedence;
    if SendFlag then begin
        ConnectionState := TRYINGtoOPEN;
        LocalSocket.Address := UNSPECIFIEDaddress;
        LocalSocket.Port := UNSPECIFIEDport;
        ForeignSocket.Address := HostAddress;
        ForeignSocket.Port := PORTnumber;
    end
    else begin
        ConnectionState := LISTENING;
        LocalSocket.Address := HostAddress;
        LocalSocket.Port := PORTnumber;
        ForeignSocket.Address := UNSPECIFIEDaddress;
        ForeignSocket.Port := UNSPECIFIEDport;
    end;
end;
TcpWaitOpen (ConnectionInfo, ReturnCode);
if ReturnCode <> OK then begin
    Restore ('TcpWaitOpen: ', ReturnCode);
    return;
end;

{ Initialization }
BufferAddress := Addr(Buffer(.1.));
StartingTime := ClockTime;
TotalBytes := 0;
Count := 0;
PushFlag := false; { let TcpIp buffer data for efficiency }
UrgentFlag := false; { none of out data is Urgent }

{ 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 }
    case Note.NotificationTag of
        DATAdelivered:
            begin
                TotalBytes := TotalBytes + Note.BytesDelivered;
            end;
    end;
end;

```

```

        {issue next TcpFReceive
        TcpFReceive (ConnectionInfo.Connection, BufferAddress,
            BUFFERlength, ReturnCode);
        if ReturnCode <> OK then begin
            Restore('TcpFReceive: ', ReturnCode);
            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: ', ReturnCode);
            return;
        end;
    end;
end;
BUFFERSpaceAVAILABLE:
{ do nothing };
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. '); *}
RoundRealRate := Round(RealRate);
Msg1(Output, 8, addr(RoundRealRate) );

StartingTime := ClockTime;
TotalBytes := 0;
Count := Count + 1;
end; {Loop while Count < 5 }

{ Close TCP connection and wait till partner also drops connection }
TcpClose (ConnectionInfo.Connection, ReturnCode);
if ReturnCode <> OK then begin
    Restore ('TcpClose: ', ReturnCode);
    return;
end;

{ when partner also drops connection, program will receive
{ CONNECTIONstateCHANGED notification with NewState = NONEXISTENT }
repeat
    GetNextNote (Note, True, ReturnCode);
    if ReturnCode <> OK then begin
        Restore ('GetNextNote: ', ReturnCode);
        return;
    end;
until (Note.NotificationTag = CONNECTIONstateCHANGED) &
    ((Note.NewState = NONEXISTENT) |
    (Note.NewState = CONNECTIONclosing));

Restore ('Program terminated successfully. ', OK);
end.

```

Figure 228. Sample Pascal application program

Appendix A. Multitasking C socket sample program

The first sample program is the server in the C language. It allocates a socket, binds to a port, calls `listen()` to perform a passive open, and uses `select()` to block until a client request arrives. When a client requests a connection, `select()` returns and `accept()` is called to establish the connection.

Note: Some hosts have more than one network address. By specifying a particular network address for the `bind()` call, a server specifies that it wants to honor connections from one particular network address only. If the server specifies the constant `INADDR_ANY` for this address, it accepts connections from any of the machine's network addresses.

This program uses the Multitasking Facility (MTF). The server has started a number of subtasks with the MTF task initialization service `tinit()`. When the server has accepted a connection, it calls `tsched()` to start the subtask that will handle the client. The server then uses `givesocket()` and `takesocket()` to pass the connection to the subtask. When the connection has been passed to the subtask, the main loop blocks in `select()` waiting for another client.

The second program is the subtask in C. When it begins, it does a `takesocket()`. It was passed two 8-byte names that define the parent task from which it will obtain the socket. After it gets the socket, it sends a message to this new client and then waits for the client to send a message back.

The third program is the client in C. It allocates a socket, binds to a port, and connects to a server port that is passed as the second parameter port number 691. Then it has a conversation with the server (actually the server's subtask) sending and receiving messages alternatively.

Notes:

1. When you compile the C sample programs, use `DEF(MVS)` in the CPARM list.
2. When you run the server program, specify `PARM='9999'` to use port 9999.
3. When you run the client program, specify `PARM='MVSF 9999'` to use port 9999. Replace `MVSF` with the host name of your MVS system.

Server sample program in C

The following C socket server program is the `MTCSRVR` member in the `SEZAINST` data set.

MTCSRVR C socket server program sample

```
/* IBM COPYR ***** */
/*
/* Component Name: MTCSRVR (alias EZAEC047)
/*
/*
/* Copyright:      Licensed Materials - Property of IBM
/*
/*                  "Restricted Materials of IBM"
/*
/*                  5650-ZOS
/*
/*                  Copyright IBM Corp. 1977, 2023
/*
/*                  US Government Users Restricted Rights -
/*                  Use, duplication or disclosure restricted by
/*                  GSA ADP Schedule Contract with IBM Corp.
/*
/* Status:         ZCSV3R1
/*
/* Flag Reason    Release Date   Origin   Description
/* -----
/* $71= RJWORDSM ZCSV3R1  221201 bkelsey: Words Matter      JP33
/*
/* SMP/E Distribution Name: EZAEC049
/*
```

```

/*                                                                    */
/*                                                                    */
/**** IBMCPYR *****/

/*****/
/* C socket Server Program                                           */
/*                                                                    */
/* This code performs the server functions for multitasking, which  */
/* include                                                            */
/*     . creating subtasks                                           */
/*     . socket(), bind(), listen(), accept()                        */
/*     . getclientid                                                 */
/*     . givesocket() to TCP/IP in preparation for the subtask       */
/*         to do a takesocket()                                       */
/*     . select()                                                    */
/*                                                                    */
/* There are three test tasks running:                                */
/*     . listener                                                     @71C*/
/*     . listener subtask - subtask in listener address space       @71C*/
/*     . client                                                       */
/*                                                                    */
/*****/

static char ibmcopyr[] =
    "MTC SRVR - Licensed Materials - Property of IBM. "
    "This module is \"Restricted Materials of IBM\" "
    "5647-A01 (C) Copyright IBM Corp. 1994, 1996. "
    "See IBM Copyright Instructions.";

#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>
#include <netdb.h>
#include <socket.h>
#include <inet.h>
#include <fcntl.h>
#include <errno.h>
#include <tcperrno.h>
#include <bsdtime.h>
#include <mtf.h>
#include <stdio.h>

int dotinit(void);
void getsock(int *s);
int dobind(int *s, unsigned short port);
int dolisten(int *s);
int getname(char *myname, char *mysname);
int doaccept(int *s);
int testgive(int *s);
int dogive(int *clsocket, char *myname);

/*
 * Server Main.
 */
main(argc, argv)
int argc;
char **argv;
{
    unsigned short port;      /* port server for bind          */
    int s;                   /* socket for accepting connections */
    int rc;                  /* return code                    */
    int count;               /* counter for number of sockets  */
    int clsocket;            /* client socket                  */
    char myname[8];          /* 8 char name of this address space */
    char mysname[8];         /* my subtask 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]);
    fprintf(stdout, "Server: port = %d \n", port);

    /*
     * Create subtasks

```

```

*/
rc = dotinit();
if (rc < 0)
    perror("Srvr: error for tinit");
printf("rc from tinit is %d\n", rc);

getsock(&s);
printf("Srvr: socket = %d\n", s);

rc = dobind(&s, port);
if (rc < 0)
    tcperror("Srvr: error for bind");
printf("Srvr: rc from bind is %d\n", rc);

rc = dolisten(&s);
if (rc < 0)
    tcperror("Srvr: error for listen");
printf("Srvr: rc from listen is %d\n", rc);

/*****
 * To do nonblocking mode,
 * uncomment out this code.
 */
rc = fcntl(s, F_SETFL, FNDELAY);
if (rc != 0)
    tcperror("Error for fcntl");
printf("rc from fcntl is %d\n", rc);

*****/

rc = getname(myname, myname);
if (rc < 0)
    tcperror("Srvr: error for getclientid");
printf("Srvr: rc from getclientid is %d\n", rc);

/*-----*/
/* . issue accept(), waiting for client connection */
/* . issue givesocket() to pass client's socket to TCP/IP */
/* . issue select(), waiting for subtask to complete takesocket() */
/* . close our local socket associated with client's socket */
/* . loop on accept(), waiting for another client connection */
/*-----*/
rc = 0;
count = 0; /* number of sockets */
while (rc == 0) {
    clsocket = doaccept(&s);
    printf("Srvr: clsocket from accept is %d\n", clsocket);
    count = count + 1;
    printf("Srvr: ###number of sockets is %d\n", count);
    if (clsocket != 0) {
        rc = dogive(&clsocket, myname);
        if (rc < 0)
            tcperror("Srvr: error for dogive");
        printf("Srvr: rc from dogive is %d\n", rc);
        if (rc == 0) {
            rc = tsched(MTF_ANY, "csub", &clsocket,
                        myname, myname);
            if (rc < 0)
                perror("error for tsched");
            printf("Srvr: rc from tsched is %d\n", rc);

            rc = testgive(&clsocket);
            printf("Srvr: rc from testgive is %d\n", rc);

            sleep(60); /* do simplified situation first */

            printf("Srvr: closing client socket %d\n", clsocket);
            rc = close(clsocket); /* give back this socket */
            if (rc < 0)
                tcperror("error for close of clsocket");
            printf("Srvr: rc from close of clsocket is %d\n", rc);
            /*****/
            exit(0); /* do this simplified situation first */
            /*****/
        } /* end of if (rc == 0) */
    } /* end of if (clsocket != 0) */
} /* end of while (rc == 0) */
} /* end of main */

/*-----*/
/* dotinit() */
/* Call tinit() to ATTACH subtask and fetch() subtask load module */

```

```

/*-----*/
int dotinit(void)
{
    int rc;
    int numsubs = 1;
    printf("Srvr: calling __tinit\n");
    rc = __tinit("mtccsub", numsubs);
    return rc;
}

/*-----*/
/*    getsock()                                */
/*    Get a socket                             */
/*-----*/
void getsock(int *s)
{
    int temp;
    temp = socket(AF_INET, SOCK_STREAM, 0);
    *s = temp;
    return;
}

/*-----*/
/*    dobind()                                */
/*    Bind to all interfaces                   */
/*-----*/
int dobind(int *s, unsigned short port)
{
    int rc;
    int temps;
    struct sockaddr_in tsock;
    memset(&tsock, 0, sizeof(tsock)); /* clear tsock to 0's */
    tsock.sin_family = AF_INET;
    tsock.sin_addr.s_addr = INADDR_ANY; /* bind to all interfaces */
    tsock.sin_port = htons(port);

    temps = *s;
    rc = bind(temps, (struct sockaddr *)&tsock, sizeof(tsock));
    return rc;
}

/*-----*/
/*    dolisten()                              */
/*    Listen to prepare for client connections. */
/*-----*/
int dolisten(int *s)
{
    int rc;
    int temps;
    temps = *s;
    rc = listen(temps, 10); /* backlog of 10 */
    return rc;
}

/*-----*/
/*    getname()                              */
/*    Get the identifiers by which TCP/IP knows this server. */
/*-----*/
int getname(char *myname, char *mysname)
{
    int rc;
    struct clientid cid;
    memset(&cid, 0, sizeof(cid));
    rc = getclientid(AF_INET, &cid);
    memcpy(myname, cid.name, 8);
    memcpy(mysname, cid.subtaskname, 8);
    return rc;
}

/*-----*/
/*    doaccept()                              */
/*    Select() on this socket, waiting for another client connection. */
/*    If connection is pending, issue accept() to get client's socket */
/*-----*/
int doaccept(int *s)
{
    int temps;
    int clsocket;
    struct sockaddr clientaddress;
    int addrlen;
    int maxfdpl;
    struct fd_set readmask;

```

```

struct fd_set writemask;
struct fd_set excpmask;
int rc;
struct timeval time;

temps = *s;
time.tv_sec = 1000;
time.tv_usec = 0;
maxfdpl = temps + 1;

FD_ZERO(&readmask);
FD_ZERO(&writemask);
FD_ZERO(&excpmask);

FD_SET(temps, &readmask);

rc = select(maxfdpl, &readmask, &writemask, &excpmask, &time);
printf("Srvr: rc from select is %d\n", rc);
if (rc < 0) {
    tcperror("error from select");
    return rc;
}
else if (rc == 0) { /* time limit expired */
    return rc;
}
else {
    /* this socket is ready */
    addrlen = sizeof(clientaddress);
    clsocket = accept(temps, &clientaddress, &addrlen);
    return clsocket;
}
}

/*-----*/
/*  testgive() */
/*  Issue select(), checking for an exception condition, which */
/*  indicates that takesocket() by the subtask was successful. */
/*-----*/
int testgive(int *s)
{
    int temps;
    struct sockaddr clientaddress;
    int addrlen;
    int maxfdpl;
    struct fd_set readmask;
    struct fd_set writemask;
    struct fd_set excpmask;
    int rc;
    struct timeval time;

    temps = *s;
    time.tv_sec = 1000;
    time.tv_usec = 0;
    maxfdpl = temps + 1;

    FD_ZERO(&readmask);
    FD_ZERO(&writemask);
    FD_ZERO(&excpmask);

    /* FD_SET(temps, &readmask); */
    /* FD_SET(temps, &writemask); */
    /* FD_SET(temps, &excpmask); */

    rc = select(maxfdpl, &readmask, &writemask, &excpmask, &time);
    printf("Srvr: rc from select for testgive is %d\n", rc);
    if (rc < 0) {
        tcperror("Srvr: error from testgive");
    }
    else
        rc = 0;

    return rc;
}

/*-----*/
/*  dogive() */
/*  Issue givesocket() for giving client's socket to subtask. */
/*-----*/
int dogive(int *clsocket, char *myname)
{
    int rc;
    struct clientid cid;
    int temps;

```

```

    temps = *clsocket;
    memset(&cid, 0, sizeof(cid));
    cid.domain = AF_INET;

    memcpy(cid.name,          myname,      8);
    memcpy(cid.subtaskname, "", 8);
    printf("Srvr: givesocket socket is %d\n", temps);
    printf("Srvr: givesocket name is %s\n", cid.name);

    rc = givesocket(temps, &cid);
    return rc;
}

```

The subtask sample program in C

The following C socket server program is the MTCCSUB member in the SEZAINST data set.

```

/**** IBMCOPYR *****/
/*
/* Component Name: MTCCSUB
/*
/* Copyright:      Licensed Materials - Property of IBM
/*
/*                  "Restricted Materials of IBM"
/*
/*                  5647-A01
/*
/*                  (C) Copyright IBM Corp. 1977, 1998
/*
/*                  US Government Users Restricted Rights -
/*                  Use, duplication or disclosure restricted by
/*                  GSA ADP Schedule Contract with IBM Corp.
/*
/* Status:         CSV2R6
/*
/* SMP/E Distribution Name: EZAEC048
/*
/**** IBMCOPYR *****/

/*****
/* C Socket Server Subtask Program
/*
/* This code is started by the tsched() routine of C/370 MTF.
/* Its purpose is to do a takesocket() and then send/rcv with the
/* client process.
*****/
#pragma runopts(noargparse,plist(mvs),noexecops)

static char ibmcopyr[] =
    "MTCCSUB - Licensed Materials - Property of IBM. "
    "This module is \"Restricted Materials of IBM\" "
    "5647-A01 (C) Copyright IBM Corp. 1994, 1996. "
    "See IBM Copyright Instructions.";

#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>
#include <netdb.h>
#include <socket.h>
#include <inet.h>
#include <fcntl.h>
#include <errno.h>
#include <tcperrno.h>
#include <bsdtime.h>
#include <stdio.h>

/*
 * Server subtask
 */
csub(int *clsock,          /* address of socket passed */
     char *tskname,       /* address of caller's name */
     char *tsksname)      /* address of caller's sname */

```

```

{
    int temps;                /* */
    int sendbytes;           /* # bytes sent */
    int recvbytes;           /* # bytes received */
    int clsocket;            /* client socket */
    int rc;                  /* */
    char xtskname[8];         /* caller's name */
    char xtksname[8];         /* caller's subtask name */

    clsocket = *clsock;
    memcpy(&xtskname, tskname, 8); /* local copy */
    memcpy(&xtksname, tskname, 8); /* local copy */
    rc = doget(&clsocket, xtskname, xtksname);
    printf("Csub: returned from doget()\n");
    if (rc < 0)
        tcperror("Csub: Error from doget");
    printf("Csub: rc from doget is %d\n", rc);

    temps = rc; /* new socket number */
    if (temps > -1) do {
        sendbytes = dosend(&temps);
        recvbytes = dorecv(&temps);
    } while (0);
    /* } while (recvbytes > 0); do simplified situation first */
    fflush(stdout);
    sleep(30);
}

/*-----*/
/* doget() */
/*-----*/
int doget(int *clsocket, char *xtskname, char *xtksname)
{
    int rc;
    int temps;
    struct clientid cid;

    memset(&cid, 0, sizeof(cid));
    temps = *clsocket;
    memcpy(cid.name, xtskname, 8);
    memcpy(cid.subtaskname, xtksname, 8);
    cid.domain = AF_INET;
    rc = takesocket(&cid, temps);
    *clsocket = temps;
    return rc;
}

/*-----*/
/* dosend() */
/*-----*/
int dosend(int *clsocket)
{
    int temps;
    int sendbytes;
    char data[80] = "Message from subtask: I sent this data";

    /******
       note: stream mode means that data is not sent
       as a record and can therefore flow in
       variable sized chunks across the network.
       This example is a simplified situation.
    *****/
    temps = *clsocket;
    sendbytes = send(temps, data, sizeof(data), 0);
    printf("Csub: sendbytes = %d\n", sendbytes);
    return sendbytes;
}

/*-----*/
/* dorecv() */
/*-----*/
int dorecv(int *clsocket)
{
    int temps;
    int recvbytes;
    char data[80];
    char *datap;

    /******

```

```

        note: stream mode means that data is not sent
              as a record and can therefore flow in
              variable sized chunks across the network.
              This example is a simplified situation.
    *****/
    temps = *clsocket;
    recvbytes = recv(temps, data, sizeof(data), 0);
    if (recvbytes > 0)
        printf("Csub: data recv: %s\n", data);
    else
        printf("Csub: client stopped sending data\n");
    printf("Csub: recvbytes = %d\n", recvbytes);
    return recvbytes;
}

```

Figure 229. MTCCSUB C socket server program sample

The client sample program in C

The following C socket server program is the MTCCCLNT member in the SEZAINST data set.

```

/**** IBMCOPYR *****/
/*
/* Component Name: MTCCCLNT
/*
/*
/* Copyright:    Licensed Materials - Property of IBM
/*
/*              "Restricted Materials of IBM"
/*
/*              5647-A01
/*
/*              (C) Copyright IBM Corp. 1977, 1998
/*
/*              US Government Users Restricted Rights -
/*              Use, duplication or disclosure restricted by
/*              GSA ADP Schedule Contract with IBM Corp.
/*
/* Status:      CSV2R6
/*
/* SMP/E Distribution Name: EZAEC047
/*
/*
/**** IBMCOPYR *****/

/*****
/* C Socket Client Program
/*
/* This code sends and receives mgs with the server subtask.
*****/

static char ibmcopyr[] =
    "MTCCCLNT - Licensed Materials - Property of IBM. "
    "This module is \"Restricted Materials of IBM\" "
    "5647-A01 (C) Copyright IBM Corp. 1994, 1996. "
    "See IBM Copyright Instructions.";

#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>
#include <netdb.h>
#include <socket.h>
#include <inet.h>
#include <errno.h>
#include <tcperrno.h>
#include <bsdtime.h>
#include <stdio.h>

int dosend(int *s);
int dorecv(int *s);
int doconn(int *s, unsigned long *octaddrp, unsigned short port);
void getsock(int *s);

/*
* Client
*/

```



```

main(int argc, char **argv)
{
    int gotbytes;           /* number of bytes received */
    int sndbytes;           /* number of bytes sent */
    int s;                  /* socket descriptor */
    int rc;                 /* return code */
    struct in_addr octaddr;  /* host internet address (binary) */
    unsigned short port;    /* port number sent as parameter */
    char * charaddr;        /* host internet address (dotted dec) */
    struct hostent *hostnm;  /* server host name information */

    /*
     * 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(2);
    }
    octaddr.s_addr = *((unsigned long *)hostnm->h_addr);

    /*
     * The port is the second argument.
     */
    port = (unsigned short) atoi(argv[2]);
    fprintf(stdout, "Clnt: port = %d\n", port);

    getsock(&s);
    printf("Clnt: our socket is %d\n", s);

    charaddr = inet_ntoa(octaddr);
    printf("Clnt: address of host is %8s\n", charaddr);

    rc = doconn(&s, &octaddr.s_addr, port);
    if (rc < 0)
        tcperror("Clnt: error for connect");
    else {
        printf("Clnt: rc from connect is %d\n", rc);
        do {
            gotbytes = dorecv(&s);
            sndbytes = dosend(&s);
        } while (0);
        /* } while (sndbytes > 0); do simplified situation first */
        sleep(15);
    }
}

/*-----*/
/*  getsock() */
/*-----*/
void getsock(int *s)
{
    int temp;
    temp = socket(AF_INET, SOCK_STREAM, 0);
    *s = temp;
    return;
}

/*-----*/
/*  doconn() */
/*-----*/
int doconn(int *s, unsigned long *octaddrp, unsigned short port)
{
    int rc;
    int temps;
    struct sockaddr_in tsock;

    memset(&tsock, 0, sizeof(tsock));
    tsock.sin_family = AF_INET;
    tsock.sin_port = htons(port);
    tsock.sin_addr.s_addr = *octaddrp;

    temps = *s;
    rc = connect(temps, (struct sockaddr *)&tsock, sizeof(tsock));
    return rc;
}

```

```

}

/*-----*/
/*    dorecv()    */
/*-----*/
int dorecv(int *s)
{
    int temps;
    int gotbytes;
    char data[100];

    temps = *s;

    gotbytes = recv(temps, data, sizeof(data), 0);
    if (gotbytes < 0) {
        tcperror("Clnt: error for recv");
    }
    else
        printf("Clnt: data recv: %s\n", data);
    return gotbytes;
}

/*-----*/
/*    dosend()    */
/*-----*/
int dosend(int *s)
{
    int temps;
    int sndbytes;
    char data[50];

    temps = *s;
    gets(data);
    printf("clnt: data to send: %s\n", data);
    sndbytes = send(temps, data, sizeof(data), 0);
    if (sndbytes < 0) {
        tcperror("Clnt: error for send");
    }
    else
        printf("Clnt: sent %d bytes to server subtask\n", sndbytes);
    return sndbytes;
}

```

Figure 230. MTCCLNT C socket server program sample

Appendix B. Socket call error return codes

This information contains error return codes for socket calls. The error codes apply to all of the following socket APIs:

- TCP/IP C socket API
- X/Open Transport Interface
- Macro API for IPv4 or IPv6 that is written in z/OS assembler language
- Call instruction API for IPv4 or IPv6 socket applications
- z/OS Communications Server socket API for REXX
- Pascal language for IPv4 socket API

It also contains sockets extended return codes that apply only to the macro, call instruction, and REXX socket APIs.

If the return code is not listed in this information, it is a return code that is received from z/OS UNIX. See [z/OS UNIX System Services Messages and Codes](#) for the z/OS UNIX ERRNOs.

See [“User abend U4093” on page 759](#) for a description of user abend U4093.

System error codes for socket calls

This information contains the error codes and the message names that refer to the following APIs:

- C sockets
- Macro
- Call instruction
- REXX sockets

The names in the Socket Type column are identifiers that apply to all of the above APIs and do not follow the naming convention for any specific API. These message numbers and codes are in the TCPERRNO.H include file.

When a socket call is processed, both a return code and an error number are returned to your program. If the return code is 0 or a positive number, the call completed normally. If the return code is a negative number, the call did not complete normally and an error number is returned. See the following table for the meaning of the error number that is returned.

For the following error conditions, a name is returned by C socket calls and a number is returned by the sockets extended interface calls. The error condition return codes can originate from the socket application programming interface or from a peer server program.

Sockets return codes (ERRNOs)

This section provides the system-wide message numbers and codes set by the system calls. These message numbers and codes are in the TCPERRNO.H include file supplied with TCP/IP Services.

Table 24. Sockets ERRNOs

Error number	Message name	Socket API type	Error description	Programmer's response
1	EAI_NONAME	GETADDRINFO GETNAMEINFO	NODE or HOST cannot be found.	Ensure the NODE or HOST name can be resolved.
1	EDOM	All	Argument too large.	Check parameter values of the function call.
1	EPERM	All	Permission is denied. No owner exists.	Check that TCP/IP is still active; check protocol value of socket () call.

Table 24. Sockets ERRNOs (continued)

Error number	Message name	Socket API type	Error description	Programmer's response
1	EPERM	IOCTL (SIOCGPARTNERINFO)	Both endpoints do not reside in the same security domain.	Check and modify the security domain name for the endpoints. After you correct the security domain name, the application might need to close the connection if the IOCTL is needed.
1	EPERM	IOCTL (SIOCGPARTNERINFO, SIOCSPARTNERINFO)	The security domain name is not defined.	Define the security domain name on both endpoints. After you define the security domain name, the application might need to close the connection if the IOCTL is needed.
1	EPERM	IOCTL (SIOCTLTSCTL)	<p>Denotes one of the following error conditions:</p> <ul style="list-style-type: none"> The TTLS_INIT_CONNECTION option was requested with either TTLS_RESET_SESSION, TTLS_RESET_CIPHER, TTLS_RESET_WRITE_CIPHER, TTLS_SEND_SESSION_TICKET, or TTLS_STOP_CONNECTION The TTLS_STOP_CONNECTION option was requested along with TTLS_RESET_SESSION, TTLS_RESET_CIPHER, TTLS_RESET_WRITE_CIPHER, or TTLS_SEND_SESSION_TICKET The TTLS_ALLOW_HSTIMEOUT option was requested without TTLS_INIT_CONNECTION The TTLS_RESET_WRITE_CIPHER option was requested along with TTLS_RESET_SESSION or TTLS_RESET_CIPHER The TTLS_SEND_SESSION_TICKET option was requested along with TTLS_RESET_SESSION, TTLS_RESET_CIPHER, or TTLS_RESET_WRITE_CIPHER 	<p>Request TTLS_RESET_SESSION, TTLS_RESET_CIPHER, TTLS_RESET_WRITE_CIPHER, or TTLS_SEND_SESSION_TICKET only when TTLS_INIT_CONNECTION and TTLS_STOP_CONNECTION are not requested. Always request TTLS_INIT_CONNECTION when TTLS_ALLOW_HSTIMEOUT is requested. Use separate SIOCTLTSCTL ioctls to request TTLS_INIT_CONNECTION and TTLS_STOP_CONNECTION.</p> <p>Use separate SIOCTLTSCTL ioctls to request TTLS_RESET_WRITE_CIPHER or TTLS_SEND_SESSION_TICKET.</p>
2	EAI_AGAIN	FREEADDRINFO GETADDRINFO GETNAMEINFO	For GETADDRINFO, NODE could not be resolved within the configured time interval. For GETNAMEINFO, HOST could not be resolved within the configured time interval. The Resolver address space has not been started. The request can be retried later.	Ensure the Resolver is active, then retry the request.
2	ENOENT	All	The data set or directory was not found.	Check files used by the function call.
2	ERANGE	All	The result is too large.	Check parameter values of the function call.
3	EAI_FAIL	FREEADDRINFO GETADDRINFO GETNAMEINFO	This is an unrecoverable error. NODELEN, HOSTLEN, or SERVLN is incorrect. For FREEADDRINFO, the resolver storage does not exist.	Correct the NODELEN, HOSTLEN, or SERVLN. Otherwise, call your system administrator.
3	ESRCH	All	The process was not found. A table entry was not located.	Check parameter values and structures pointed to by the function parameters.
4	EAI_OVERFLOW	GETNAMEINFO	The output buffer for the host name or service name was too small.	Increase the size of the buffer to 255 characters, which is the maximum size permitted.
4	EINTR	All	A system call was interrupted.	Check that the socket connection and TCP/IP are still active.
5	EAI_FAMILY	GETADDRINFO GETNAMEINFO	The AF or the FAMILY is incorrect.	Correct the AF or the FAMILY.
5	EIO	All	An I/O error occurred.	Check status and contents of source database if this occurred during a file access.
6	EAI_MEMORY	GETADDRINFO GETNAMEINFO	The resolver cannot obtain storage to process the host name.	Contact your system administrator.
6	ENXIO	All	The device or driver was not found.	Check status of the device attempting to access.
7	E2BIG	All	The argument list is too long.	Check the number of function parameters.
7	EAI_BADFLAGS	GETADDRINFO GETNAMEINFO	FLAGS has an incorrect value.	Correct the FLAGS.
8	EAI_SERVICE	GETADDRINFO	The SERVICE was not recognized for the specified socket type.	Correct the SERVICE.

Table 24. Sockets ERRNOs (continued)

Error number	Message name	Socket API type	Error description	Programmer's response
8	ENOEXEC	All	An EXEC format error occurred.	Check that the target module on an exec call is a valid executable module.
9	EAI_SOCKTYPE	GETADDRINFO	The SOCKTYPE was not recognized.	Correct the SOCKTYPE.
9	EBADF	All	An incorrect socket descriptor was specified.	Check socket descriptor value. It might be currently not in use or incorrect.
9	EBADF	Givesocket	The socket has already been given. The socket domain is not AF_INET or AF_INET6.	Check the validity of function parameters.
9	EBADF	Select	One of the specified descriptor sets is an incorrect socket descriptor.	Check the validity of function parameters.
9	EBADF	Takesocket	The socket has already been taken.	Check the validity of function parameters.
9	EAI_SOCKTYPE	GETADDRINFO	The SOCKTYPE was not recognized.	Correct the SOCKTYPE.
10	ECHILD	All	There are no children.	Check if created subtasks still exist.
11	EAGAIN	All	There are no more processes.	Retry the operation. Data or condition might not be available at this time.
11	EAGAIN	All	TCP/IP is not active at the time of the request.	Start TCP/IP, and retry the request.
11	EAGAIN	IOCTL (SIOCGPARTNERINFO)	The IOCTL was issued in no-suspend mode and the SIOCGPARTNERINFO IOCTL has not been issued.	Reissue the IOCTL with a timeout value to set the amount of time to wait while the partner security credentials are being retrieved. Restriction: You cannot use a select mask to determine when an IOCTL is complete, because an IOCTL is not affected by whether the socket is running in blocking or nonblocking mode. If the IOCTL times out, reissue the IOCTL to retrieve the partner security credentials.
12	ENOMEM	All	There is not enough storage.	Check the validity of function parameters.
13	EACCES	All	Permission denied, caller not authorized.	Check access authority of file.
13	EACCES	IOCTL (SIOCGPARTNERINFO)	The application is not running in supervisor state, is not APF authorized, or is not permitted to the appropriate SERVAUTH profile.	Allow the application to issue this IOCTL, or provide the user ID with the proper SERVAUTH permission.
13	EACCES	IOCTL (SIOCTLISCTL)	The IOCTL is requesting a function that requires that the socket be mapped to policy that specifies ApplicationControlled On.	Check policy and add ApplicationControlled On if the application should be permitted to issue the controlled SIOCTLISCTL functions.
13	EACCES	Takesocket	The other application (listener) did not give the socket to your application. Permission denied, caller not authorized.	Check access authority of file.
14	EFAULT	All	An incorrect storage address or length was specified.	Check the validity of function parameters.
14	EFAULT	All EZASMI macros when using an asynchronous exit routine.	The exit routine has abnormally ended (ABEND condition).	Correct the error in the routine's code. Add an ESTAE routine to the exit.
14	EFAULT	IOCTL (SIOCSAPPLDATA)	An abend occurred while attempting to copy the SetADcontainer structure from the address provided in the SetAD_ptr field.	Check the validity of function parameters.
15	ENOTBLK	All	A block device is required.	Check device status and characteristics.
16	EBUSY	All	Listen has already been called for this socket. Device or file to be accessed is busy.	Check if the device or file is in use.
17	EEXIST	All	The data set exists.	Remove or rename existing file.
18	EXDEV	All	This is a cross-device link. A link to a file on another file system was attempted.	Check file permissions.
19	ENODEV	All	The specified device does not exist.	Check file name and if it exists.

Table 24. Sockets ERRNOs (continued)

Error number	Message name	Socket API type	Error description	Programmer's response
20	ENOTDIR	All	The specified directory is not a directory.	Use a valid file that is a directory.
21	EISDIR	All	The specified directory is a directory.	Use a valid file that is not a directory.
22	EINVAL	All types	An incorrect argument was specified.	Check the validity of function parameters.
22	EINVAL	Multicast Source filter APIs	Mix of any-source, source-specific or full-state APIs	Specify the correct type of APIs.
22	EINVAL	MCAST_JOIN_GROUP, MCAST_JOIN_SOURCE_GROUP, MCAST_BLOCK_SOURCE, MCAST_LEAVE_GROUP, MCAST_LEAVE_SOURCE_GROUP, MCAST_UNBLOCK_SOURCE, SIOCGMSFILTER, SIOCSMSFILTER	The socket address family or the socket length of the input multicast group or the source IP address is not correct.	Specify the correct value.
22	EINVAL	SIOCSMSFILTER, SIOCSIPMSFILTER	The specified filter mode is not correct.	Specify the correct value.
23	ENFILE	All	Data set table overflow occurred.	Reduce the number of open files.
24	EMFILE	All	The socket descriptor table is full.	Check the maximum sockets specified in MAXDESC().
25	ENOTTY	All	An incorrect device call was specified.	Check specified IOCTL() values.
26	ETXTBSY	All	A text data set is busy.	Check the current use of the file.
27	EFBIG	All	The specified data set is too large.	Check size of accessed dataset.
28	ENOSPC	All	There is no space left on the device.	Increase the size of accessed file.
29	ESPIPE	All	An incorrect seek was attempted.	Check the offset parameter for seek operation.
30	EROFS	All	The data set system is Read only.	Access data set for read only operation.
31	EMLINK	All	There are too many links.	Reduce the number of links to the accessed file.
32	EPIPE	All	The connection is broken. For socket write/send, peer has shut down one or both directions.	Reconnect with the peer.
32	EPIPE	IOCTL (SIOCTLSCTL requesting TTLS_INIT_CONNECTION, TTLS_RESET_CIPHER, TTLS_RESET_WRITE_CIPHER, TTLS_SEND_SESSION_TICKET or TTLS_STOP_CONNECTION)	The TCP connection is not in the established state.	Issue the SIOCTLSCTL IOCTL when the socket is connected.
33	EDOM	All	The specified argument is too large.	Check and correct function parameters.
34	ERANGE	All	The result is too large.	Check function parameter values.
35	EWOULDBLOCK	Accept	The socket is in nonblocking mode and connections are not queued. This is not an error condition.	Reissue Accept().
35	EWOULDBLOCK	IOCTL (SIOCTLSCTL)	The handshake is in progress and the socket is a nonblocking socket.	For a nonblocking socket, you can wait for the handshake to complete by issuing Select or Poll for Socket Writable.
35	EWOULDBLOCK	Read Recvfrom	The socket is in nonblocking mode and read data is not available. This is not an error condition.	Issue a select on the socket to determine when data is available to be read or reissue the Read()/Recvfrom().
35	EWOULDBLOCK	All receive calls (RECV, RECVMMSG, RECVFROM, READV, READ), when the socket is set with the SO_RCVTIMEO socket option	The socket is in blocking mode and the receive call has blocked for the time period that was specified in the SO_RCVTIMEO option. No data was received.	The application should reissue the receive call.
35	EWOULDBLOCK	All receive calls (RECV, RECVMMSG, RECVFROM, READV, READ), when AT-TLS is used to provide transparent TLS/SSL support for a TCP socket	The socket is in blocking mode and application data is not available. This can occur when a TLS alert message is at the head of the receive queue. The receive call triggers the processing of the alert message and returns EWOULDBLOCK to indicate that no application data is processed. A select() for read might indicate that the TCP socket is ready due to the alert being present.	The application should reissue the select() call on the socket to determine when data is available to be read or reissue the receive call.
35	EWOULDBLOCK	Send Sendto Write	The socket is in nonblocking mode and buffers are not available.	Issue a select on the socket to determine when data is available to be written or reissue the Send(), Sendto(), or Write().

Table 24. Sockets ERRNOs (continued)

Error number	Message name	Socket API type	Error description	Programmer's response
35	EWOULDBLOCK	All send calls (SEND, SENDMSG, SENDTO, WRITEV, WRITE), when the socket is set with the SO_SNDTIMEO socket option	The socket is in blocking mode and the send call has blocked for the time period that was specified in the SO_SNDTIMEO option. No data was sent.	The application should reissue the send call.
36	EINPROGRESS	Connect	The socket is marked nonblocking and the connection cannot be completed immediately. This is not an error condition.	See the Connect() description for possible responses.
36	EINPROGRESS	IOCTL (SIOCGPARTNERINFO)	The IOCTL was issued in no-suspend mode after the SIOCSPARTNERINFO IOCTL was issued, but the partner security credentials are not currently available.	Retry the IOCTL, or issue the IOCTL with a timeout value to set the amount of time to wait while the partner security credentials are being retrieved. Restriction: You cannot use a select mask to determine when an IOCTL is complete, because an IOCTL is not affected by whether the socket is running in blocking or nonblocking mode. If the IOCTL times out, reissue the IOCTL to retrieve the partner security credentials.
36	EINPROGRESS	IOCTL (SIOCTTLSTCTL requesting TTLS_INIT_CONNECTION or TTLS_STOP_CONNECTION)	The handshake is already in progress and the socket is a nonblocking socket.	For a nonblocking socket, you can wait for the handshake to complete by issuing Select or Poll for Socket Writable.
37	EALREADY	Connect	The socket is marked nonblocking and the previous connection has not been completed.	Reissue Connect().
37	EALREADY	IOCTL (SIOCGPARTNERINFO)	The request is already in progress. Only one IOCTL can be outstanding.	Check and modify the socket descriptor, if specified; otherwise, no action is needed.
37	EALREADY	IOCTL (SIOCTTLSTCTL requesting TTLS_INIT_CONNECTION or TTLS_STOP_CONNECTION)	For TTLS_INIT_CONNECTION, the socket is already secure. For TTLS_STOP_CONNECTION, the socket is not secure.	Modify the application so that it issues the SIOCTTLSTCTL IOCTL that requests TTLS_INIT_CONNECTION only when the socket is not already secure and that requests TTLS_STOP_CONNECTION only when the socket is secure.
37	EALREADY	Maxdesc	A socket has already been created calling Maxdesc() or multiple calls to Maxdesc().	Issue Getablesize() to query it.
37	EALREADY	Setibmopt	A connection already exists to a TCP/IP image. A call to SETIBMOPT (IBMTCP_IMAGE), has already been made.	Call Setibmopt() only once.
38	ENOTSOCK	All	A socket operation was requested on a nonsocket connection. The value for socket descriptor was not valid.	Correct the socket descriptor value and reissue the function call.
39	EDESTADDRREQ	All	A destination address is required.	Fill in the destination field in the correct parameter and reissue the function call.
40	EMSGSIZE	Sendto Sendmsg Send Write for Datagram (UDP) or RAW sockets	The message is too long. It exceeds the IP limit of 64K or the limit set by the setsockopt() call.	Either correct the length parameter, or send the message in smaller pieces.
41	EPROTOTYPE	All	The specified protocol type is incorrect for this socket.	Correct the protocol type parameter.
41	EPROTOTYPE	bind2addrsel	The referenced socket is not a stream (TCP) or datagram (UDP) socket.	Issue bind2addrsel() on TCP or UDP sockets only.
41	EPROTOTYPE	IOCTL (SIOCGPARTNERINFO, SIOCSAPPLDATA, SIOCSPARTNERINFO, SIOCTTLSTCTL)	Socket is not a TCP socket.	Issue the IOCTL on TCP sockets only.
42	ENOPROTOOPT	Getsockopt Setsockopt	The socket option specified is incorrect or the level is not SOL_SOCKET. Either the level or the specified optname is not supported.	Correct the level or optname.
42	ENOPROTOOPT	Getibmsockopt Setibmsockopt	Either the level or the specified optname is not supported.	Correct the level or optname.
43	EPROTONOSUPPORT	Socket	The specified protocol is not supported.	Correct the protocol parameter.
44	ESOCKTNOSUPPORT	All	The specified socket type is not supported.	Correct the socket type parameter.
45	EOPNOTSUPP	Accept Givesocket	The selected socket is not a stream socket.	Use a valid socket.

Table 24. Sockets ERRNOs (continued)

Error number	Message name	Socket API type	Error description	Programmer's response
45	EOPNOTSUPP	bind2addrsel	The referenced socket is not a type that supports the requested function	Use a socket of the correct type.
45	EOPNOTSUPP	Getibmopt Setibmopt	The socket does not support this function call. This command is not supported for this function.	Correct the command parameter. See Getibmopt() for valid commands. Correct by ensuring a Listen() was not issued before the Connect().
45	EOPNOTSUPP	GETSOCKOPT	The specified GETSOCKOPT OPTNAME option is not supported by this socket API.	Correct the GETSOCKOPT OPTNAME option.
45	EOPNOTSUPP	IOCTL	The specified IOCTL command is not supported by this socket API.	Correct the IOCTL COMMAND.
45	EOPNOTSUPP	IOCTL (SIOCSPARTNERINFO)	The request must be issued before the listen call or the connect call.	Check and modify the socket descriptor, or close the connection and reissue the call.
45	EOPNOTSUPP	IOCTL (SIOCTLSCTL requesting TTLS_INIT_CONNECTION, TTLS_RESET_SESSION, TTLS_RESET_CIPHER, TTLS_RESET_WRITE_CIPHER, TTLS_SEND_SESSION_TICKET, or TTLS_STOP_CONNECTION)	Mapped policy indicates that AT-TLS is not enabled for the connection.	Modify the policy to enable AT-TLS for the connection.
45	EOPNOTSUPP	Listen	The socket does not support the Listen call.	Change the type on the Socket() call when the socket was created. Listen() supports only a socket type of SOCK_STREAM.
45	EOPNOTSUPP	RECV, RECVFROM, RECVMMSG, SEND, SENDTO, SENDMSG	The specified flags are not supported on this socket type or protocol.	Correct the FLAG.
46	EPFNOSUPPORT	All	The specified protocol family is not supported or the specified domain for the client identifier is not AF_INET=2.	Correct the protocol family.
47	EAFNOSUPPORT	bind2addrsel inet6_is_srcaddr	You specified an IP address that is not an AF_INET6 IP address	Correct the IP address. If the IP address is an IPv4 address, you must specify it as an IPv4-mapped IPv6 address.
47	EAFNOSUPPORT	bind2addrsel inet6_is_srcaddr	You attempted an IPv6-only API for a stack that does not support the AF_INET6 domain.	Activate the AF_INET6 stack, and retry the request.
47	EAFNOSUPPORT	Bind Connect Socket	The specified address family is not supported by this protocol family.	For Socket(), set the domain parameter to AF_INET. For Bind() and Connect(), set Sin_Family in the socket address structure to AF_INET.
47	EAFNOSUPPORT	Getclient Givesocket	The socket specified by the socket descriptor parameter was not created in the AF_INET domain.	The Socket() call used to create the socket should be changed to use AF_INET for the domain parameter.
47	EAFNOSUPPORT	IOCTL	You attempted to use an IPv4-only ioctl on an AF_INET6 socket.	Use the correct socket type for the ioctl or use an ioctl that supports AF_INET6 sockets.
48	EADDRINUSE	Bind, Connect	The address is in a timed wait because a LINGER delay from a previous close or another process is using the address. This error can also occur if the port specified in the bind call has been configured as RESERVED on a port reservation statement in the TCP/IP profile.	To reuse the same address, use Setsockopt() with SO_REUSEADDR. See the section about Setsockopt() in z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference for more information. Otherwise, use a different address or port in the socket address structure.
48	EADDRINUSE	IP_ADD_MEMBERSHIP, IP_ADD_SOURCE_MEMBERSHIP, IPV6_JOIN_GROUP, MCAST_JOIN_GROUP, MCAST_JOIN_SOURCE_GROUP	The specified multicast address and interface address (or interface index) pair is already in use.	Correct the specified multicast address, interface address, or interface index.
49	EADDRNOTAVAIL	Bind	The specified address is incorrect for this host.	Correct the function address parameter.
49	EADDRNOTAVAIL	Connect	The calling host cannot reach the specified destination.	Correct the function address parameter.

Table 24. Sockets ERRNOs (continued)

Error number	Message name	Socket API type	Error description	Programmer's response
49	EADDRNOTAVAIL	bind2addrsel	For the specified destination address, there is no source address that the application can bind to. Possible reasons can be one of the following situations: <ul style="list-style-type: none"> The socket is a stream socket, but the specified destination address is a multicast address. No ephemeral ports are available to assign to the socket. 	Correct the function address parameter or issue the request when ephemeral ports are available.
49	EADDRNOTAVAIL	inet6_is_srcaddr	The address specified is not correct for one of these reasons: <ul style="list-style-type: none"> The address is not an address on this node. The address was not active at the time of the request. The scope ID specified for a link-local IPV6 address is incorrect. 	Correct or activate the address
49	EADDRNOTAVAIL	IP_BLOCK_SOURCE, IP_ADD_SOURCE_MEMBERSHIP, MCAST_BLOCK_SOURCE, MCAST_JOIN_SOURCE_GROUP	A duplicate source IP address is specified on the multicast group and interface pair.	Correct the specified source IP address.
49	EADDRNOTAVAIL	IP_UNBLOCK_SOURCE, IP_DROP_SOURCE_MEMBERSHIP, MCAST_UNBLOCK_SOURCE, MCAST_LEAVE_SOURCE_GROUP	A previously blocked source multicast group cannot be found.	Correct the specified address.
49	EADDRNOTAVAIL	Multicast APIs	The specified multicast address, interface address, or interface index is not correct.	Correct the specified address.
50	ENETDOWN	All	The network is down.	Retry when the connection path is up.
51	ENETUNREACH	Connect	The network cannot be reached.	Ensure that the target application is active.
52	ENETRESET	All	The network dropped a connection on a reset.	Reestablish the connection between the applications.
53	ECONNABORTED	All	The software caused a connection abend.	Reestablish the connection between the applications.
54	ECONNRESET	All	The connection to the destination host is not available.	N/A
54	ECONNRESET	Send Write	The connection to the destination host is not available.	The socket is closing. Issue Send() or Write() before closing the socket.
55	ENOBUFS	All	No buffer space is available.	Check the application for massive storage allocation call.
55	ENOBUFS	Accept	Not enough buffer space is available to create the new socket.	Call your system administrator.
55	ENOBUFS	IOCTL (SIOCGPARTNERINFO)	The buffer size provided is too small.	Create a larger input buffer based on the value returned in the PI_Buflen field.
55	ENOBUFS	IOCTL (SIOCSAPPLDATA)	There is no storage available to store the associated data.	Call your system administrator.
55	ENOBUFS	IOCTL (SIOCTTLSCCTL TTLS_Version1 requesting TTLS_RETURN_CERTIFICATE or TTLS_Version2 query)	The buffer size provided is too small.	For TTLS_Version1 use the returned certificate length to allocate a larger buffer and reissue IOCTL with the larger buffer.
55	ENOBUFS	IP_BLOCK_SOURCE, IP_ADD_SOURCE_MEMBERSHIP, MCAST_BLOCK_SOURCE, MCAST_JOIN_SOURCE_GROUP, SIOCSIPMSFILTER, SIOCSMSFILTER, setipv4sourcefilter, setsourcefilter	A maximum of 64 source filters can be specified per multicast address, interface address pair.	Remove unneeded source IP addresses and reissue the command.
55	ENOBUFS	Send Sendto Write	Not enough buffer space is available to send the new message.	Call your system administrator.
55	ENOBUFS	Takesocket	Not enough buffer space is available to create the new socket.	Call your system administrator.
56	EISCONN	Connect	The socket is already connected.	Correct the socket descriptor on Connect() or do not issue a Connect() twice for the socket.
57	ENOTCONN	All	The socket is not connected.	Connect the socket before communicating.

Table 24. Sockets ERRNOs (continued)

Error number	Message name	Socket API type	Error description	Programmer's response
57	ENOTCONN	IOCTL (SIOCGPARTNERINFO)	The requested socket is not connected.	Check and modify the socket descriptor, or reissue the IOCTL after the connect call from the client side or after the accept call from the server side.
57	ENOTCONN	IOCTL (SIOCTLTLCTL)	The socket is not connected.	Issue the SIOCTLTLCTL IOCTL only after the socket is connected.
58	ESHUTDOWN	All	A Send cannot be processed after socket shutdown.	Issue read/receive before shutting down the read side of the socket.
59	ETOOMANYREFS	All	There are too many references. A splice cannot be completed.	Call your system administrator.
59	ETOOMANYREFS	IP_ADD_MEMBERSHIP, IP_ADD_SOURCE_MEMBERSHIP, MCAST_JOIN_GROUP, MCAST_JOIN_SOURCE_GROUP, IPV6_JOIN_GROUP	A maximum of 20 multicast groups per single UDP socket or a maximum of 256 multicast groups per single RAW socket can be specified.	Remove unneeded multicast groups and reissue the command.
60	ETIMEDOUT	Connect	The connection timed out before it was completed.	Ensure the server application is available.
61	ECONNREFUSED	Connect	The requested connection was refused.	Ensure server application is available and at specified port.
62	ELOOP	All	There are too many symbolic loop levels.	Reduce symbolic links to specified file.
63	ENAMETOOLONG	All	The file name is too long.	Reduce size of specified file name.
64	EHOSTDOWN	All	The host is down.	Restart specified host.
65	EHOSTUNREACH	All	There is no route to the host.	Set up network path to specified host and verify that host name is valid.
66	ENOTEMPTY	All	The directory is not empty.	Clear out specified directory and reissue call.
67	EPROCLIM	All	There are too many processes in the system.	Decrease the number of processes or increase the process limit.
68	EUSERS	All	There are too many users on the system.	Decrease the number of users or increase the user limit.
69	EDQUOT	All	The disk quota has been exceeded.	Call your system administrator.
70	ESTALE	All	An old NFS** data set handle was found.	Call your system administrator.
71	EREMOTE	All	There are too many levels of remote in the path.	Call your system administrator.
72	ENOSTR	All	The device is not a stream device.	Call your system administrator.
73	ETIME	All	The timer has expired.	Increase timer values or reissue function.
73	ETIME	IOCTL (SIOCGPARTNERINFO)	The wait time for the request has expired, possibly as the result of network problems.	<p>Retry the request.</p> <p>Restriction: You cannot use a select mask to determine when an IOCTL is complete, because an IOCTL is not affected by whether the socket is running in blocking or nonblocking mode. If the IOCTL times out, reissue the IOCTL to retrieve the partner security credentials.</p>
74	ENOSR	All	There are no more stream resources.	Call your system administrator.
75	ENOMSG	All	There is no message of the desired type.	Call your system administrator.
76	EBADMSG	All	The system cannot read the message.	Verify that z/OS Communications Server installation was successful and that message files were properly loaded.
77	EIDRM	All	The identifier has been removed.	Call your system administrator.
78	EDEADLK	All	A deadlock condition has occurred.	Call your system administrator.
78	EDEADLK	Select Selectex	None of the sockets in the socket descriptor sets are either AF_INET or AF_IUCV sockets and there is no timeout value or no ECB specified. The select/selectex would never complete.	Correct the socket descriptor sets so that an AF_INET or AF_IUCV socket is specified. A timeout or ECB value can also be added to avoid the select/selectex from waiting indefinitely.
79	ENOLCK	All	No record locks are available.	Call your system administrator.

Table 24. Sockets ERRNOs (continued)

Error number	Message name	Socket API type	Error description	Programmer's response
80	ENONET	All	The requested machine is not on the network.	Call your system administrator.
81	ERREMOTE	All	The object is remote.	Call your system administrator.
82	ENOLINK	All	The link has been severed.	Release the sockets and reinitialize the client-server connection.
83	EADV	All	An ADVERTISE error has occurred.	Call your system administrator.
84	ESRMNT	All	An SRMOUNT error has occurred.	Call your system administrator.
85	ECOMM	All	A communication error has occurred on a Send call.	Call your system administrator.
86	EPROTO	All	A protocol error has occurred.	Call your system administrator.
86	EPROTO	IOCTL (SIOCTLSCTL requesting TTLS_RESET_SESSION, TTLS_RESET_CIPHER, TTLS_RESET_WRITE_CIPHER, TTLS_SEND_SESSION_TICKET, TTLS_STOP_CONNECTION, or TTLS_ALLOW_HSTIMEOUT)	<p>One of the following errors occurred:</p> <ul style="list-style-type: none"> A TTLS_INIT_CONNECTION request was not received for the connection. TTLS_STOP_CONNECTION was requested on a connection that has outstanding application data. For unread application data, the errno junior is JrTTLSStopReadDataPending. For unwritten application data, the errno junior is JrTTLSStopWriteDataPending. TTLS_RESET_CIPHER or TTLS_STOP_CIPHER was requested on a connection that is secured using SSL version 2. TTLS_RESET_WRITE_CIPHER or TTLS_SEND_SESSION_TICKET was requested on a connection that is secured using a protocol version less than TLS version 1.3. TTLS_ALLOW_HSTIMEOUT was requested but the policy has the HandshakeRole value client or the HandshakeTimeout value is 0. TTLS_SEND_SESSION_TICKET was requested but the policy has the HandshakeRole value client or GSK_SESSION_TICKET_SERVER_ENABLE value Off or the GSK_SESSION_TICKET_SERVER_COUNT value is not 0. 	<ul style="list-style-type: none"> Request TTLS_INIT_CONNECTION before requesting TTLS_RESET_SESSION, TTLS_RESET_CIPHER, TTLS_RESET_WRITE_CIPHER, or TTLS_SEND_SESSION_TICKET. Request TTLS_STOP_CONNECTION after all application data is cleared from the connection. For JrTTLSStopReadDataPending, read all available application data. For JrTTLSStopWriteDataPending, wait for all the outstanding application data to be written. Request TTLS_RESET_CIPHER or TTLS_STOP_CONNECTION only on connections secured using SSL version 3 or TLS version 1.0 or higher. Request TTLS_RESET_WRITE_CIPHER or TTLS_SEND_SESSION_TICKET only on connections secured using TLS version 1.3 or higher. Request TTLS_ALLOW_HSTIMEOUT only when the security type is TTLS_SEC_SERVER or higher and the HandshakeTimeout value is not 0. Request TTLS_SEND_SESSION_TICKET only when the security type is TTLS_SEC_SERVER or higher and when GSK_SESSION_TICKET_SERVER_ENABLE is On and GSK_SESSION_TICKET_SERVER_COUNT is 0.
87	EMULTIHOP	All	A multi-hop address link was attempted.	Call your system administrator.
88	EDOTDOT	All	A cross-mount point was detected. This is not an error.	Call your system administrator.
89	EREMCHG	All	The remote address has changed.	Call your system administrator.
90	ECONNCLOSED	All	The connection was closed by a peer.	Check that the peer is running.
113	EBADF	All	Socket descriptor is not in correct range. The maximum number of socket descriptors is set by MAXDESC(). The default range is 0–49.	Reissue function with corrected socket descriptor.
113	EBADF	Bind socket	The socket descriptor is already being used.	Correct the socket descriptor.
113	EBADF	Givesocket	The socket has already been given. The socket domain is not AF_INET.	Correct the socket descriptor.
113	EBADF	Select	One of the specified descriptor sets is an incorrect socket descriptor.	Correct the socket descriptor. Set on Select() or Selectex().
113	EBADF	Takesocket	The socket has already been taken.	Correct the socket descriptor.
113	EBADF	Accept	A Listen() has not been issued before the Accept().	Issue Listen() before Accept().
121	EINVAL	All	An incorrect argument was specified.	Check and correct all function parameters.

Table 24. Sockets ERRNOs (continued)

Error number	Message name	Socket API type	Error description	Programmer's response
121	EINVAL	IOCTL (SIOCSAPPLDATA)	<p>The input parameter is not a correctly formatted SetApplData structure.</p> <ul style="list-style-type: none"> The SetAD_eye1 value is not valid. The SetAD_ver value is not valid. The storage pointed to by SetAD_ptr does not contain a correctly formatted SetADcontainer structure. The SetAD_eye2 value is not valid. The SetAD_len value contains an incorrect length for the SetAD_ver version of the SetADcontainer structure. 	Check and correct all function parameters.
121	EINVAL	inet6_is_srcaddr	<ul style="list-style-type: none"> One or more invalid IPV6_ADDR_PREFERENCES flags were specified A scope ID was omitted for a link local IP address A scope ID was specified for an IP address that is not link-local The socket address length was not valid 	Correct the function parameters
122	ECLOSED			
126	ENMELONG			
134	ENOSYS	IOCTL	The function is not implemented	Either configure the system to support the ioctl command or remove the ioctl command from your program.
134	ENOSYS	IOCTL - siocgifnameindex	The TCP/IP stack processing the siocgifnameindex IOCTL is configured as a pure IPv4 TCP/IP stack. Additionally, UNIX System Services is configured to process as INET.	Either configure the system to support the ioctl command or remove the ioctl command from your program.
136	ENOTEMPT			
145	E2BIG	All	The argument list is too long.	Eliminate excessive number of arguments.
156	EMVSiNITIAL	All	<p>Process initialization error.</p> <p>This indicates an z/OS UNIX process initialization failure. This is usually an indication that a proper OMVS RACF segment is not defined for the user ID associated with application. The RACF OMVS segment might not be defined or might contain errors such as an improper HOME() directory specification.</p>	Attempt to initialize again. After ensuring that an OMVS Segment is defined, if the errno is still returned, call your MVS system programmer to have IBM service contacted.
157	EMISSED			
157	EMVSERR		An MVS environmental or internal error occurred.	
1002	EIBMSOCKOUTOFRANGE	Socket, Accept, Takesocket	A new socket cannot be created because the MAXSOC value, which is specified on the INITAPI call, has been reached.	<p>Take either one of the following actions:</p> <ul style="list-style-type: none"> Verify whether all open sockets are intended to be in use. Increase the MAXSOC value to a value that is appropriate for the current workload. If the default value is currently being used, you might be required to add the INITAPI call.
1003	EIBMSOCKINUSE	Socket	A socket number assigned by the client interface code is already in use.	Use a different socket descriptor.
1004	EIBMIUCVERR	All	The request failed because of an IUCV error. This error is generated by the client stub code.	Ensure IUCV/VMCF is functional.
1008	EIBMCONFLICT	All	This request conflicts with a request already queued on the same socket.	Cancel the existing call or wait for its completion before reissuing this call.
1009	EIBMCANCELLED	All	The request was canceled by the CANCEL call.	Informational, no action needed.
1011	EIBMBADTCPNAME	All	A TCP/IP name that is not valid was detected.	Correct the name specified in the IBM_TCPIIMAGE structure.

Table 24. Sockets ERRNOs (continued)

Error number	Message name	Socket API type	Error description	Programmer's response
1011	EIBMBADTCPNAME	Setibmopt	A TCP/IP name that is not valid was detected.	Correct the name specified in the IBM_TCPIIMAGE structure.
1011	EIBMBADTCPNAME	INITAPI	A TCP/IP name that is not valid was detected.	Correct the name specified on the IDENT option TCPNAME field.
1012	EIBMBADREQUESTCODE	All	A request code that is not valid was detected.	Contact your system administrator.
1013	EIBMBADCONNECTIONSTATE	All	A connection token that is not valid was detected; bad state.	Verify TCP/IP is active.
1014	EIBMUNAUTHORIZEDCALLER	All	An unauthorized caller specified an authorized keyword.	Ensure user ID has authority for the specified operation.
1015	EIBMBADCONNECTIONMATCH	All	A connection token that is not valid was detected. There is no such connection.	Verify TCP/IP is active.
1016	EIBMTCPABEND	All	An abend occurred when TCP/IP was processing this request.	Verify that TCP/IP has restarted.
1023	EIBMTERMERROR	All	Encountered a terminating error while processing.	Call your system administrator.
1026	EIBMINVDELETE	All	Delete requestor did not create the connection.	Delete the request from the process that created it.
1027	EIBMINVSOCKET	All	A connection token that is not valid was detected. No such socket exists.	Call your system programmer.
1028	EIBMINTVTCPCONNECTION	All	Connection terminated by TCP/IP. The token was invalidated by TCP/IP.	Reestablish the connection to TCP/IP.
1032	EIBMCALLINPROGRESS	All	Another call was already in progress.	Reissue after previous call has completed.
1036	EIBMNOACTIVETCP	All	TCP/IP is not installed or not active.	Correct TCP/IP name used.
1036	EIBMNOACTIVETCP	Select	EIBMNOACTIVETCP	Ensure TCP/IP is active.
1036	EIBMNOACTIVETCP	Getibmopt	No TCP/IP image was found.	Ensure TCP/IP is active.
1037	EIBMINTVSRBUSERDATA	All	The request control block contained data that is not valid.	Call your system programmer.
1038	EIBMINTVUSERDATA	All	The request control block contained user data that is not valid.	Check your function parameters and call your system programmer.
1040	EIBMSELECTEXPOST	SELECTEX	SELECTEX passed an ECB that was already posted.	Check whether the user's ECB was already posted.
1112	ECANCEL			
1162	ENOPARTNERINFO	IOCTL (SIOCGPARTNERINFO)	The partner resides in a TCP/IP stack running a release that is earlier than V1R12, or the partner is not in the same sysplex.	Ensure that both endpoints reside in TCP/IP stacks that are running V1R12 or any later release, or check and modify the socket descriptor. If the partner is not in the same sysplex, security credentials will not be returned.
2001	EINVALIDRXSOCKETCALL	REXX	A syntax error occurred in the RXSOCKET parameter list.	Correct the parameter list passed to the REXX socket call.
2002	ECONSOLEINTERRUPT	REXX	A console interrupt occurred.	Retry the task.
2003	ESUBTASKINVALID	REXX	The subtask ID is incorrect.	Correct the subtask ID on the INITIALIZE call.
2004	ESUBTASKALREADYACTIVE	REXX	The subtask is already active.	Issue the INITIALIZE call only once in your program.
2005	ESUBTASKNOTACTIVE	REXX	The subtask is not active.	Issue the INITIALIZE call before any other socket call.
2006	ESOCKETNOTALLOCATED	REXX	The specified socket or needed control block could not be allocated.	Increase the user storage allocation for this job.
2007	EMAXSOCKETSREACHED	REXX	The maximum number of sockets has been reached.	Increase the number of allocate sockets, or decrease the number of sockets used by your program.
2009	ESOCKETNOTDEFINED	REXX	The socket is not defined.	Issue the SOCKET call before the call that fails.
2011	EDOMAINSERVERFAILURE	REXX	A Domain Name Server failure occurred.	Call your MVS system programmer.
2012	EINVALIDNAME	REXX	An incorrect <i>name</i> was received from the TCP/IP server.	Call your MVS system programmer.

Table 24. Sockets ERRNOs (continued)

Error number	Message name	Socket API type	Error description	Programmer's response
2013	EINVALIDCLIENTID	REXX	An incorrect <i>clientid</i> was received from the TCP/IP server.	Call your MVS system programmer.
2014	ENIVALIDFILENAME	REXX	An error occurred during NUCEXT processing.	Specify the correct translation table file name, or verify that the translation table is valid.
2016	EHOSTNOTFOUND	REXX	The host is not found.	Call your MVS system programmer.
2017	EIPADDRNOTFOUND	REXX	Address not found.	Call your MVS system programmer.
2019	ENORECOVERY	REXX	A non-recoverable failure occurred during the Resolver's processing of the GETHOSTBYADDR or GETHOSTBYNAME call.	Contact the IBM support center.
2020	EINVALIDCOMBINATION	REXX	An invalid combination of IPV6_ADDR_PREFERENCES flags was received from the caller.	Correct the specified flags
2021	EOPTNAMEMISMATCH	REXX	The caller specified an OPTNAME that is invalid for the LEVEL that it specified.	Correct either the OPTNAME or the LEVEL.
2022	EFLAGSMISMATCH	REXX	The caller issued a GETADDRINFO with conflicting FLAGS and EFLAGS parameters: either AI_EXT_FLAGS was specified with a null EFLAGS, or AI_EXT_FLAGS was not specified but EFLAGS was not null.	Correct either the FLAGS parameter or the EFLAGS parameter. A non-null EFLAGS should be specified if and only if AI_EXT_FLAGS is specified in the FLAGS.
2051	EFORMATERROR	REXX	The name server was unable to interpret the query	Contact the IBM support center.
3412	ENODATA		Message does not exist.	
3416	ELINKED		Stream is linked.	
3419	ERECURSE		Recursive attempt rejected.	
3420	EASYNC		Asynchronous I/O scheduled. This is a normal, internal event that is NOT returned to the user.	
3448	EUNATCH		The protocol required to support the specified address family is not available.	
3464	ETERM		Operation terminated.	
3474	EUNKNOWN		Unknown system state.	
3495	EBADOBJ		You attempted to reference an object that does not exist.	
3513	EOUTOFSTATE		Protocol engine has received a command that is not acceptable in its current state.	

z/OS UNIX return codes

All return codes not listed in either “Sockets return codes (ERRNOs)” on page 743 or “Sockets extended ERRNOs” on page 755 are z/OS UNIX error condition codes that are not translated to a TCP/IP errno. This is an errno that is received from z/OS UNIX. These errnos are found in the SYS1.MACLIB(BPXYERNO) and are defined in [z/OS UNIX System Services Messages and Codes](#).

For more information about z/OS UNIX error codes, see [z/OS UNIX System Services Messages and Codes](#).

Additional return codes

The following table contains the error condition codes that are returned in the ERRNO field by the API when you use the sockets extended interfaces. The RETCODE field contains a -1 when an error condition is returned.

Sockets extended ERRNOs

Table 25. Sockets extended ERRNOs

Error code	Problem description	System action	Programmer's response
10100	An ESTAE macro did not complete normally.	End the call.	Call your MVS system programmer.
10101	A STORAGE OBTAIN failed.	End the call.	Increase MVS storage in the application's address space.
10108	The first call issued was not a valid first call.	End the call.	Almost all sockets programs that are written in COBOL, PL/I, or assembler language must issue the INITAPI call before they issue other sockets calls.
10110	LOAD of EZBSOH03 (alias EZASOH03) failed.	End the call.	Call the IBM Software Support Center.
10154	Errors were found in the parameter list for an IOCTL call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the IOCTL call. You might have incorrect sequencing of socket calls.
10155	The length parameter for an IOCTL call is less than or equal to 0.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the IOCTL call. You might have incorrect sequencing of socket calls.
10156	The length parameter for an IOCTL call is 3200 (32 x 100).	Disable the subtask for interrupts. Return an error code to the caller.	Correct the IOCTL call. You might have incorrect sequencing of socket calls.
10159	A 0 or negative data length was specified for a READ or READV call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the length in the READ call.
10161	The REQARG parameter in the IOCTL parameter list is 0.	End the call.	Correct the program.
10163	A 0 or negative data length was found for a RECV, RECVFROM, or RECVMSG call.	Disable the subtask for interrupts. Sever the DLC path. Return an error code to the caller.	Correct the data length.
10167	The descriptor set size for a SELECT or SELECTEX call is less than or equal to 0.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the SELECT or SELECTEX call. You might have incorrect sequencing of socket calls.
10168	The descriptor set size <i>in bytes</i> for a SELECT or SELECTEX call is greater than 8192. A number greater than the maximum number of allowed sockets (65534 is the maximum) has been specified.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the descriptor set size.

Table 25. Sockets extended ERRNOs (continued)

Error code	Problem description	System action	Programmer's response
10170	A 0 or negative data length was found for a SEND or SENDMSG call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the data length in the SEND call.
10174	A 0 or negative data length was found for a SENDTO call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the data length in the SENDTO call.
10178	The SETSOCKOPT option length is less than the minimum length.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the OPTLEN parameter.
10179	The SETSOCKOPT option length is greater than the maximum length.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the OPTLEN parameter.
10184	A data length of 0 was specified for a WRITE call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the data length in the WRITE call.
10186	A negative data length was specified for a WRITE or WRITEV call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the data length in the WRITE call.
10190	The GETHOSTNAME option length is not in the range of 1–255.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the length parameter.
10193	The SETSOCKOPT or GETSOCKOPT option length is shorter than the minimum length or longer than the maximum length.	End the call.	Correct the length parameter.
10197	The application issued an INITAPI call after the connection was already established.	Bypass the call.	Correct the logic that produces the INITAPI call that is not valid.
10198	The maximum number of sockets specified for an INITAPI exceeds 65535.	Return to the user.	Correct the INITAPI call.
10200	The first call issued was not a valid first call.	End the call.	Almost all sockets programs that are written in COBOL, PL/I, or assembler language must issue the INITAPI call before they issue other sockets calls.
10202	The RETARG parameter in the IOCTL call is 0.	End the call.	Correct the parameter list. You might have incorrect sequencing of socket calls.

Table 25. Sockets extended ERRNOs (continued)

Error code	Problem description	System action	Programmer's response
10203	The requested socket number is a negative value.	End the call.	Correct the requested socket number.
10205	The requested socket number is a duplicate.	End the call.	Correct the requested socket number.
10208	The NAMELEN parameter for a GETHOSTBYNAME call was not specified.	End the call.	Correct the NAMELEN parameter. You might have incorrect sequencing of socket calls.
10209	The NAME parameter on a GETHOSTBYNAME call was not specified.	End the call.	Correct the NAME parameter. You might have incorrect sequencing of socket calls.
10210	The HOSTENT parameter on a GETHOSTBYNAME or GETHOSTBYADDR call was not specified.	End the call.	Correct the HOSTENT parameter. You might have incorrect sequencing of socket calls.
10211	The HOSTADDR parameter on a GETHOSTBYNAME or GETHOSTBYADDR call is incorrect.	End the call.	Correct the HOSTADDR parameter. You might have incorrect sequencing of socket calls.
10212	The resolver program failed to load correctly for a GETHOSTBYNAME or GETHOSTBYADDR call.	End the call.	Check the JOBLIB, STEPLIB, and linklib datasets and rerun the program.
10213	Not enough storage is available to allocate the HOSTENT structure.	End the call.	Increase the user storage allocation for this job.
10214	The HOSTENT structure was not returned by the resolver program.	End the call.	Ensure that the domain name server is available. This can be a nonerror condition indicating that the name or address specified in a GETHOSTBYADDR or GETHOSTBYNAME call could not be matched.
10215	The APITYPE parameter on an INITAPI call instruction was not 2 or 3.	End the call.	Correct the APITYPE parameter.
10218	The application programming interface (API) cannot locate the specified TCP/IP.	End the call.	Ensure that an API that supports the performance improvements related to CPU conservation is installed on the system and verify that a valid TCP/IP name was specified on the INITAPI call. This error call might also mean that EZASOKIN could not be loaded.
10219	The NS parameter is greater than the maximum socket for this connection.	End the call.	Correct the NS parameter on the ACCEPT, SOCKET or TAKESOCKET call.

Table 25. Sockets extended ERRNOs (continued)

Error code	Problem description	System action	Programmer's response
10221	The AF parameter of a SOCKET call is not AF_INET.	End the call.	Set the AF parameter equal to AF_INET.
10222	The SOCTYPE parameter of a SOCKET call must be stream, datagram, or raw (1, 2, or 3).	End the call.	Correct the SOCTYPE parameter.
10223	No ASYNC parameter specified for INITAPI with APITYPE=3 call.	End the call.	Add the ASYNC parameter to the INITAPI call.
10224	The IOVCNT parameter is less than or equal to 0, for a READV, RECVMSG, SENDMSG, or WRITEV call.	End the call.	Correct the IOVCNT parameter.
10225	The IOVCNT parameter is greater than 120, for a READV, RECVMSG, SENDMSG, or WRITEV call.	End the call.	Correct the IOVCNT parameter.
10226	Not valid COMMAND parameter specified for a GETIBMOPT call.	End the call.	Correct the COMMAND parameter of the GETIBMOPT call.
10229	A call was issued on an APITYPE=3 connection without an ECB or REQAREA parameter.	End the call.	Add an ECB or REQAREA parameter to the call.
10300	Termination is in progress for either the CICS transaction or the socket interface.	End the call.	None.
10330	A SELECT call was issued without a MAXSOC value and a TIMEOUT parameter.	End the call.	Correct the call by adding a TIMEOUT parameter.
10331	A call that is not valid was issued while in SRB mode.	End the call.	Get out of SRB mode and reissue the call.
10332	A SELECT call is invoked with a MAXSOC value greater than that which was returned in the INITAPI function (MAXSNO field).	End the call.	Correct the MAXSOC parameter and reissue the call.
10334	An error was detected in creating the data areas required to process the socket call.	End the call.	Call the IBM Software Support Center.
10335	An INITAPI or first call was issued by using a TIE that another task used.	End the call.	Change the application to allocate a new TIE or to ensure that a TERMAPI is done before the TIE is reused.

Table 25. Sockets extended ERRNOs (continued)

Error code	Problem description	System action	Programmer's response
10999	An abend has occurred in the subtask.	Write message EZY1282E to the system console. End the subtask and post the TRUE ECB.	If the call is correct, call your system programmer.
20000	An unknown function code was found in the call.	End the call.	Correct the SOC-FUNCTION parameter.
20001	The call passed an incorrect number of parameters.	End the call.	Correct the parameter list.
20002	The user ID associated with the program linking EZACIC25 does not have the proper authority to execute a CICS EXTRACT EXIT.	End the call.	Start the CICS socket interface before executing this call.
20003	The CICS socket interface is not in operation.	End the call.	Contact the CICS system programmer. Ensure that the user ID being used is permitted to have at least UPDATE access to the EXITPROGRAM resource.
20004	The CICS socket TRUE failed to suspend the task.	End the call.	Call the IBM Software Support Center.
20005	The socket task was purged by CICS while the task was being suspended by the CICS socket TRUE.	End the call.	None.

User abend U4093

An abend U4093 indicates that a sockets extended call that is not valid has been detected. It is issued by EZASOKET following a call to EZASOKFN if EZASOKFN has detected an error in the socket call parameter list. The registers at the time of the abend are:

- R2 contains the address of the save area containing the calling program registers.
- R11 contains the error code passed to EZASOKET by EZASOKFN.

Code

Description

X'4E20' (20000)

Indicates EZASOKFN could not find the requested CALL function name.

X'4E21' (20001)

Indicates that EZASOKFN found an incorrect number of parameters in the parameter list for the requested function.

- R12 contains the address of the incorrect parameter list.

Figure 231 on page 760 is an example of abend U4093:

```

USER COMPLETION CODE=4093
TIME=15.01.58 SEQ=00074 CPU=0000 ASID=000E
PSW AT TIME OF ERROR 078D1000 80018F14 ILC 2 INTC 0D
ACTIVE LOAD MODULE=DLSV2AS2 ADDRESS=00018670 OFFSET=000008A4
DATA AT PSW 00018F0E - 00181610 0A0D4100 35185000
GPR 0-3 80000000 80000FFD 000189E4 00018DC0
GPR 4-7 00019DC0 00018CE0 00018AA6 00018D18
GPR 8-11 00013780 00019378 00019088 00004E21
GPR 12-15 000187D4 0001902C 80018EF4 00004E21

```

Figure 231. Example of abend U4093

Appendix C. Address family cross reference

This information contains AF_INET, AF_INET6, and AF_IUCV address family cross reference information for the following APIs:

- TCP/IP C socket API
- X/Open Transport Interface
- Macro API for IPv4 or IPv6 that is written in z/OS assembler language
- Call instruction API for IPv4 or IPv6 socket applications
- z/OS Communications Server socket API for REXX
- Pascal language for IPv4 socket API

Address families define different styles of addressing. All hosts in the same addressing family understand and use the same method for addressing socket endpoints. TCP/IP supports the following addressing families:

- AF_INET
- AF_INET6
- AF_IUCV

The AF_INET and AF_INET6 families both define addressing in the internet domain. The AF_IUCV family defines addressing in the IUCV domain. In the IUCV domain, address spaces can use the socket interface to communicate with other address spaces on the same host.

The INET, INET6, and IUCV column entries are:

yes

The call applies to this address family.

no

If you use this call with this address family, an error is returned.

n/a

If you use this call with this address family, no error is returned and the call is not processed.

blank

The call does not apply to this API.

Notes:

1. Pascal API supports only AF_INET address family.
2. XTI API supports only AF_INET address family.
3. INET6 is not supported.

Table 26. C socket address families cross reference

Application Programming Interface (API)		
Function	C SOCKETS	
	INET	IUCV
accept()	yes	yes
bind()	yes	yes
close()	yes	yes
connect()	yes	yes
endhostent()	yes	n/a

Table 26. C socket address families cross reference (continued)

Application Programming Interface (API)		
Function	C SOCKETS	
	INET	IUCV
endnetent()	yes	n/a
endprotoent()	yes	n/a
endservent()	yes	n/a
fcntl()	yes	no
getclientid()	yes	no
getdtablesize()	yes	yes
gethostbyaddr()	yes	no
gethostbyname()	yes	n/a
gethostent()	yes	n/a
gethostid()	yes	no
gethostname()	yes	no
getibmopt()	yes	no
getibmsockopt()	yes	no
getnetbyaddr()	yes	n/a
getnetbyname()	yes	n/a
getnetent()	yes	n/a
getpeername()	yes	yes
getprotobyname()	yes	n/a
getprotobynumber()	yes	n/a
getprotoent()	yes	n/a
getservbyname()	yes	n/a
getservbyport()	yes	n/a
getservent()	yes	n/a
getsockname()	yes	yes
getsockopt()	yes	no
givesocket()	yes	no
htonl()	yes	n/a
htons()	yes	n/a
inet_addr()	yes	n/a
inet_anaof()	yes	n/a
inet_makeaddr()	yes	n/a
inet_netof()	yes	n/a

Table 26. C socket address families cross reference (continued)

Application Programming Interface (API)		
Function	C SOCKETS	
	INET	IUCV
inet_network()	yes	n/a
inet_ntoa()	yes	n/a
ioctl()	yes	no
listen()	yes	yes
maxdesc()	yes	yes
ntohl()	yes	n/a
ntohs()	yes	n/a
read()	yes	yes
readv()	yes	yes
recv()	yes	yes
recvfrom()	yes	yes
recvmsg()	yes	yes
select()	yes	yes
selectex()	yes	yes
send()	yes	yes
sendmsg()	yes	no
sendto()	yes	no
setibmopt()	yes	no
setibmsockopt()	yes	no
sethostent()	yes	n/a
setnetent()	yes	n/a
setprotoent()	yes	n/a
setservent()	yes	n/a
setsockopt()	yes	no
shutdown()	yes	yes
sock_debug()	yes	yes
sock_do_teststor()	yes	yes
socket()	yes	yes
takesocket()	yes	no
tcperror()	yes	yes
write()	yes	yes
writv()	yes	yes

Note: In the following table, IUCV is not supported.

Table 27. MACRO, CALL, REXX, socket address families cross reference

Application Programming Interface (API)						
COMMAND	MACRO		CALL		REXX	
	INET	INET6	INET	INET6	INET	INET6
ACCEPT	yes	yes	yes	yes	yes	yes
BIND	yes	yes	yes	yes	yes	yes
BIND2ADDRSEL	no	yes	no	yes	no	yes
CANCEL	yes	yes				
CLOSE	yes	yes	yes	yes	yes	yes
CONNECT	yes	yes	yes	yes	yes	yes
FCNTL	yes	yes	yes	yes	yes	yes
FREEADDRINFO	yes	yes	yes	yes		
GETADDRINFO	yes	yes	yes	yes	yes	yes
GETCLIENTID	yes	yes	yes	yes	yes	yes
GETDOMAINNAME					yes	yes
GETHOSTBYADDR	yes	yes	yes	yes	yes	yes
GETHOSTBYNAME	yes	yes	yes	yes	yes	yes
GETHOSTID	yes	yes	yes	yes	yes	yes
GETHOSTNAME	yes	yes	yes	yes	yes	yes
GETIBMOPT	yes	yes	yes	yes		
GETNAMEINFO	yes	yes	yes	yes	yes	yes
GETPEERNAME	yes	yes	yes	yes	yes	yes
GETPROTOBYNAME					n/a	n/a
GETPROTOBYNUMBER					yes	yes
GETSERVBYNAME					yes	yes
GETSERVBYPOR					yes	yes
GETSOCKNAME	yes	yes	yes	yes	yes	yes
GETSOCKOPT see Table 28 on page 765 for exceptions.	yes	yes	yes	yes	yes	yes
GIVESOCKET	yes	yes	yes	yes	yes	yes
GLOBAL	yes	yes	yes	yes		
INET6_IS_SRCADDR	no	yes	no	yes	no	yes
INITAPI	yes	yes	yes	yes		
IOCTL see Table 28 on page 765 for exceptions.	yes	yes	yes	yes	yes	yes
LISTEN	yes	yes	yes	yes	yes	yes

Table 27. MACRO, CALL, REXX, socket address families cross reference (continued)

Application Programming Interface (API)						
COMMAND	MACRO		CALL		REXX	
	INET	INET6	INET	INET6	INET	INET6
NTOP	yes	yes	yes	yes		
PTON	yes	yes	yes	yes		
READ	yes	yes	yes	yes	yes	yes
READV	yes	yes	yes	yes		
RECV	yes	yes	yes	yes	yes	yes
RECVFROM	yes	yes	yes	yes	yes	yes
RECVMSG	yes	yes	yes	yes		
RESOLVE					yes	yes
SELECT	yes	yes	yes	yes	yes	yes
SELECTEX	yes	yes	yes	yes		
SEND	yes	yes	yes	yes	yes	yes
SENDMSG	yes	yes	yes	yes		
SENDTO	yes	yes	yes	yes	yes	yes
SETSOCKOPT see Table 28 on page 765 for exceptions.	yes	yes	yes	yes	yes	yes
SHUTDOWN	yes	yes	yes	yes	yes	yes
SOCKET	yes	yes	yes	yes	yes	yes
TAKESOCKET	yes	yes	yes	yes	yes	yes
TASK	yes	yes	yes	yes		
TERMAPI	yes	yes	yes	yes		
VERSION					yes	yes
WRITE	yes	yes	yes	yes	yes	yes
WRITEV	yes	yes	yes	yes		

Table 28. MACRO, CALL, REXX, exceptions

Application Programming Interface (API)						
COMMAND	MACRO		CALL		REXX	
	INET	INET6	INET	INET6	INET	INET6
GETSOCKOPT exceptions						
IP_MULTICAST_IF	yes	no	yes	no	yes	no
IP_MULTICAST_LOOP	yes	no	yes	no	yes	no
IP_MULTICAST_TTL	yes	no	yes	no	yes	no
IPV6_ADDR_PREFERENCES	no	yes	no	yes	no	yes

Table 28. MACRO, CALL, REXX, exceptions (continued)

Application Programming Interface (API)						
COMMAND	MACRO		CALL		REXX	
	INET	INET6	INET	INET6	INET	INET6
IPV6_MULTICAST_HOPS	no	yes	no	yes	no	yes
IPV6_MULTICAST_IF	no	yes	no	yes	no	yes
IPV6_MULTICAST_LOOP	no	yes	no	yes	no	yes
IPV6_UNICAST_HOPS	no	yes	no	yes	no	yes
IPV6_V6ONLY	no	yes	no	yes	no	yes
SETSOCKOPT exceptions						
IP_ADD_MEMBERSHIP	yes	no	yes	no	yes	no
IP_ADD_SOURCE_MEMBERSHIP	yes	no	yes	no	yes	no
IP_BLOCK_SOURCE	yes	no	yes	no	yes	no
IP_DROP_MEMBERSHIP	yes	no	yes	no	yes	no
IP_DROP_SOURCE_MEMBERSHIP	yes	no	yes	no	yes	no
IP_MULTICAST_IF	yes	no	yes	no	yes	no
IP_MULTICAST_LOOP	yes	no	yes	no	yes	no
IP_MULTICAST_TTL	yes	no	yes	no	yes	no
IP_UNBLOCK_SOURCE	yes	no	yes	no	yes	no
IPV6_ADDR_PREFERENCES	no	yes	no	yes	no	yes
IPV6_JOIN_GROUP	no	yes	no	yes	no	yes
IPV6_LEAVE_GROUP	no	yes	no	yes	no	yes
IPV6_MULTICAST_HOPS	no	yes	no	yes	no	yes
IPV6_MULTICAST_IF	no	yes	no	yes	no	yes
IPV6_MULTICAST_LOOP	no	yes	no	yes	no	yes
IPV6_UNICAST_HOPS	no	yes	no	yes	no	yes
IPV6_V6ONLY	no	yes	no	yes	no	yes
IOCTL exceptions						
SIOCGHOMEIF6	no	yes	no	yes		
SIOCGIPMSFILTER	yes	no	yes	no	yes	no
SIOCSIPMSFILTER	yes	no	yes	no	yes	no

Appendix D. GETSOCKOPT/SETSOCKOPT command values

You can use the following table to determine the decimal or hexadecimal value associated with the GETSOCKOPT/SETSOCKOPT OPTNAMES supported by the APIs discussed in this document.

The command names are shown with underscores for the assembler language. The underscores should be changed to dashes if using the COBOL programming language.

Languages that cannot easily handle binary values, such as COBOL, should use the decimal value associated with the command where necessary.

The hexadecimal value can be used in Macro, Assembler and PL/I programs.

Table 29. GETSOCKOPT/SETSOCKOPT command values for Macro, Assembler, COBOL and PL/I		
Command name	Decimal value	Hex value
IP_ADD_MEMBERSHIP	1048581	X'00100005'
IP_ADD_SOURCE_MEMBERSHIP	1048588	X'0010000C'
IP_BLOCK_SOURCE	1048586	X'0010000A'
IP_DROP_MEMBERSHIP	1048582	X'00100006'
IP_DROP_SOURCE_MEMBERSHIP	1048589	X'0010000D'
IP_MULTICAST_IF	1048583	X'00100007'
IP_MULTICAST_LOOP	1048580	X'00100004'
IP_MULTICAST_TTL	1048579	X'00100003'
IP_UNBLOCK_SOURCE	1048587	X'0010000B'
IPV6_ADDR_PREFERENCES	65568	X'00010020'
IPV6_JOIN_GROUP	65541	X'00010005'
IPV6_LEAVE_GROUP	65542	X'00010006'
IPV6_MULTICAST_HOPS	65545	X'00010009'
IPV6_MULTICAST_IF	65543	X'00010007'
IPV6_MULTICAST_LOOP	65540	X'00010004'
IPV6_UNICAST_HOPS	65539	X'00010003'
IPV6_V6ONLY	65546	X'0001000A'
MCAST_BLOCK_SOURCE	1048620	X'0010002C'
MCAST_JOIN_GROUP	1048616	X'00100028'
MCAST_JOIN_SOURCE_GROUP	1048618	X'0010002A'
MCAST_LEAVE_GROUP	1048617	X'00100029'
MCAST_LEAVE_SOURCE_GROUP	1048619	X'0010002B'
MCAST_UNBLOCK_SOURCE	1048621	X'0010002D'
SO_BROADCAST	32	X'00000020'

Table 29. GETSOCKOPT/SETSOCKOPT command values for Macro, Assembler, COBOL and PL/I (continued)

Command name	Decimal value	Hex value
SO_ERROR	4103	X'00001007'
SO_LINGER	128	X'00000080'
SO_KEEPALIVE	8	X'00000008'
SO_OOINLINE	256	X'00000100'
SO_RCVBUF	4098	X'00001002'
SO_RCVTIMEO	4102	X'00001006'
SO_REUSEADDR	4	X'00000004'
SO_SNDBUF	4097	X'00001001'
SO_SNDTIMEO	4101	X'00001005'
SO_TYPE	4104	X'00001008 '
TCP_KEEPALIVE	2147483654	X'80000008 '
TCP_NODELAY	2147483649	X'80000001'

Table 30. GETSOCKOPT/SETSOCKOPT optname value for C programs

Option name	Decimal value
IP_ADD_MEMBERSHIP	5
IP_ADD_SOURCE_MEMBERSHIP	12
IP_BLOCK_SOURCE	10
IP_DROP_MEMBERSHIP	6
IP_DROP_SOURCE_MEMBERSHIP	13
IP_MULTICAST_IF	7
IP_MULTICAST_LOOP	4
IP_MULTICAST_TTL	3
IP_UNBLOCK_SOURCE	11
MCAST_BLOCK_SOURCE	44
MCAST_JOIN_GROUP	40
MCAST_JOIN_SOURCE_GROUP	42
MCAST_LEAVE_GROUP	41
MCAST_LEAVE_SOURCE_GROUP	43
MCAST_UNBLOCK_SOURCE	45
SO_ACCEPTCONN	2
SO_BROADCAST	32
SO_CLUSTERCONNTYPE	16385
SO_DEBUG	1

<i>Table 30. GETSOCKOPT/SETSOCKOPT optname value for C programs (continued)</i>	
Option name	Decimal value
SO_ERROR	4103
SO_KEEPALIVE	8
SO_LINGER	128
SO_OOBINLINE	256
SO_RCVBUF	4098
SO_REUSEADDR	4
SO_SNDBUF	4097
SO_TYPE	4104
TCP_KEEPALIVE	8
TCP_NODELAY	1

Appendix E. Abbreviations and acronyms

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
AT-TLS	Application Transparent Transport Layer Security
AUI	Attachment Unit Interface
BIOS	Basic Input/Output System
BNC	Bayonet Neill-Concelman
CCITT	Comite Consultatif International Telegraphique et Telephonique. The International Telegraph and Telephone Consultative Committee
CETI	Continuously Executing Transfer Interface
CLAW	Common Link Access to Workstation
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

ELANS
IBM Ethernet LAN Subsystem

ESCON
Enterprise Systems Connection

FAT
File Allocation Table

FDDI
Fiber Distributed Data Interface

FTAM
File Transfer Access Management

FTP
File Transfer Protocol

FTP API
File Transfer Protocol Applications Programming Interface

GCS
Group Control System

GDDM
Graphical Data Display Manager

GDF
Graphics Data File

HCH**
HYPERchannel device**

HIPPI
High Performance Parallel Interface

HPFS
High Performance File System

ICAT
Installation Configuration Automation Tool

ICMP
Internet Control Message Protocol

IEEE
Institute of Electrical and Electronic Engineers

IETF
Internet Engineering Task Force

ILANS
IBM Token-Ring LAN Subsystem

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

LCS
IBM LAN Channel Station

LPD
Line Printer Daemon

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

MHS
Message Handling System

MIB
Management Information Base

MIH
Missing Interrupt Handler

MILNET
Military Network

MTU
Maximum Transmission Unit

MVS
Multiple Virtual Storage

MX
Mail Exchange

NCP
Network Control Program

NCS
Network Computing System

NDIS
Network Driver Interface Specification

NFS**
Network File System**

NIC
Network Information Center

NLS
multicultural support

NSFNET
National Science Foundation Network

OS/2
Operating System/2®

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
program call

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
System Application Architecture

SBCS
Single Byte Character Set

SDLC
Synchronous Data Link Control

SLIP
Serial Line Internet Protocol

SMI
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

TFTP

Trivial File Transfer 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/SP

Virtual Machine/System Product

VM/XA

Virtual Machine/Extended Architecture

VTAM

Virtual Telecommunications Access Method

WAN

Wide Area Network

XDR

eXternal Data Representation

Appendix F. GETNAMEINFO flags and returned information examples

Flag name	Hexadecimal value
NI_NOFQDN	01
NI_NUMERICHOST	02
NI_NAMEREQD	04
NI_NUMERICSERV	08
NI_DGRAM	10
NI_NUMERICSCOPE	20

The following table shows returned results of information for all 63 possible combinations of flags (x'01' to x'3F') using a REXX socket call for GETNAMEINFO:

```
af      = 2                /* AF_INET */
portnum = 23              /* port number */
ipaddr  = 10.x.y.z        /* IP address */
flags   = one or more flag name separated by blanks

Result  = SOCKET( 'GETNAMEINFO', af ipaddr portnum, flags)
```

Flags (in hexadecimal value)	Result of returned information (return code, data1, data2)
01, 05, 21, or 25	0 mvshost telnet
02, 03, 22, or 23	0 10.x.y.z telnet
04, 20, or 24	0 mvshost.tcp.raleigh.ibm.com telnet
06, 07, 0E, 0F, 16, 17, 1E, 1F, 26, 27, 2E, 2F, 36, 37, 3E, or 3F	3 EAI_FAIL No recovery - resolver
08, 0C, 10, 14, 18, 1C, 28, 2C, 30, 34, 38, or 3C	0 mvshost.tcp.raleigh.ibm.com 23
09, 0D, 11, 15, 19, 1D, 29, 2D, 31, 35, 39, or 3D	0 mvshost 23
0A, 0B, 18, 19, 1A, 1B, 2A, 2B, 32, 33, 3A, or 3B	0 10.x.y.z 23

Appendix G. Related protocol specifications

This appendix lists the related protocol specifications (RFCs) for TCP/IP. The Internet Protocol suite is still evolving through requests for comments (RFC). New protocols are being designed and implemented by researchers and are brought to the attention of the Internet community in the form of RFCs. Some of these protocols are so useful that they become recommended protocols. That is, all future implementations for TCP/IP are recommended to implement these particular functions or protocols. These become the *de facto* standards, on which the TCP/IP protocol suite is built.

RFCs are available at <http://www.rfc-editor.org/rfc.html>.

Draft RFCs that have been implemented in this and previous Communications Server releases are listed at the end of this topic.

Many features of TCP/IP Services are based on the following RFCs:

RFC

Title and Author

RFC 652

Telnet output carriage-return disposition option D. Crocker

RFC 653

Telnet output horizontal tabstops option D. Crocker

RFC 654

Telnet output horizontal tab disposition option D. Crocker

RFC 655

Telnet output formfeed disposition option D. Crocker

RFC 657

Telnet output vertical tab disposition option D. Crocker

RFC 658

Telnet output linefeed disposition D. Crocker

RFC 698

Telnet extended ASCII option T. Mock

RFC 726

Remote Controlled Transmission and Echoing Telnet option J. Postel, D. Crocker

RFC 727

Telnet logout option M.R. Crispin

RFC 732

Telnet Data Entry Terminal option J.D. Day

RFC 733

Standard for the format of ARPA network text messages D. Crocker, J. Vittal, K.T. Pogran, D.A. Henderson

RFC 734

SUPDUP Protocol M.R. Crispin

RFC 735

Revised Telnet byte macro option D. Crocker, R.H. Gumpertz

RFC 736

Telnet SUPDUP option M.R. Crispin

RFC 749

Telnet SUPDUP—Output option B. Greenberg

RFC 765

File Transfer Protocol specification J. Postel

RFC 768

User Datagram Protocol J. Postel

RFC 779

Telnet send-location option E. Killian

RFC 791

Internet Protocol J. Postel

RFC 792

Internet Control Message Protocol J. Postel

RFC 793

Transmission Control Protocol J. Postel

RFC 820

Assigned numbers J. Postel

RFC 823

DARPA Internet gateway R. Hinden, A. Sheltzer

RFC 826

Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware D. Plummer

RFC 854

Telnet Protocol Specification J. Postel, J. Reynolds

RFC 855

Telnet Option Specification J. Postel, J. Reynolds

RFC 856

Telnet Binary Transmission J. Postel, J. Reynolds

RFC 857

Telnet Echo Option J. Postel, J. Reynolds

RFC 858

Telnet Suppress Go Ahead Option J. Postel, J. Reynolds

RFC 859

Telnet Status Option J. Postel, J. Reynolds

RFC 860

Telnet Timing Mark Option J. Postel, J. Reynolds

RFC 861

Telnet Extended Options: List Option J. Postel, J. Reynolds

RFC 862

Echo Protocol J. Postel

RFC 863

Discard Protocol J. Postel

RFC 864

Character Generator Protocol J. Postel

RFC 865

Quote of the Day Protocol J. Postel

RFC 868

Time Protocol J. Postel, K. Harrenstien

RFC 877

Standard for the transmission of IP datagrams over public data networks J.T. Korb

RFC 883

Domain names: Implementation specification P.V. Mockapetris

RFC 884

Telnet terminal type option M. Solomon, E. Wimmers

- RFC 885**
Telnet end of record option J. Postel
- RFC 894**
Standard for the transmission of IP datagrams over Ethernet networks C. Hornig
- RFC 896**
Congestion control in IP/TCP internetworks J. Nagle
- RFC 903**
Reverse Address Resolution Protocol R. Finlayson, T. Mann, J. Mogul, M. Theimer
- RFC 904**
Exterior Gateway Protocol formal specification D. Mills
- RFC 919**
Broadcasting Internet Datagrams J. Mogul
- RFC 922**
Broadcasting Internet datagrams in the presence of subnets J. Mogul
- RFC 927**
TACACS user identification Telnet option B.A. Anderson
- RFC 933**
Output marking Telnet option S. Silverman
- RFC 946**
Telnet terminal location number option R. Nedved
- RFC 950**
Internet Standard Subnetting Procedure J. Mogul, J. Postel
- RFC 952**
DoD Internet host table specification K. Harrenstien, M. Stahl, E. Feinler
- RFC 959**
File Transfer Protocol J. Postel, J.K. Reynolds
- RFC 961**
Official ARPA-Internet protocols J.K. Reynolds, J. Postel
- RFC 974**
Mail routing and the domain system C. Partridge
- RFC 1001**
Protocol standard for a NetBIOS service on a TCP/UDP transport: Concepts and methods NetBios Working Group in the Defense Advanced Research Projects Agency, Internet Activities Board, End-to-End Services Task Force
- RFC 1002**
Protocol Standard for a NetBIOS service on a TCP/UDP transport: Detailed specifications NetBios Working Group in the Defense Advanced Research Projects Agency, Internet Activities Board, End-to-End Services Task Force
- RFC 1006**
ISO transport services on top of the TCP: Version 3 M.T. Rose, D.E. Cass
- RFC 1009**
Requirements for Internet gateways R. Braden, J. Postel
- RFC 1011**
Official Internet protocols J. Reynolds, J. Postel
- RFC 1013**
X Window System Protocol, version 11: Alpha update April 1987 R. Scheifler
- RFC 1014**
XDR: External Data Representation standard Sun Microsystems
- RFC 1027**
Using ARP to implement transparent subnet gateways S. Carl-Mitchell, J. Quarterman

- RFC 1032**
Domain administrators guide M. Stahl
- RFC 1033**
Domain administrators operations guide M. Lottor
- RFC 1034**
Domain names—concepts and facilities P.V. Mockapetris
- RFC 1035**
Domain names—implementation and specification P.V. Mockapetris
- RFC 1038**
Draft revised IP security option M. St. Johns
- RFC 1041**
Telnet 3270 regime option Y. Rekhter
- RFC 1042**
Standard for the transmission of IP datagrams over IEEE 802 networks J. Postel, J. Reynolds
- RFC 1043**
Telnet Data Entry Terminal option: DODIIS implementation A. Yasuda, T. Thompson
- RFC 1044**
Internet Protocol on Network System's HYPERchannel: Protocol specification K. Hardwick, J. Lekashman
- RFC 1053**
Telnet X.3 PAD option S. Levy, T. Jacobson
- RFC 1055**
Nonstandard for transmission of IP datagrams over serial lines: SLIP J. Romkey
- RFC 1057**
RPC: Remote Procedure Call Protocol Specification: Version 2 Sun Microsystems
- RFC 1058**
Routing Information Protocol C. Hedrick
- RFC 1060**
Assigned numbers J. Reynolds, J. Postel
- RFC 1067**
Simple Network Management Protocol J.D. Case, M. Fedor, M.L. Schoffstall, J. Davin
- RFC 1071**
Computing the Internet checksum R.T. Braden, D.A. Borman, C. Partridge
- RFC 1072**
TCP extensions for long-delay paths V. Jacobson, R.T. Braden
- RFC 1073**
Telnet window size option D. Waitzman
- RFC 1079**
Telnet terminal speed option C. Hedrick
- RFC 1085**
ISO presentation services on top of TCP/IP based internets M.T. Rose
- RFC 1091**
Telnet terminal-type option J. VanBokkelen
- RFC 1094**
NFS: Network File System Protocol specification Sun Microsystems
- RFC 1096**
Telnet X display location option G. Marcy
- RFC 1101**
DNS encoding of network names and other types P. Mockapetris

- RFC 1112**
Host extensions for IP multicasting S.E. Deering
- RFC 1113**
Privacy enhancement for Internet electronic mail: Part I — message encipherment and authentication procedures J. Linn
- RFC 1118**
Hitchhikers Guide to the Internet E. Krol
- RFC 1122**
Requirements for Internet Hosts—Communication Layers R. Braden, Ed.
- RFC 1123**
Requirements for Internet Hosts—Application and Support R. Braden, Ed.
- RFC 1146**
TCP alternate checksum options J. Zweig, C. Partridge
- RFC 1155**
Structure and identification of management information for TCP/IP-based internets M. Rose, K. McCloghrie
- RFC 1156**
Management Information Base for network management of TCP/IP-based internets K. McCloghrie, M. Rose
- RFC 1157**
Simple Network Management Protocol (SNMP) J. Case, M. Fedor, M. Schoffstall, J. Davin
- RFC 1158**
Management Information Base for network management of TCP/IP-based internets: MIB-II M. Rose
- RFC 1166**
Internet numbers S. Kirkpatrick, M.K. Stahl, M. Recker
- RFC 1179**
Line printer daemon protocol L. McLaughlin
- RFC 1180**
TCP/IP tutorial T. Socolofsky, C. Kale
- RFC 1183**
New DNS RR Definitions C.F. Everhart, L.A. Mamakos, R. Ullmann, P.V. Mockapetris
- RFC 1184**
Telnet Linemode Option D. Borman
- RFC 1186**
MD4 Message Digest Algorithm R.L. Rivest
- RFC 1187**
Bulk Table Retrieval with the SNMP M. Rose, K. McCloghrie, J. Davin
- RFC 1188**
Proposed Standard for the Transmission of IP Datagrams over FDDI Networks D. Katz
- RFC 1190**
Experimental Internet Stream Protocol: Version 2 (ST-II) C. Topolcic
- RFC 1191**
Path MTU discovery J. Mogul, S. Deering
- RFC 1198**
FYI on the X window system R. Scheifler
- RFC 1207**
FYI on Questions and Answers: Answers to commonly asked “experienced Internet user” questions G. Malkin, A. Marine, J. Reynolds
- RFC 1208**
Glossary of networking terms O. Jacobsen, D. Lynch

RFC 1213

Management Information Base for Network Management of TCP/IP-based internets: MIB-II K. McCloghrie, M.T. Rose

RFC 1215

Convention for defining traps for use with the SNMP M. Rose

RFC 1227

SNMP MUX protocol and MIB M.T. Rose

RFC 1228

SNMP-DPI: Simple Network Management Protocol Distributed Program Interface G. Carpenter, B. Wijnen

RFC 1229

Extensions to the generic-interface MIB K. McCloghrie

RFC 1230

IEEE 802.4 Token Bus MIB K. McCloghrie, R. Fox

RFC 1231

IEEE 802.5 Token Ring MIB K. McCloghrie, R. Fox, E. Decker

RFC 1236

IP to X.121 address mapping for DDN L. Morales, P. Hasse

RFC 1256

ICMP Router Discovery Messages S. Deering, Ed.

RFC 1267

Border Gateway Protocol 3 (BGP-3) K. Lougheed, Y. Rekhter

RFC 1268

Application of the Border Gateway Protocol in the Internet Y. Rekhter, P. Gross

RFC 1269

Definitions of Managed Objects for the Border Gateway Protocol: Version 3 S. Willis, J. Burruss

RFC 1270

SNMP Communications Services F. Kastenholz, ed.

RFC 1285

FDDI Management Information Base J. Case

RFC 1315

Management Information Base for Frame Relay DTEs C. Brown, F. Baker, C. Carvalho

RFC 1321

The MD5 Message-Digest Algorithm R. Rivest

RFC 1323

TCP Extensions for High Performance V. Jacobson, R. Braden, D. Borman

RFC 1325

FYI on Questions and Answers: Answers to Commonly Asked "New Internet User" Questions G. Malkin, A. Marine

RFC 1327

Mapping between X.400 (1988)/ISO 10021 and RFC 822 S. Hardcastle-Kille

RFC 1340

Assigned Numbers J. Reynolds, J. Postel

RFC 1344

Implications of MIME for Internet Mail Gateways N. Bornstein

RFC 1349

Type of Service in the Internet Protocol Suite P. Almquist

RFC 1351

SNMP Administrative Model J. Davin, J. Galvin, K. McCloghrie

- RFC 1352**
SNMP Security Protocols J. Galvin, K. McCloghrie, J. Davin
- RFC 1353**
Definitions of Managed Objects for Administration of SNMP Parties K. McCloghrie, J. Davin, J. Galvin
- RFC 1354**
IP Forwarding Table MIB F. Baker
- RFC 1356**
Multiprotocol Interconnect[®] on X.25 and ISDN in the Packet Mode A. Malis, D. Robinson, R. Ullmann
- RFC 1358**
Charter of the Internet Architecture Board (IAB) L. Chapin
- RFC 1363**
A Proposed Flow Specification C. Partridge
- RFC 1368**
Definition of Managed Objects for IEEE 802.3 Repeater Devices D. McMaster, K. McCloghrie
- RFC 1372**
Telnet Remote Flow Control Option C. L. Hedrick, D. Borman
- RFC 1374**
IP and ARP on HIPPI J. Renwick, A. Nicholson
- RFC 1381**
SNMP MIB Extension for X.25 LAPB D. Throop, F. Baker
- RFC 1382**
SNMP MIB Extension for the X.25 Packet Layer D. Throop
- RFC 1387**
RIP Version 2 Protocol Analysis G. Malkin
- RFC 1388**
RIP Version 2 Carrying Additional Information G. Malkin
- RFC 1389**
RIP Version 2 MIB Extensions G. Malkin, F. Baker
- RFC 1390**
Transmission of IP and ARP over FDDI Networks D. Katz
- RFC 1393**
Traceroute Using an IP Option G. Malkin
- RFC 1398**
Definitions of Managed Objects for the Ethernet-Like Interface Types F. Kastenholtz
- RFC 1408**
Telnet Environment Option D. Borman, Ed.
- RFC 1413**
Identification Protocol M. St. Johns
- RFC 1416**
Telnet Authentication Option D. Borman, ed.
- RFC 1420**
SNMP over IPX S. Bostock
- RFC 1428**
Transition of Internet Mail from Just-Send-8 to 8bit-SMTP/MIME G. Vaudreuil
- RFC 1442**
Structure of Management Information for version 2 of the Simple Network Management Protocol (SNMPv2) J. Case, K. McCloghrie, M. Rose, S. Waldbusser
- RFC 1443**
Textual Conventions for version 2 of the Simple Network Management Protocol (SNMPv2) J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 1445

Administrative Model for version 2 of the Simple Network Management Protocol (SNMPv2) J. Galvin, K. McCloghrie

RFC 1447

Party MIB for version 2 of the Simple Network Management Protocol (SNMPv2) K. McCloghrie, J. Galvin

RFC 1448

Protocol Operations for version 2 of the Simple Network Management Protocol (SNMPv2) J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 1464

Using the Domain Name System to Store Arbitrary String Attributes R. Rosenbaum

RFC 1469

IP Multicast over Token-Ring Local Area Networks T. Pusateri

RFC 1483

Multiprotocol Encapsulation over ATM Adaptation Layer 5 Juha Heinanen

RFC 1514

Host Resources MIB P. Grillo, S. Waldbusser

RFC 1516

Definitions of Managed Objects for IEEE 802.3 Repeater Devices D. McMaster, K. McCloghrie

RFC 1521

MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies N. Borenstein, N. Freed

RFC 1535

A Security Problem and Proposed Correction With Widely Deployed DNS Software E. Gavron

RFC 1536

Common DNS Implementation Errors and Suggested Fixes A. Kumar, J. Postel, C. Neuman, P. Danzig, S. Miller

RFC 1537

Common DNS Data File Configuration Errors P. Beertema

RFC 1540

Internet Official Protocol Standards J. Postel

RFC 1571

Telnet Environment Option Interoperability Issues D. Borman

RFC 1572

Telnet Environment Option S. Alexander

RFC 1573

Evolution of the Interfaces Group of MIB-II K. McCloghrie, F. Kastenholz

RFC 1577

Classical IP and ARP over ATM M. Laubach

RFC 1583

OSPF Version 2 J. Moy

RFC 1591

Domain Name System Structure and Delegation J. Postel

RFC 1592

Simple Network Management Protocol Distributed Protocol Interface Version 2.0 B. Wijnen, G. Carpenter, K. Curran, A. Sehgal, G. Waters

RFC 1594

FYI on Questions and Answers—Answers to Commonly Asked "New Internet User" Questions A. Marine, J. Reynolds, G. Malkin

RFC 1644

T/TCP — TCP Extensions for Transactions Functional Specification R. Braden

- RFC 1646**
TN3270 Extensions for LUsername and Printer Selection C. Graves, T. Butts, M. Angel
- RFC 1647**
TN3270 Enhancements B. Kelly
- RFC 1652**
SMTP Service Extension for 8bit-MIMEtransport J. Klensin, N. Freed, M. Rose, E. Stefferud, D. Crocker
- RFC 1664**
Using the Internet DNS to Distribute RFC1327 Mail Address Mapping Tables C. Allochio, A. Bonito, B. Cole, S. Giordano, R. Hagens
- RFC 1693**
An Extension to TCP: Partial Order Service T. Connolly, P. Amer, P. Conrad
- RFC 1695**
Definitions of Managed Objects for ATM Management Version 8.0 using SMIPv2 M. Ahmed, K. Tesink
- RFC 1701**
Generic Routing Encapsulation (GRE) S. Hanks, T. Li, D. Farinacci, P. Traina
- RFC 1702**
Generic Routing Encapsulation over IPv4 networks S. Hanks, T. Li, D. Farinacci, P. Traina
- RFC 1706**
DNS NSAP Resource Records B. Manning, R. Colella
- RFC 1712**
DNS Encoding of Geographical Location C. Farrell, M. Schulze, S. Pleitner D. Baldoni
- RFC 1713**
Tools for DNS debugging A. Romao
- RFC 1723**
RIP Version 2—Carrying Additional Information G. Malkin
- RFC 1752**
The Recommendation for the IP Next Generation Protocol S. Bradner, A. Mankin
- RFC 1766**
Tags for the Identification of Languages H. Alvestrand
- RFC 1771**
A Border Gateway Protocol 4 (BGP-4) Y. Rekhter, T. Li
- RFC 1794**
DNS Support for Load Balancing T. Brisco
- RFC 1819**
Internet Stream Protocol Version 2 (ST2) Protocol Specification—Version ST2+ L. Delgrossi, L. Berger Eds.
- RFC 1826**
IP Authentication Header R. Atkinson
- RFC 1828**
IP Authentication using Keyed MD5 P. Metzger, W. Simpson
- RFC 1829**
The ESP DES-CBC Transform P. Karn, P. Metzger, W. Simpson
- RFC 1830**
SMTP Service Extensions for Transmission of Large and Binary MIME Messages G. Vaudreuil
- RFC 1831**
RPC: Remote Procedure Call Protocol Specification Version 2 R. Srinivasan
- RFC 1832**
XDR: External Data Representation Standard R. Srinivasan
- RFC 1833**
Binding Protocols for ONC RPC Version 2 R. Srinivasan

RFC 1850

OSPF Version 2 Management Information Base F. Baker, R. Coltun

RFC 1854

SMTP Service Extension for Command Pipelining N. Freed

RFC 1869

SMTP Service Extensions J. Klensin, N. Freed, M. Rose, E. Stefferud, D. Crocker

RFC 1870

SMTP Service Extension for Message Size Declaration J. Klensin, N. Freed, K. Moore

RFC 1876

A Means for Expressing Location Information in the Domain Name System C. Davis, P. Vixie, T. Goodwin, I. Dickinson

RFC 1883

Internet Protocol, Version 6 (IPv6) Specification S. Deering, R. Hinden

RFC 1884

IP Version 6 Addressing Architecture R. Hinden, S. Deering, Eds.

RFC 1886

DNS Extensions to support IP version 6 S. Thomson, C. Huitema

RFC 1888

OSI NSAPs and IPv6 J. Bound, B. Carpenter, D. Harrington, J. Houldsworth, A. Lloyd

RFC 1891

SMTP Service Extension for Delivery Status Notifications K. Moore

RFC 1892

The Multipart/Report Content Type for the Reporting of Mail System Administrative Messages G. Vaudreuil

RFC 1894

An Extensible Message Format for Delivery Status Notifications K. Moore, G. Vaudreuil

RFC 1901

Introduction to Community-based SNMPv2 J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 1902

Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMPv2) J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 1903

Textual Conventions for Version 2 of the Simple Network Management Protocol (SNMPv2) J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 1904

Conformance Statements for Version 2 of the Simple Network Management Protocol (SNMPv2) J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 1905

Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2) J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 1906

Transport Mappings for Version 2 of the Simple Network Management Protocol (SNMPv2) J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 1907

Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2) J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 1908

Coexistence between Version 1 and Version 2 of the Internet-standard Network Management Framework J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 1912

Common DNS Operational and Configuration Errors D. Barr

- RFC 1918**
Address Allocation for Private Internets Y. Rekhter, B. Moskowitz, D. Karrenberg, G.J. de Groot, E. Lear
- RFC 1928**
SOCKS Protocol Version 5 M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, L. Jones
- RFC 1930**
Guidelines for creation, selection, and registration of an Autonomous System (AS) J. Hawkinson, T. Bates
- RFC 1939**
Post Office Protocol-Version 3 J. Myers, M. Rose
- RFC 1981**
Path MTU Discovery for IP version 6 J. McCann, S. Deering, J. Mogul
- RFC 1982**
Serial Number Arithmetic R. Elz, R. Bush
- RFC 1985**
SMTP Service Extension for Remote Message Queue Starting J. De Winter
- RFC 1995**
Incremental Zone Transfer in DNS M. Ohta
- RFC 1996**
A Mechanism for Prompt Notification of Zone Changes (DNS NOTIFY) P. Vixie
- RFC 2010**
Operational Criteria for Root Name Servers B. Manning, P. Vixie
- RFC 2011**
SNMPv2 Management Information Base for the Internet Protocol using SMIPv2 K. McCloghrie, Ed.
- RFC 2012**
SNMPv2 Management Information Base for the Transmission Control Protocol using SMIPv2 K. McCloghrie, Ed.
- RFC 2013**
SNMPv2 Management Information Base for the User Datagram Protocol using SMIPv2 K. McCloghrie, Ed.
- RFC 2018**
TCP Selective Acknowledgement Options M. Mathis, J. Mahdavi, S. Floyd, A. Romanow
- RFC 2026**
The Internet Standards Process — Revision 3 S. Bradner
- RFC 2030**
Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI D. Mills
- RFC 2033**
Local Mail Transfer Protocol J. Myers
- RFC 2034**
SMTP Service Extension for Returning Enhanced Error Codes N. Freed
- RFC 2040**
The RC5, RC5-CBC, RC-5-CBC-Pad, and RC5-CTS Algorithms R. Baldwin, R. Rivest
- RFC 2045**
Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies N. Freed, N. Borenstein
- RFC 2052**
A DNS RR for specifying the location of services (DNS SRV) A. Gulbrandsen, P. Vixie
- RFC 2065**
Domain Name System Security Extensions D. Eastlake 3rd, C. Kaufman
- RFC 2066**
TELNET CHARSET Option R. Gellens

RFC 2080

RIPng for IPv6 G. Malkin, R. Minnear

RFC 2096

IP Forwarding Table MIB F. Baker

RFC 2104

HMAC: Keyed-Hashing for Message Authentication H. Krawczyk, M. Bellare, R. Canetti

RFC 2119

Keywords for use in RFCs to Indicate Requirement Levels S. Bradner

RFC 2133

Basic Socket Interface Extensions for IPv6 R. Gilligan, S. Thomson, J. Bound, W. Stevens

RFC 2136

Dynamic Updates in the Domain Name System (DNS UPDATE) P. Vixie, Ed., S. Thomson, Y. Rekhter, J. Bound

RFC 2137

Secure Domain Name System Dynamic Update D. Eastlake 3rd

RFC 2163

Using the Internet DNS to Distribute MIXER Conformant Global Address Mapping (MCGAM) C. Allocchio

RFC 2168

Resolution of Uniform Resource Identifiers using the Domain Name System R. Daniel, M. Mealling

RFC 2178

OSPF Version 2 J. Moy

RFC 2181

Clarifications to the DNS Specification R. Elz, R. Bush

RFC 2205

Resource ReSerVation Protocol (RSVP)—Version 1 Functional Specification R. Braden, Ed., L. Zhang, S. Berson, S. Herzog, S. Jamin

RFC 2210

The Use of RSVP with IETF Integrated Services J. Wroclawski

RFC 2211

Specification of the Controlled-Load Network Element Service J. Wroclawski

RFC 2212

Specification of Guaranteed Quality of Service S. Shenker, C. Partridge, R. Guerin

RFC 2215

General Characterization Parameters for Integrated Service Network Elements S. Shenker, J. Wroclawski

RFC 2217

Telnet Com Port Control Option G. Clarke

RFC 2219

Use of DNS Aliases for Network Services M. Hamilton, R. Wright

RFC 2228

FTP Security Extensions M. Horowitz, S. Lunt

RFC 2230

Key Exchange Delegation Record for the DNS R. Atkinson

RFC 2233

The Interfaces Group MIB using SMIV2 K. McCloghrie, F. Kastenholz

RFC 2240

A Legal Basis for Domain Name Allocation O. Vaughn

RFC 2246

The TLS Protocol Version 1.0 T. Dierks, C. Allen

RFC 2251

Lightweight Directory Access Protocol (v3) M. Wahl, T. Howes, S. Kille

RFC 2253

Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names M. Wahl, S. Kille, T. Howes

RFC 2254

The String Representation of LDAP Search Filters T. Howes

RFC 2261

An Architecture for Describing SNMP Management Frameworks D. Harrington, R. Presuhn, B. Wijnen

RFC 2262

Message Processing and Dispatching for the Simple Network Management Protocol (SNMP) J. Case, D. Harrington, R. Presuhn, B. Wijnen

RFC 2271

An Architecture for Describing SNMP Management Frameworks D. Harrington, R. Presuhn, B. Wijnen

RFC 2273

SNMPv3 Applications D. Levi, P. Meyer, B. Stewartz

RFC 2274

User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3) U. Blumenthal, B. Wijnen

RFC 2275

View-based Access Control Model (VACM) for the Simple Network Management Protocol (SNMP) B. Wijnen, R. Presuhn, K. McCloghrie

RFC 2279

UTF-8, a transformation format of ISO 10646 F. Yergeau

RFC 2292

Advanced Sockets API for IPv6 W. Stevens, M. Thomas

RFC 2308

Negative Caching of DNS Queries (DNS NCACHE) M. Andrews

RFC 2317

Classless IN-ADDR.ARPA delegation H. Eidnes, G. de Groot, P. Vixie

RFC 2320

Definitions of Managed Objects for Classical IP and ARP Over ATM Using SMIPv2 (IPOA-MIB) M. Greene, J. Luciani, K. White, T. Kuo

RFC 2328

OSPF Version 2 J. Moy

RFC 2345

Domain Names and Company Name Retrieval J. Klensin, T. Wolf, G. Oglesby

RFC 2352

A Convention for Using Legal Names as Domain Names O. Vaughn

RFC 2355

TN3270 Enhancements B. Kelly

RFC 2358

Definitions of Managed Objects for the Ethernet-like Interface Types J. Flick, J. Johnson

RFC 2373

IP Version 6 Addressing Architecture R. Hinden, S. Deering

RFC 2374

An IPv6 Aggregatable Global Unicast Address Format R. Hinden, M. O'Dell, S. Deering

RFC 2375

IPv6 Multicast Address Assignments R. Hinden, S. Deering

RFC 2385

Protection of BGP Sessions via the TCP MD5 Signature Option A. Hefferman

RFC 2389

Feature negotiation mechanism for the File Transfer Protocol P. Hethmon, R. Elz

RFC 2401

Security Architecture for Internet Protocol S. Kent, R. Atkinson

RFC 2402

IP Authentication Header S. Kent, R. Atkinson

RFC 2403

The Use of HMAC-MD5-96 within ESP and AH C. Madson, R. Glenn

RFC 2404

The Use of HMAC-SHA-1-96 within ESP and AH C. Madson, R. Glenn

RFC 2405

The ESP DES-CBC Cipher Algorithm With Explicit IV C. Madson, N. Doraswamy

RFC 2406

IP Encapsulating Security Payload (ESP) S. Kent, R. Atkinson

RFC 2407

The Internet IP Security Domain of Interpretation for ISAKMPD Piper

RFC 2408

Internet Security Association and Key Management Protocol (ISAKMP) D. Maughan, M. Schertler, M. Schneider, J. Turner

RFC 2409

The Internet Key Exchange (IKE) D. Harkins, D. Carrel

RFC 2410

The NULL Encryption Algorithm and Its Use With IPsec R. Glenn, S. Kent,

RFC 2428

FTP Extensions for IPv6 and NATs M. Allman, S. Ostermann, C. Metz

RFC 2445

Internet Calendaring and Scheduling Core Object Specification (iCalendar) F. Dawson, D. Stenerson

RFC 2459

Internet X.509 Public Key Infrastructure Certificate and CRL Profile R. Housley, W. Ford, W. Polk, D. Solo

RFC 2460

Internet Protocol, Version 6 (IPv6) Specification S. Deering, R. Hinden

RFC 2461

Neighbor Discovery for IP Version 6 (IPv6) T. Narten, E. Nordmark, W. Simpson

RFC 2462

IPv6 Stateless Address Autoconfiguration S. Thomson, T. Narten

RFC 2463

Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification A. Conta, S. Deering

RFC 2464

Transmission of IPv6 Packets over Ethernet Networks M. Crawford

RFC 2466

Management Information Base for IP Version 6: ICMPv6 Group D. Haskin, S. Onishi

RFC 2476

Message Submission R. Gellens, J. Klensin

RFC 2487

SMTP Service Extension for Secure SMTP over TLS P. Hoffman

RFC 2505

Anti-Spam Recommendations for SMTP MTAs G. Lindberg

RFC 2523

Photuris: Extended Schemes and Attributes P. Karn, W. Simpson

RFC 2535

Domain Name System Security Extensions D. Eastlake 3rd

RFC 2538

Storing Certificates in the Domain Name System (DNS) D. Eastlake 3rd, O. Gudmundsson

RFC 2539

Storage of Diffie-Hellman Keys in the Domain Name System (DNS) D. Eastlake 3rd

RFC 2540

Detached Domain Name System (DNS) Information D. Eastlake 3rd

RFC 2554

SMTP Service Extension for Authentication J. Myers

RFC 2570

Introduction to Version 3 of the Internet-standard Network Management Framework J. Case, R. Mundy, D. Partain, B. Stewart

RFC 2571

An Architecture for Describing SNMP Management Frameworks B. Wijnen, D. Harrington, R. Presuhn

RFC 2572

Message Processing and Dispatching for the Simple Network Management Protocol (SNMP) J. Case, D. Harrington, R. Presuhn, B. Wijnen

RFC 2573

SNMP Applications D. Levi, P. Meyer, B. Stewart

RFC 2574

User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3) U. Blumenthal, B. Wijnen

RFC 2575

View-based Access Control Model (VACM) for the Simple Network Management Protocol (SNMP) B. Wijnen, R. Presuhn, K. McCloghrie

RFC 2576

Co-Existence between Version 1, Version 2, and Version 3 of the Internet-standard Network Management Framework R. Frye, D. Levi, S. Routhier, B. Wijnen

RFC 2578

Structure of Management Information Version 2 (SMIv2) K. McCloghrie, D. Perkins, J. Schoenwaelder

RFC 2579

Textual Conventions for SMIv2 K. McCloghrie, D. Perkins, J. Schoenwaelder

RFC 2580

Conformance Statements for SMIv2 K. McCloghrie, D. Perkins, J. Schoenwaelder

RFC 2581

TCP Congestion Control M. Allman, V. Paxson, W. Stevens

RFC 2583

Guidelines for Next Hop Client (NHC) Developers R. Carlson, L. Winkler

RFC 2591

Definitions of Managed Objects for Scheduling Management Operations D. Levi, J. Schoenwaelder

RFC 2625

IP and ARP over Fibre Channel M. Rajagopal, R. Bhagwat, W. Rickard

RFC 2635

Don't SPEW A Set of Guidelines for Mass Unsolicited Mailings and Postings (spam)* S. Hambridge, A. Lunde

RFC 2637

Point-to-Point Tunneling Protocol K. Hamzeh, G. Pall, W. Verthein, J. Taarud, W. Little, G. Zorn

- RFC 2640**
Internationalization of the File Transfer Protocol B. Curtin
- RFC 2665**
Definitions of Managed Objects for the Ethernet-like Interface Types J. Flick, J. Johnson
- RFC 2671**
Extension Mechanisms for DNS (EDNS0) P. Vixie
- RFC 2672**
Non-Terminal DNS Name Redirection M. Crawford
- RFC 2675**
IPv6 Jumbograms D. Borman, S. Deering, R. Hinden
- RFC 2710**
Multicast Listener Discovery (MLD) for IPv6 S. Deering, W. Fenner, B. Haberman
- RFC 2711**
IPv6 Router Alert Option C. Partridge, A. Jackson
- RFC 2740**
OSPF for IPv6 R. Coltun, D. Ferguson, J. Moy
- RFC 2753**
A Framework for Policy-based Admission Control R. Yavatkar, D. Pendarakis, R. Guerin
- RFC 2782**
A DNS RR for specifying the location of services (DNS SRV) A. Gubrandsen, P. Vixie, L. Esibov
- RFC 2821**
Simple Mail Transfer Protocol J. Klensin, Ed.
- RFC 2822**
Internet Message Format P. Resnick, Ed.
- RFC 2840**
TELNET KERMIT OPTION J. Altman, F. da Cruz
- RFC 2845**
Secret Key Transaction Authentication for DNS (TSIG) P. Vixie, O. Gudmundsson, D. Eastlake 3rd, B. Wellington
- RFC 2851**
Textual Conventions for Internet Network Addresses M. Daniele, B. Haberman, S. Routhier, J. Schoenwaelder
- RFC 2852**
Deliver By SMTP Service Extension D. Newman
- RFC 2874**
DNS Extensions to Support IPv6 Address Aggregation and Renumbering M. Crawford, C. Huitema
- RFC 2915**
The Naming Authority Pointer (NAPTR) DNS Resource Record M. Mealling, R. Daniel
- RFC 2920**
SMTP Service Extension for Command Pipelining N. Freed
- RFC 2930**
Secret Key Establishment for DNS (TKEY RR) D. Eastlake, 3rd
- RFC 2941**
Telnet Authentication Option T. Ts'o, ed., J. Altman
- RFC 2942**
Telnet Authentication: Kerberos Version 5 T. Ts'o
- RFC 2946**
Telnet Data Encryption Option T. Ts'o
- RFC 2952**
Telnet Encryption: DES 64 bit Cipher Feedback T. Ts'o

RFC 2953

Telnet Encryption: DES 64 bit Output Feedback T. Ts'o

RFC 2992

Analysis of an Equal-Cost Multi-Path Algorithm C. Hopps

RFC 3019

IP Version 6 Management Information Base for The Multicast Listener Discovery Protocol B. Haberman, R. Worzella

RFC 3060

Policy Core Information Model—Version 1 Specification B. Moore, E. Ellessen, J. Strassner, A. Westerinen

RFC 3152

Delegation of IP6.ARPA R. Bush

RFC 3164

The BSD Syslog Protocol C. Lonvick

RFC 3207

SMTP Service Extension for Secure SMTP over Transport Layer Security P. Hoffman

RFC 3226

DNSSEC and IPv6 A6 aware server/resolver message size requirements O. Gudmundsson

RFC 3291

Textual Conventions for Internet Network Addresses M. Daniele, B. Haberman, S. Routhier, J. Schoenwaelder

RFC 3363

Representing Internet Protocol version 6 (IPv6) Addresses in the Domain Name System R. Bush, A. Durand, B. Fink, O. Gudmundsson, T. Hain

RFC 3376

Internet Group Management Protocol, Version 3 B. Cain, S. Deering, I. Kouvelas, B. Fenner, A. Thyagarajan

RFC 3390

Increasing TCP's Initial Window M. Allman, S. Floyd, C. Partridge

RFC 3410

Introduction and Applicability Statements for Internet-Standard Management Framework J. Case, R. Mundy, D. Partain, B. Stewart

RFC 3411

An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks D. Harrington, R. Presuhn, B. Wijnen

RFC 3412

Message Processing and Dispatching for the Simple Network Management Protocol (SNMP) J. Case, D. Harrington, R. Presuhn, B. Wijnen

RFC 3413

Simple Network Management Protocol (SNMP) Applications D. Levi, P. Meyer, B. Stewart

RFC 3414

User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3) U. Blumenthal, B. Wijnen

RFC 3415

View-based Access Control Model (VACM) for the Simple Network Management Protocol (SNMP) B. Wijnen, R. Presuhn, K. McCloghrie

RFC 3416

Version 2 of the Protocol Operations for the Simple Network Management Protocol (SNMP) R. Presuhn, J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 3417

Transport Mappings for the Simple Network Management Protocol (SNMP) R. Presuhn, J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 3418

Management Information Base (MIB) for the Simple Network Management Protocol (SNMP) R. Presuhn, J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 3419

Textual Conventions for Transport Addresses M. Daniele, J. Schoenwaelder

RFC 3484

Default Address Selection for Internet Protocol version 6 (IPv6) R. Draves

RFC 3493

Basic Socket Interface Extensions for IPv6 R. Gilligan, S. Thomson, J. Bound, J. McCann, W. Stevens

RFC 3513

Internet Protocol Version 6 (IPv6) Addressing Architecture R. Hinden, S. Deering

RFC 3526

More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE) T. Kivinen, M. Kojo

RFC 3542

Advanced Sockets Application Programming Interface (API) for IPv6 W. Richard Stevens, M. Thomas, E. Nordmark, T. Jinmei

RFC 3566

The AES-XCBC-MAC-96 Algorithm and Its Use With IPsec S. Frankel, H. Herbert

RFC 3569

An Overview of Source-Specific Multicast (SSM) S. Bhattacharyya, Ed.

RFC 3584

Coexistence between Version 1, Version 2, and Version 3 of the Internet-standard Network Management Framework R. Frye, D. Levi, S. Routhier, B. Wijnen

RFC 3602

The AES-CBC Cipher Algorithm and Its Use with IPsec S. Frankel, R. Glenn, S. Kelly

RFC 3629

UTF-8, a transformation format of ISO 10646 R. Kermode, C. Vicisano

RFC 3658

Delegation Signer (DS) Resource Record (RR) O. Gudmundsson

RFC 3678

Socket Interface Extensions for Multicast Source Filters D. Thaler, B. Fenner, B. Quinn

RFC 3715

IPsec-Network Address Translation (NAT) Compatibility Requirements B. Aboba, W. Dixon

RFC 3810

Multicast Listener Discovery Version 2 (MLDv2) for IPv6 R. Vida, Ed., L. Costa, Ed.

RFC 3826

The Advanced Encryption Standard (AES) Cipher Algorithm in the SNMP User-based Security Model U. Blumenthal, F. Maino, K. McCloghrie.

RFC 3947

Negotiation of NAT-Traversal in the IKE T. Kivinen, B. Swander, A. Huttunen, V. Volpe

RFC 3948

UDP Encapsulation of IPsec ESP Packets A. Huttunen, B. Swander, V. Volpe, L. DiBurro, M. Stenberg

RFC 4001

Textual Conventions for Internet Network Addresses M. Daniele, B. Haberman, S. Routhier, J. Schoenwaelder

RFC 4007

IPv6 Scoped Address Architecture S. Deering, B. Haberman, T. Jinmei, E. Nordmark, B. Zill

- RFC 4022**
Management Information Base for the Transmission Control Protocol (TCP) R. Raghunarayan
- RFC 4106**
The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP) J. Viega, D. McGrew
- RFC 4109**
Algorithms for Internet Key Exchange version 1 (IKEv1) P. Hoffman
- RFC 4113**
Management Information Base for the User Datagram Protocol (UDP) B. Fenner, J. Flick
- RFC 4191**
Default Router Preferences and More-Specific Routes R. Draves, D. Thaler
- RFC 4217**
Securing FTP with TLS P. Ford-Hutchinson
- RFC 4292**
IP Forwarding Table MIB B. Haberman
- RFC 4293**
Management Information Base for the Internet Protocol (IP) S. Routhier
- RFC 4301**
Security Architecture for the Internet Protocol S. Kent, K. Seo
- RFC 4302**
IP Authentication Header S. Kent
- RFC 4303**
IP Encapsulating Security Payload (ESP) S. Kent
- RFC 4304**
Extended Sequence Number (ESN) Addendum to IPsec Domain of Interpretation (DOI) for Internet Security Association and Key Management Protocol (ISAKMP) S. Kent
- RFC 4307**
Cryptographic Algorithms for Use in the Internet Key Exchange Version 2 (IKEv2) J. Schiller
- RFC 4308**
Cryptographic Suites for IPsec P. Hoffman
- RFC 4434**
The AES-XCBC-PRF-128 Algorithm for the Internet Key Exchange Protocol P. Hoffman
- RFC 4443**
Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification A. Conta, S. Deering
- RFC 4552**
Authentication/Confidentiality for OSPFv3 M. Gupta, N. Melam
- RFC 4678**
Server/Application State Protocol v1 A. Bivens
- RFC 4753**
ECP Groups for IKE and IKEv2 D. Fu, J. Solinas
- RFC 4754**
IKE and IKEv2 Authentication Using the Elliptic Curve Digital Signature Algorithm (ECDSA) D. Fu, J. Solinas
- RFC 4809**
Requirements for an IPsec Certificate Management Profile C. Bonatti, Ed., S. Turner, Ed., G. Lebovitz, Ed.
- RFC 4835**
Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH) V. Manral

RFC 4862

IPv6 Stateless Address Autoconfiguration S. Thomson, T. Narten, T. Jinmei

RFC 4868

Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec S. Kelly, S. Frankel

RFC 4869

Suite B Cryptographic Suites for IPsec L. Law, J. Solinas

RFC 4941

Privacy Extensions for Stateless Address Autoconfiguration in IPv6 T. Narten, R. Draves, S. Krishnan

RFC 4945

The Internet IP Security PKI Profile of IKEv1/ISAKMP, IKEv2, and PKIX B. Korver

RFC 5014

IPv6 Socket API for Source Address Selection E. Nordmark, S. Chakrabarti, J. Laganier

RFC 5095

Deprecation of Type 0 Routing Headers in IPv6 J. Abley, P. Savola, G. Neville-Neil

RFC 5175

IPv6 Router Advertisement Flags Option B. Haberman, Ed., R. Hinden

RFC 5282

Using Authenticated Encryption Algorithms with the Encrypted Payload of the Internet Key Exchange version 2 (IKEv2) Protocol D. Black, D. McGrew

RFC 5996

Internet Key Exchange Protocol Version 2 (IKEv2) C. Kaufman, P. Hoffman, Y. Nir, P. Eronen

RFC 7627

Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension K. Bhargavan, A. Delignat-Lavaud, A. Pironti, Inria Paris-Rocquencourt, A. Langley, M. Ray

RFC 8446

The Transport Layer Security (TLS) Protocol Version 1.3 E. Rescorla

Internet drafts

Internet drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Other groups can also distribute working documents as Internet drafts. You can see Internet drafts at <http://www.ietf.org/ID.html>.

Appendix H. Accessibility

Accessible publications for this product are offered through [IBM Documentation for z/OS](#).

If you experience difficulty with the accessibility of any z/OS documentation see [How to Send Feedback to IBM](#) to leave documentation feedback.

Notices

This information was developed for products and services that are offered in the USA or elsewhere.

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 United States of America

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

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: 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 states 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.

This information could include missing, incorrect, or broken hyperlinks. Hyperlinks are maintained in only the HTML plug-in output for the IBM Documentation. Use of hyperlinks in other output formats of this information is at your own risk.

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 supply 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 Corporation Site Counsel 2455 South Road Poughkeepsie, NY 12601-5400 USA

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.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

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.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains 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 the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains 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.

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.

Depending upon the configurations deployed, this Software Offering may use session cookies that collect each user's name, email address, phone number, or other personally identifiable information for purposes of enhanced user usability and single sign-on configuration. These cookies can be disabled, but disabling them will also eliminate the functionality they enable.

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 IBM's Privacy Policy at ibm.com/privacy and IBM's Online Privacy Statement at ibm.com/privacy/details in the section entitled "Cookies, Web Beacons and Other Technologies," and the "IBM Software Products and Software-as-a-Service Privacy Statement" at ibm.com/software/info/product-privacy.

Policy for unsupported hardware

Various z/OS elements, such as DFSMS, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: [IBM Lifecycle Support for z/OS \(www.ibm.com/software/support/systemsz/lifecycle\)](http://www.ibm.com/software/support/systemsz/lifecycle)
- For information about currently-supported IBM hardware, contact your IBM representative.

Programming interface information

This publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of z/OS Communications Server.

Policy for unsupported hardware

Various z/OS elements, such as DFSMS, HCD, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at [Copyright and trademark information](http://www.ibm.com/legal/copytrade.shtml) at www.ibm.com/legal/copytrade.shtml.

Bibliography

This bibliography contains descriptions of the documents in the z/OS Communications Server library.

z/OS Communications Server documentation is available online at the z/OS Internet Library web page at <http://www.ibm.com/systems/z/os/zos/library/bkserv/>.

z/OS Communications Server library updates

Updates to documents are also available on RETAIN and in information APARs (info APARs). Go to <https://www.ibm.com/mysupport> to view information APARs.

- [z/OS Communications Server V2R1 New Function APAR Summary](#)
- [z/OS Communications Server V2R2 New Function APAR Summary](#)
- [z/OS Communications Server V2R3 New Function APAR Summary](#)
- [z/OS Communications Server V2R4 New Function APAR Summary](#)

z/OS Communications Server information

z/OS Communications Server product information is grouped by task in the following tables.

Planning

Title	Number	Description
z/OS Communications Server: New Function Summary	GC27-3664	This document is intended to help you plan for new IP or SNA functions, whether you are migrating from a previous version or installing z/OS for the first time. It summarizes what is new in the release and identifies the suggested and required modifications needed to use the enhanced functions.
z/OS Communications Server: IPv6 Network and Appl Design Guide	SC27-3663	This document is a high-level introduction to IPv6. It describes concepts of z/OS Communications Server's support of IPv6, coexistence with IPv4, and migration issues.

Resource definition, configuration, and tuning

Title	Number	Description
z/OS Communications Server: IP Configuration Guide	SC27-3650	This document describes the major concepts involved in understanding and configuring an IP network. Familiarity with the z/OS operating system, IP protocols, z/OS UNIX System Services, and IBM Time Sharing Option (TSO) is recommended. Use this document with the z/OS Communications Server: IP Configuration Reference .

Title	Number	Description
z/OS Communications Server: IP Configuration Reference	SC27-3651	This document presents information for people who want to administer and maintain IP. Use this document with the z/OS Communications Server: IP Configuration Guide . The information in this document includes: <ul style="list-style-type: none"> • TCP/IP configuration data sets • Configuration statements • Translation tables • Protocol number and port assignments
z/OS Communications Server: SNA Network Implementation Guide	SC27-3672	This document presents the major concepts involved in implementing an SNA network. Use this document with the z/OS Communications Server: SNA Resource Definition Reference .
z/OS Communications Server: SNA Resource Definition Reference	SC27-3675	This document describes each SNA definition statement, start option, and macroinstruction for user tables. It also describes NCP definition statements that affect SNA. Use this document with the z/OS Communications Server: SNA Network Implementation Guide .
z/OS Communications Server: SNA Resource Definition Samples	SC27-3676	This document contains sample definitions to help you implement SNA functions in your networks, and includes sample major node definitions.
z/OS Communications Server: IP Network Print Facility	SC27-3658	This document is for systems programmers and network administrators who need to prepare their network to route SNA, JES2, or JES3 printer output to remote printers using TCP/IP Services.

Operation

Title	Number	Description
z/OS Communications Server: IP User's Guide and Commands	SC27-3662	This document describes how to use TCP/IP applications. It contains requests with which a user can log on to a remote host using Telnet, transfer data sets using FTP, send electronic mail, print on remote printers, and authenticate network users.
z/OS Communications Server: IP System Administrator's Commands	SC27-3661	This document describes the functions and commands helpful in configuring or monitoring your system. It contains system administrator's commands, such as TSO NETSTAT, PING, TRACERTE and their UNIX counterparts. It also includes TSO and MVS commands commonly used during the IP configuration process.
z/OS Communications Server: SNA Operation	SC27-3673	This document serves as a reference for programmers and operators requiring detailed information about specific operator commands.
z/OS Communications Server: Quick Reference	SC27-3665	This document contains essential information about SNA and IP commands.

Customization

Title	Number	Description
z/OS Communications Server: SNA Customization	SC27-3666	<p>This document enables you to customize SNA, and includes the following information:</p> <ul style="list-style-type: none"> • Communication network management (CNM) routing table • Logon-interpret routine requirements • Logon manager installation-wide exit routine for the CLU search exit • TSO/SNA installation-wide exit routines • SNA installation-wide exit routines

Writing application programs

Title	Number	Description
z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference	SC27-3660	This document describes the syntax and semantics of program source code necessary to write your own application programming interface (API) into TCP/IP. You can use this interface as the communication base for writing your own client or server application. You can also use this document to adapt your existing applications to communicate with each other using sockets over TCP/IP.
z/OS Communications Server: IP CICS Sockets Guide	SC27-3649	This document is for programmers who want to set up, write application programs for, and diagnose problems with the socket interface for CICS using z/OS TCP/IP.
z/OS Communications Server: IP IMS Sockets Guide	SC27-3653	This document is for programmers who want application programs that use the IMS TCP/IP application development services provided by the TCP/IP Services of IBM.
z/OS Communications Server: IP Programmer's Guide and Reference	SC27-3659	This document describes the syntax and semantics of a set of high-level application functions that you can use to program your own applications in a TCP/IP environment. These functions provide support for application facilities, such as user authentication, distributed databases, distributed processing, network management, and device sharing. Familiarity with the z/OS operating system, TCP/IP protocols, and IBM Time Sharing Option (TSO) is recommended.
z/OS Communications Server: SNA Programming	SC27-3674	This document describes how to use SNA macroinstructions to send data to and receive data from (1) a terminal in either the same or a different domain, or (2) another application program in either the same or a different domain.
z/OS Communications Server: SNA Programmer's LU 6.2 Guide	SC27-3669	This document describes how to use the SNA LU 6.2 application programming interface for host application programs. This document applies to programs that use only LU 6.2 sessions or that use LU 6.2 sessions along with other session types. (Only LU 6.2 sessions are covered in this document.)
z/OS Communications Server: SNA Programmer's LU 6.2 Reference	SC27-3670	This document provides reference material for the SNA LU 6.2 programming interface for host application programs.

Title	Number	Description
z/OS Communications Server: CSM Guide	SC27-3647	This document describes how applications use the communications storage manager.

Diagnosis

Title	Number	Description
z/OS Communications Server: IP Diagnosis Guide	GC27-3652	This document explains how to diagnose TCP/IP problems and how to determine whether a specific problem is in the TCP/IP product code. It explains how to gather information for and describe problems to the IBM Software Support Center.
z/OS Communications Server: ACF/TAP Trace Analysis Handbook	GC27-3645	This document explains how to gather the trace data that is collected and stored in the host processor. It also explains how to use the Advanced Communications Function/Trace Analysis Program (ACF/TAP) service aid to produce reports for analyzing the trace data information.
z/OS Communications Server: SNA Diagnosis Vol 1, Techniques and Procedures and z/OS Communications Server: SNA Diagnosis Vol 2, FFST Dumps and the VIT	GC27-3667 GC27-3668	These documents help you identify an SNA problem, classify it, and collect information about it before you call the IBM Support Center. The information collected includes traces, dumps, and other problem documentation.
z/OS Communications Server: SNA Data Areas Volume 1 and z/OS Communications Server: SNA Data Areas Volume 2	GC31-6852 GC31-6853	These documents describe SNA data areas and can be used to read an SNA dump. They are intended for IBM programming service representatives and customer personnel who are diagnosing problems with SNA.

Messages and codes

Title	Number	Description
z/OS Communications Server: SNA Messages	SC27-3671	This document describes the ELM, IKT, IST, IUT, IVT, and USS messages. Other information in this document includes: <ul style="list-style-type: none"> • Command and RU types in SNA messages • Node and ID types in SNA messages • Supplemental message-related information
z/OS Communications Server: IP Messages Volume 1 (EZA)	SC27-3654	This volume contains TCP/IP messages beginning with EZA.
z/OS Communications Server: IP Messages Volume 2 (EZB, EZD)	SC27-3655	This volume contains TCP/IP messages beginning with EZB or EZD.
z/OS Communications Server: IP Messages Volume 3 (EZY)	SC27-3656	This volume contains TCP/IP messages beginning with EYZ.
z/OS Communications Server: IP Messages Volume 4 (EZZ, SNM)	SC27-3657	This volume contains TCP/IP messages beginning with EZZ and SNM.
z/OS Communications Server: IP and SNA Codes	SC27-3648	This document describes codes and other information that appear in z/OS Communications Server messages.

Index

A

abbreviations and acronyms [771](#)
abend U4093, user [759](#)
abends
 C program [91](#)
 errno value dependency [91](#)
 return values [91](#)
 RTL functions [91](#)
 SCEERUN [91](#)
 uninitialized storage [91](#)
ACCEPT (call) [393](#)
ACCEPT (macro) [202](#)
accept() [94](#)
accessibility
 contact IBM [799](#)
addr parameter on C socket calls
 on accept() [94](#)
 on gethostbyaddr() [105](#)
address families [8](#)
address parameter on TCP/UDP/IP (pascal), on gethoststring [703](#)
address, loopback [19](#)
addressing sockets in internet domain [8](#)
addressing within sockets, network byte order [9](#)
adrrlen parameter on C socket calls
 on accept() [94](#)
 on gethostbyaddr() [105](#)
AddUserNote [699](#)
AF parameter on call interface, on SOCKET [516](#)
AF parameter on macro interface, on socket [348](#)
AF_INET [23](#)
AF_INET address family [8](#), [11](#), [761](#)
AF_IUCV [23](#)
AF_IUCV address family [761](#)
ALET parameter on macro socket interface
 on RECV [300](#)
 on RECVFROM [303](#)
 on SEND [319](#)
 on SENDTO [325](#)
 on WRITE [355](#)
allocate, socket call [23](#)
AmountOfTime parameter on TCP/UDP/IP (Pascal) , on SetTimer [714](#)
APITYPE parameter on macro interface, INITAPI call [277](#)
application program, organizing [13](#)
applications program interface (API)
 C language API [85](#)
 Pascal language API [687](#)
arg parameter on C socket calls
 on fcntl() [103](#)
 on ioctl() [129](#)
Assembler Callable Services, z/OS UNIX, general description [8](#)
assembler calls [697](#)
assembler programs, macro instructions [202](#)
assistive technologies [799](#)

asynchronous communication, (Pascal API) [687](#)
asynchronous ECB routine [199](#)
asynchronous exit routine [199](#)
asynchronous macro, coding example [200](#)
asynchronous select [40](#)
AtoETable parameter on TCP/UDP/IP (Pascal), on ReadXlateTable [711](#)
ATTACH supervisor call instruction [33](#), [43](#)

B

backlog parameter on C socket call, listen() [131](#)
BACKLOG parameter on call interface, LISTEN call [465](#)
BACKLOG parameter on macro interface, LISTEN call [288](#)
BeginTcpIp (Pascal) [700](#)
Berkeley socket implementation [91](#)
bind () [95](#)
BIND (call) [395](#)
BIND (macro) [205](#)
BIND2ADDRSEL (call) [397](#)
BIND2ADDRSEL (macro) [208](#)
bit set macros on C socket calls
 FD_CLR [141](#)
 FD_ISSET [141](#)
 FD_SET [141](#)
 FD_ZERO [141](#)
bit-mask on call interface, on EZACIC06 call [526](#)
bit-mask-length on call interface, on EZACIC06 call [526](#)
buf parameter on C socket calls
 on read() [134](#)
 on recv() [136](#)
 on recvfrom() [138](#)
 on write() [162](#)
BUF parameter on call socket interface
 on GETIBMOPT [423](#)
 on READ [470](#)
 on RECV [474](#)
 on RECVFROM [476](#)
 on SEND [491](#)
 on SENDTO [496](#)
 on WRITE [520](#)
BUF parameter on macro socket interface
 on GETIBMOPT [239](#)
 on RECV [299](#)
 on RECVFROM [302](#)
 on SEND [319](#)
 on SENDTO [325](#)
 on WRITE [355](#)
Buffer parameter on TCP/UDP/IP (Pascal) procedure
 on MonQuery [706](#)
 on RawIpReceive [709](#)
 on RawIpSend [710](#)
 on TcpFReceive, TcpReceive, TcpWaitReceive [716](#)
 on TcpFSend, TcpSend, TcpWaitSend [719](#)
bufferaddress parameter on TCP/UDP/IP (Pascal)
 procedure
 on UdpNReceive [724](#)

- bufferaddress parameter on TCP/UDP/IP (Pascal) procedure (continued)
 - on UdpSend [727](#)
- BufferLength parameter on TCP/UDP/IP (Pascal) interface
 - on RawIpReceive [709](#)
 - on TcpFSend, TcpSend, TcpWaitSend [719](#)
 - on UdpNReceive [724](#)
- BUFFERspaceAVAILABLE (Pascal) [693](#)
- bufSize parameter on TCP/UDP/IP (Pascal), on MonQuery [706](#)
- byte order parameter on C socket calls
 - on htonl() [125](#)
 - on htons() [126](#)
 - on ntohl() [133](#)
 - on ntohs() [133](#)
- byte ordering convention
 - big endian [9](#)
 - little endian [9](#)
- BytesRead parameter on TCP/UDP/IP (Pascal) procedure, on TcpFReceive, TcpReceive, TcpWaitReceive [717](#)
- BytesToRead [690](#)
- BytesToRead parameter on TCP/UDP/IP (Pascal) procedure
 - on TcpFReceive, TcpReceive, TcpWaitReceive [717](#)

C

C applications

- compiling and link-editing non-reentrant modules [86](#)
- compiling and link-editing reentrant modules [88](#)

C socket application programming interface [85](#)

C socket call syntax [93](#)

C socket calls

- accept()
 - description [94](#)
 - example [95](#)
 - return values [95](#)
 - use example [94](#)
- bind()
 - AF_INET domain example [98](#)
 - AF_IUCV domain example [98](#)
 - created in the AF_INET domain [96](#)
 - created in the AF_IUCV domain [96](#)
 - description [95](#)
 - return values [97](#)
 - use example [25](#), [26](#)
- close()
 - description [98](#)
 - example [98](#)
 - return values [99](#)
- connect()
 - description [99](#)
 - Examples [101](#)
 - return values [100](#)
 - Servers, AF_INET domain [100](#)
 - Servers, AF_IUCV domain [100](#)
- endhostent() [102](#)
- endnetent() [102](#)
- endprotoent() [102](#)
- endservent() [103](#)
- fcntl()
 - call example [104](#)
 - description [103](#)
 - return values [104](#)
- getclientid()
 - call example [104](#)

- getsockopt calls (continued)
- getclientid() (continued)
 - description [104](#)
 - return values [104](#)
- getdtablesize() [105](#)
- gethostbyaddr()
 - call example [105](#)
 - description [105](#)
 - return values [106](#)
- gethostbyname()
 - call example [106](#)
 - description [106](#)
 - return values [107](#)
- gethostent()
 - call example [107](#)
 - description [108](#)
 - return values [108](#)
- gethostid()
 - description [108](#)
 - return values [108](#)
- gethostname()
 - description [108](#)
 - return values [109](#)
- getibmopt()
 - description [109](#)
 - return values [110](#)
- getibmsockopt()
 - call example [111](#)
 - description [110](#)
 - return values [111](#)
- getnetbyaddr()
 - description [111](#)
 - return values [112](#)
- getnetbyname()
 - description [112](#)
 - return values [113](#)
- getnetent()
 - description [113](#)
 - return values [113](#)
- getpeername()
 - description [113](#), [114](#)
 - return values [114](#)
- getprotobyname()
 - description [114](#)
 - return values [115](#)
- getprotobynumber()
 - description [115](#)
 - return values [115](#)
- getprotoent()
 - description [115](#)
 - return values [116](#)
- getservbyname()
 - description [116](#)
 - return values [117](#)
- getservbyport()
 - description [117](#)
 - return values [117](#)
- getservent()
 - description [117](#)
 - return values [118](#)
- getsockname()
 - description [118](#)
 - return values [119](#)
- getsockopt()

C socket calls (*continued*)

- [getsockopt\(\) \(*continued*\)](#)
 - call example [123](#)
 - description [119](#)
 - options [120](#), [122](#), [153](#)
 - return values [122](#)
- [givesocket\(\)](#)
 - description [123](#)
 - return values [124](#)
- [htonl\(\)](#)
 - description [125](#)
 - return values [125](#)
- [htons\(\)](#)
 - description [125](#)
 - return values [126](#)
- [inet_addr\(\)](#)
 - description [126](#)
 - return values [126](#)
- [inet_lnaof\(\)](#)
 - description [127](#)
 - return values [127](#)
- [inet_makeaddr\(\)](#)
 - description [127](#)
 - return values [127](#)
- [inet_netof\(\)](#)
 - description [128](#)
 - return values [128](#)
- [inet_network\(\)](#)
 - description [128](#)
 - return values [128](#)
- [inet_ntoa\(\)](#)
 - description [128](#)
 - return values [129](#)
- [ioctl\(\)](#)
 - call example [129](#), [131](#)
 - command [129](#)
 - description [129](#)
 - return values [130](#)
- [listen\(\)](#)
 - call example [131](#)
 - description [94](#), [131](#)
 - return values [131](#)
- [maxdesc\(\)](#)
 - description [132](#)
 - examples [133](#)
 - return values [132](#)
- [ntohl\(\)](#)
 - description [133](#)
 - return values [133](#)
- [ntohs\(\)](#)
 - description [133](#)
 - return values [134](#)
- [read\(\)](#)
 - description [134](#)
 - return values [134](#)
- [readv\(\)](#)
 - description [135](#)
 - return values [135](#)
 - use example [56](#)
- [recv\(\)](#)
 - call example [136](#)
 - description [136](#)
 - return values [137](#)
 - use example [56](#)

C socket calls (*continued*)

- [recvfrom\(\)](#)
 - call example [137](#)
 - description [137](#)
 - return values [138](#)
 - use example [58](#)
- [recvmsg\(\)](#)
 - description [138](#)
 - return values [140](#)
- [select\(\)](#)
 - bit set macros [141](#)
 - call example [140](#), [142](#)
 - description [140](#)
 - return values [142](#)
 - use example [41](#), [95](#)
- [selectex\(\)](#)
 - description [143](#)
 - return values [144](#)
- [send\(\)](#)
 - call example [144](#)
 - description [56](#), [144](#)
 - return values [145](#)
 - use example [56](#)
- [sendmsg\(\)](#)
 - description [145](#)
 - return values [146](#)
- [sendto\(\)](#)
 - call example [147](#)
 - description [147](#)
 - return values [148](#)
 - use example [58](#)
- [sethostent\(\)](#)
 - description [148](#)
 - return values [148](#)
- [setibmopt\(\)](#)
 - description [149](#)
 - return values [149](#)
 - structure elements [149](#)
- [setibmssockopt\(\)](#)
 - call example [150](#), [151](#)
 - return values [150](#)
- [setnetent\(\)](#)
 - description [151](#)
 - return values [151](#)
- [setprotoent\(\)](#)
 - description [152](#)
 - return values [152](#)
- [setservent\(\)](#)
 - description [152](#)
 - return values [152](#)
- [setsockopt\(\)](#)
 - call example [156](#)
 - description [153](#)
 - options [154](#)
 - return values [155](#)
- [shutdown\(\)](#)
 - description [156](#)
 - return values [156](#)
- [sock_debug\(\)](#) [157](#)
- [sock_do_teststor\(\)](#) [157](#)
- [socket\(\)](#)
 - call examples [160](#)
 - description [158](#)
 - limitations [159](#)

- C socket calls (*continued*)
 - socket() (*continued*)
 - return values [159](#)
 - takesocket()
 - description [160](#)
 - return values [160](#)
 - tcperror()
 - call example [161](#)
 - description [161](#)
 - write()
 - description [162](#)
 - return values [162](#)
 - use example [56](#)
 - writev()
 - description [163](#)
 - return values [163](#)
 - use examples [56](#)

- C socket header files
 - bsdtypes.h [92](#), [132](#), [141](#)
 - fcntl.h [92](#)
 - if.h [92](#)
 - in.h [92](#)
 - inet.h [92](#), [96](#)
 - ioctl.h [92](#)
 - manifest.h [92](#)
 - netdb.h [92](#)
 - rtroute.h [92](#)
 - saiucv.h [92](#), [96](#)
 - socket.h [139](#), [146](#)
 - uio.h [92](#), [135](#), [163](#)

- C socket programming concepts [5](#)

- C sockets, general description [7](#)

- C structures [93](#)

- C/C++ Sockets, general description [7](#)

- CALAREA parameter on CANCEL [212](#)

- CALL Instruction Interface for Assembler, PL/I, and COBOL [389](#)

- Call Instructions for Assembler, PL/1, and COBOL Programs

- ACCEPT [393](#)
- BIND [395](#)
- BIND2ADDRSEL [397](#)
- CLOSE [400](#)
- CONNECT [401](#)
- EZACIC04 [523](#)
- EZACIC05 [524](#)
- EZACIC06 [525](#)
- EZACIC08 [527](#)
- FCNTL [404](#)
- GETCLIENTID [414](#)
- GETHOSTBYADDR [415](#)
- GETHOSTBYNAME [418](#)
- GETHOSTID [420](#)
- GETHOSTNAME [421](#)
- GETIBMOPT [422](#)
- GETPEERNAME [428](#)
- GETSOCKNAME [430](#)
- GETSOCKOPT [432](#)
- GIVESOCKET [448](#)
- INET6_IS_SRCADDR [450](#)
- INITAPI [453](#)
- IOCTL [455](#)
- LISTEN [464](#)
- READ [469](#)
- READV [470](#)

- Call Instructions for Assembler, PL/1, and COBOL Programs (*continued*)

- RECV [472](#)
- RECVFROM [474](#)
- RECVMMSG [477](#)
- SELECT [481](#)
- SELECTEX [484](#)
- SENDMSG [491](#)
- SENDTO [495](#)
- SETSOCKOPT [497](#)
- SHUTDOWN [513](#)
- SOCKET [515](#)
- TAKESOCKET [517](#)
- TERMAPI [519](#)
- WRITE [519](#)
- WRITEV [521](#)

- Call Instructions for Assembler, PL/I, and COBOL Programs

- EZACIC14 [532](#)
- EZACIC15 [533](#)

- call interface sample PL/I programs [534](#)

- call sequence [15](#)

- call syntax, C sockets [93](#)

- CallReturn parameter on TCP/UDP/IP (Pascal), on SayCalRe [712](#)

- CANCEL (macro) [211](#)

- CHAR-MASK parameter on call interface, on EZACIC06 [526](#)

- Character Generator [19](#)

- CICS (customer information control system) sockets
 - general description [7](#)

- CICS, not using tcperror() [85](#)

- Class parameter on TCP/UDP/IP (pascal), on IsLocalHost [706](#)

- ClearTimer [700](#)

- client and server socket programs [13](#)

- CLIENT parameter on call socket interface

- on GETCLIENTID [415](#)

- on GIVESOCKET [449](#)

- on TAKESOCKET [518](#)

- CLIENT parameter on macro socket interface

- on GETCLIENTID [229](#)

- on GIVESOCKET [269](#)

- on TAKESOCKET [351](#)

- client program, designing [47](#)

- client, socket calls, general

- givesocket() and takesocket() [33](#)

- send() and recv() [55](#)

- clientid parameter on C socket call

- on getclientid() [104](#)

- on givesocket() [124](#)

- on takesocket() [160](#)

- CLOSE (macro) [213](#)

- cmd parameter on C socket calls

- on fcntl() [103](#)

- on inet_addr() [126](#)

- on inet_network() [128](#)

- on ioctl() [129](#)

- COMMAND parameter on call interface, IOCTL call [456](#)

- COMMAND parameter on call socket interface

- on EZACIC06 [526](#)

- on FCNTL [405](#)

- on GETIBMOPT [423](#)

- COMMAND parameter on macro interface

- on FCNTL [219](#)

- on IOCTL [280](#)

- COMMAND parameter on macro socket interface

- COMMAND parameter on macro socket interface (*continued*)
 - on GETIBMOPT [239](#)
- Communications Server for z/OS, online information [xxxii](#)
- compiling and linking, C sockets
 - compiling and link-editing non-reentrant modules [86](#)
 - compiling and link-editing reentrant modules [88](#)
- concepts, TCP/IP [3](#)
- concurrent server program, designing [33](#)
- concurrent server socket programs [14](#)
- CONNECT (macro) [215](#)
- connect() [99](#)
- Connection (Pascal) [690](#)
- connection information record (Pascal) [690](#)
- Connection parameter on TCP/UDP/IP (Pascal) procedure
 - on TcpAbort [715](#)
 - on TcpClose [715](#)
 - on TcpFReceive, TcpReceive, TcpWaitReceive [716](#)
 - on TcpFSend, TcpSend, TcpWaitSend [719](#)
 - on TcpOption [722](#)
- Connection States (Pascal) [689](#), [691](#)
- CONNECTIONclosing (Pascal) [689](#)
- ConnectionInfo parameter on TCP/UDP/IP (Pascal)
 - procedure
 - on TcpOpen, TcpWaitOpen [721](#)
 - on TcpStatus [723](#)
- CONNECTIONstateCHANGED (Pascal) [693](#)
- ConnIndex parameter on TCP/UDP/IP (Pascal) procedure
 - on UdpClose [724](#)
 - on UdpNReceive [724](#)
 - on UdpOpen [725](#)
 - on UdpReceive [726](#)
 - on UdpSend [727](#)
- contact
 - z/OS [799](#)
- CreateTimer [701](#)
- customer information control system sockets, general
 - description, see also CICS [7](#)

D

- Data parameter on TCP/UDP/IP (Pascal), on SetTimer [714](#)
- data sets
 - hlq.AEZAMAC4 [697](#)
 - hlq.ETC.PROTO [102](#), [114](#), [115](#), [152](#)
 - hlq.ETC.SERVICES [8](#), [103](#), [116](#), [117](#), [152](#)
 - hlq.HOSTS.ADDRINFO [102](#)
 - hlq.HOSTS.SITEINFO [102](#)
 - MANIFEST.H [92](#)
 - NETDB.H [105–107](#), [111](#), [117](#)
 - SEZACMAC [173](#)
 - SEZACMTX [173](#)
 - SEZAINST [43](#), [173](#)
 - SEZALOAD [173](#)
 - TCPIP.DATA [157](#)
 - user_id.TCPIP.DATA [703](#), [720](#)
- data sets for TCP/IP programming libraries [19](#)
- data structures, Pascal [689](#)
- data transfer between sockets [51](#)
- data transfer, sockets [6](#)
- data translation, socket interface
 - ASCII to EBCDIC [524](#)
 - bit-mask to character [525](#)
 - character to bit-mask [525](#)
 - EBCDIC to ASCII [523](#), [532](#)

- DATAdelivered (Pascal) [693](#)
- datagram sockets [5](#)
- datagram sockets, program design [49](#)
- DatagramAddress parameter on TCP/UDP/IP (Pascal), on UdpReceive [726](#)
- DataLength parameter on TCP/UDP/IP (Pascal), on RawIpSend [710](#)
- debug and measurement tools
 - Character Generator [19](#)
 - Discard [19](#)
 - Echo [19](#)
- designing an iterative server program [23](#)
- DestroyTimer [701](#)
- Discard [19](#)
- DNS, online information [xxxiii](#)
- domain name system (DNS) [720](#)
- domain parameter on C socket calls
 - on getClientid() [104](#)
 - on gethostbyaddr() [105](#)
 - on socket() [158](#)
- DomainName parameter on TCP/UDP/IP (Pascal) , on GetIdentity [704](#)
- dotted decimal notation [128](#)

E

- ECB parameter on EZASMI [40](#)
- ECB parameter on macro interface
 - on ACCEPT [205](#)
 - on BIND [208](#)
 - on BIND2ADDRSEL [211](#)
 - on CANCEL [212](#)
 - on CLOSE [214](#)
 - on CONNECT [217](#)
 - on FCNTL [220](#)
 - on GETCLIENTID [229](#)
 - on GETHOSTBYNAME [232](#)
 - on GETHOSTID [235](#)
 - on GETHOSTNAME [237](#)
 - on GETPEERNAME [247](#)
 - on GETSOCKNAME [250](#)
 - on GETSOCKOPT [253](#)
 - on GIVESOCKET [270](#)
 - on INET6_IS_SRCADDR [273](#)
 - on IOCTL [286](#)
 - on LISTEN [289](#)
 - on READ [295](#)
 - on READV [297](#)
 - on RECV [300](#)
 - on RECVMFROM [304](#)
 - on RECVMMSG [308](#)
 - on SELECT [313](#)
 - on SEND [320](#)
 - on SENDMSG [323](#)
 - on SENDTO [327](#)
 - on SETSOCKOPT [330](#)
 - on SHUTDOWN [347](#)
 - on SOCKET [350](#)
 - on TAKESOCKET [352](#)
 - on WRITE [356](#)
 - on WRITEV [358](#)
- ECBPTR parameter on C socket call, shutdown () [144](#)
- Echo [19](#)
- endhostent() [102](#)

- endnetent() [102](#)
- endprotoent() [102](#)
- endservent() [103](#)
- EndTcpIp (Pascal) [701](#)
- ERETMSK parameter on call interface, on SELECT [484](#)
- ERRNO parameter on call socket interface
 - on ACCEPT [395](#)
 - on BIND [397](#)
 - on BIND2ADDRSEL [400](#)
 - on CLOSE [401](#)
 - on CONNECT [404](#)
 - on FCNTL [405](#)
 - on GETCLIENTID [415](#)
 - on GETHOSTNMAE [422](#)
 - on GETIBMOPT [424](#)
 - on GETPEERNAME [430](#)
 - on GETSOCKNAME [432](#)
 - on GETSOCKOPT [433](#)
 - on GIVESOCKET [450](#)
 - on INET6_IS_SRCADDR [452](#)
 - on INITAPI [455](#)
 - on IOCTL [463](#)
 - on LISTEN [465](#)
 - on READ [470](#)
 - on READV [472](#)
 - on RECV [474](#)
 - on RECVFROM [477](#)
 - on RECVMSG [480](#)
 - on SELECT [484](#)
 - on SELECTEX [489](#)
 - on SEND [491](#)
 - on SENDMSG [494](#)
 - on SENDTO [497](#)
 - on SETSOCKOPT [499](#)
 - on SHUTDOWN [514](#)
 - on SOCKET [517](#)
 - on TAKESOCKET [518](#)
 - on WRITE [520](#)
 - on WRITEV [522](#)
- ERRNO parameter on macro socket interface
 - on ACCEPT [204](#)
 - on BIND [207](#)
 - on BIND2ADDRSEL [210](#)
 - on CANCEL [212](#)
 - on CLOSE [214](#)
 - on CONNECT [217](#)
 - on FCNTL [220, 221, 227, 407, 414](#)
 - on GETIBMOPT [240](#)
 - on GETSOCKOPT [253, 330](#)
 - on GETSPCKNAME [250](#)
 - on GIVESOCKET [270](#)
 - on GRTCLIENTID [229](#)
 - on GRTHOSTNAME [237](#)
 - on GRTPEERNAME [247](#)
 - on INET6_IS_SRCADDR [273](#)
 - on INITAPI [277](#)
 - on IOCTL [286](#)
 - on LISTEN [288](#)
 - on READV [297](#)
 - on RECV [300](#)
 - on RECVFROM [304](#)
 - on RECVMSG [308](#)
 - on SELECT [312](#)
 - on SELECTEX [316](#)

- ERRNO parameter on macro socket interface (*continued*)
 - on SEND [320](#)
 - on SENDMSG [323](#)
 - on SENDTO [326](#)
 - on SHUTDOWN [346](#)
 - on SOCKET [349](#)
 - on TAKESOCKET [352](#)
 - on WRITE [355](#)
 - on WRITEV [357](#)
- errno values, code dependency [91](#)
- errno values, printing [91](#)
- ERRNO.H message file [91](#)
- ERROR parameter on macro interface
 - on ACCEPT [205](#)
 - on BIND [208](#)
 - on BIND2ADDRSEL [211](#)
 - on CLOSE [215](#)
 - on CONNECT [218](#)
 - on FCNTL [220, 221, 228, 245](#)
 - on GETCLIENTID [229](#)
 - on GETHOSTBYADDR [231](#)
 - on GETHOSTBYNAME [233](#)
 - on GETHOSTID [236](#)
 - on GETHOSTNAME [238](#)
 - on GETPEERNAME [248](#)
 - on GETSOCKNAME [250](#)
 - on GETSOCKOPT [253, 330](#)
 - on GIVESOCKET [270](#)
 - on INET6_IS_SRCADDR [274](#)
 - on INITAPI [278](#)
 - on IOCTL [286](#)
 - on LISTEN [289](#)
 - on RECV [301](#)
 - on RECVFROM [304](#)
 - on SELECT [313](#)
 - on SEND [320](#)
 - on SENDTO [327](#)
 - on SHUTDOWN [347](#)
 - on SOCKET [350](#)
 - on TAKESOCKET [352](#)
 - on WRITE [356](#)
- ERROR parameter on macro socket interface
 - on CANCEL [213](#)
 - on GETIBMOPT [241](#)
 - on RECVMSG [309](#)
 - on SELECTEX [318](#)
 - on SENDMSG [323](#)
 - on WRITEV [358](#)
- ESDNMASK parameter on call interface, on SELECT [484](#)
- EtoATable parameter on TCP/UDP/IP (Pascal), on ReadXlateTable [711](#)
- EWouldBLOCK error return, call interface calls
 - RECV [472](#)
 - RECVFROM [474](#)
- EWouldBLOCK error return, macro interface calls [219, 301, 356](#)
- exceptfds parameter on C socket calls
 - on select() [141](#)
 - on selectex() [143](#)
- EZACIC04, call interface, EBCDIC to ASCII translation [523](#)
- EZACIC05, call interface, ASCII to EBCDIC translation [524](#)
- EZACIC06 [39](#)
- EZACIC06, call interface, bit-mask translation [525](#)
- EZACIC08, HOSTENT structure interpreter utility [527](#)

EZACIC09, RES structure interpreter utility [529](#)
EZACIC14, call interface, EBCDIC to ASCII translation [532](#)
EZACIC15, call interface, ASCII to EBCDIC translation [533](#)
EZASOCKET
 Assembler language call format [391](#)
 COBOL language call format [391](#)
 PL/I language call format [391](#)

F

FCNTL (call) [404](#)
fcntl() [103](#)
FD_SETSIZE on send() [141](#)
file specification record (Pascal) [695](#)
FLAGS parameter on call socket interface
 on RECV [473](#)
 on RECVFROM [475](#)
 on RECVMSG [480](#)
 on SEND [490](#)
 on SENDMSG [494](#)
 on SENDTO [496](#)
FLAGS parameter on macro socket interface
 on INET6_IS_SRCADDR [273](#)
 on RECV [300](#)
 on RECVFROM [304](#)
 on RECVMSG [308](#)
 on SEND [320](#)
 on SENDMSG [323](#)
 on SENDTO [327](#)
flags, parameter on C socket calls
 on recv() [136](#)
 on recvfrom() [138](#)
 on recvmsg() [139](#)
 on send() [145](#)
 on sendmsg() [146](#)
 on sendto() [147](#)
FNDELAY flag on call interface, on FCNTL [405](#)
ForeignAddress parameter on TCP/UDP/IP (Pascal), on PingRequest [707](#)
ForeignSocket [691](#)
ForeignSocket parameter on TCP/UDP/IP (Pascal), on UdpSend [727](#)
FSENDresponse (Pascal) [694](#)

G

GETCLIENTID (call) [414](#)
GETCLIENTID (macro) [228](#)
getclientid() [104](#)
getdtablesize() [105](#)
GETHOSTBYADDR (call) [415](#)
gethostbyaddr() [105](#)
GETHOSTBYNAME (call) [418](#)
GETHOSTBYNAME (macro) [106](#), [232](#)
gethostent() [107](#)
GETHOSTID (call) [420](#)
GETHOSTID (macro) [234](#)
gethostid() [108](#)
GETHOSTNAME (call) [421](#)
GETHOSTNAME (macro) [236](#)
gethostname() [108](#)
GetHostNumber [702](#)
GetHostResol [702](#)

GetHostString [703](#)
GetIBMOpt [109](#)
GETIBMOPT (call) [422](#)
GETIBMOPT (macro) [238](#)
GetIBMSockopt [110](#)
GetIdentity [703](#)
getnetbyaddr() [111](#)
getnetbyname() [112](#)
getnetent() [113](#)
GetNextNote [704](#)
GETPEERNAME (call) [428](#)
GETPEERNAME (macro) [245](#)
getpeername() [113](#)
getprotobyname() [114](#)
getprotobynumber() [115](#)
getprotoent() [115](#)
getservbyname() [116](#)
getservbyport() [117](#)
getservent() [117](#)
GetSmsg [704](#)
GETSOCKNAME (call) [430](#)
GETSOCKNAME (macro) [248](#)
getsockname() [118](#)
GETSOCKOPT (call) [432](#)
GETSOCKOPT (macro) [251](#)
getsockopt() [119](#)
GIVESOCKET (call) [448](#)
GIVESOCKET (macro) [268](#)
givesocket() [123](#)
GLOBAL (macro) [270](#)
guidelines for using socket types [6](#)

H

Handle (Pascal) [705](#)
header files
 C sockets
 in.h [11](#), [95](#), [96](#)
 saiucv.h [96](#), [100](#)
 general, tcperrno.h [161](#)
 Pascal [688](#), [689](#)
HiperSockets Accelerator [121](#)
hisdesc parameter on C socket call, takesocket() [160](#)
host lookup routines [697](#)
HOSTADDR parameter on call interface, on GETHOSTBYADDR [416](#)
HostAddress parameter on TCP/UDP/IP (Pascal), on Handle [705](#)
HOSTADR parameter on macro socket interface, on GETHOSTBYADDR [230](#)
HOSTENT parameter on call socket interface
 on GETHOSTBYADDR [416](#)
 on GETHOSTBYNAME [419](#)
HOSTENT parameter on macro socket interface
 on GETHOSTBYADDR [230](#)
 on GETHOSTBYNAME [233](#)
HOSTENT structure interpreter parameters, on EZACIC08 [528](#)
hostname parameter on TCP/UDP/IP (Pascal), on GetIdentity [703](#)
hostnumber parameter on TCP/UDP/IP (Pascal)
 on GetHostNumber [702](#)
 on GetHostResol [702](#), [703](#)
how parameter on C socket call, on shutdown() [156](#)

HOW parameter on call interface, on SHUTDOWN [514](#)
HOW parameter on macro interface, on SHUTDOWN [346](#)
htonl() [125](#)
htons() [125](#)

I

IDENT parameter on call interface, INITAPI call [454](#)
IDENT parameter on macro interface, INITAPI call [277](#)
IMS (information management system) sockets
 general description [7](#)
in parameter on C socket calls
 on inet_lnaof() [127](#)
 on inet_netof() [128](#)
 on inet_ntoa() [129](#)
IN-BUFFER parameter on call interface, EZACIC05 call [525](#)
inet_addr() [126](#)
inet_lnaof() [127](#)
inet_makeaddr() [127](#)
inet_netof() [128](#)
inet_network() [128](#)
inet_ntoa() [128](#)
INET6_IS_SRCADDR (call) [450](#)
INET6_IS_SRCADDR(macro) [271](#)
inetdesc parameter on maxdesc(), C socket call [132](#)
Information APARs [xxx](#)
information management system socket interface, general description, see also IMS [7](#)
INITAPI(call) [453](#)
INITAPI(macro) [274](#)
initialization procedures, TCP/UDP/IP (Pascal) [696](#)
interface, C socket [5](#)
internet control message protocol (ICMP) [5](#)
internet domain, addressing sockets [8](#)
Internet, finding z/OS information online [xxxii](#)
InternetAddress parameter on TCP/UDP/IP (Pascal)
 procedure
 on SayInAd [712](#)
 on SayIntNum [713](#)
internetwork, protocol layer [5](#)
IOCTL (call) [455](#)
IOCTL (macro) [278](#)
ioctl() [129](#)
iov parameter on C socket calls
 on readv() [135](#)
 on writev() [163](#)
IOV parameter on call socket interface
 on READV [471](#)
 on WRITEV [522](#)
IOV parameter on macro socket interface
 on RECVMSG [307](#)
 on SENDMSG [322](#)
 on WRITEV [357](#)
iovcnt parameter on C socket calls
 on readv() [135](#)
 on writev() [163](#)
IOVCNT parameter on call socket interface
 on READV [472](#)
 on RECVMSG [480](#)
 on SENDMSG [494](#)
 on WRITEV [522](#)
IOVCNT parameter on macro socket interface
 on READV [297](#)
 on RECVMSG [308](#)

IOVCNT parameter on macro socket interface (*continued*)
 on SENDMSG [323](#)
 on WRITEV [357](#)
IPv6 programs [61](#)
iQDIO [121](#)
IsLocalAddress [705](#)
IsLocalHost [706](#)
iterative server socket programs [13](#)

J

JCL
 non-reentrant modules
 compiling [87](#)
 linking [88](#)
 running [88](#)
 reentrant modules
 compiling [89](#)
 prelinking and linking [90](#)
 running [90](#)

K

keyboard
 navigation [799](#)
 PF keys [799](#)
 shortcut keys [799](#)

L

language syntax, C socket call [93](#)
len parameter on C socket calls
 on read() [134](#)
 on recv() [136](#)
 on recvfrom() [138](#)
 on send() [145](#)
 on sendto() [147](#)
 on write() [162](#)
LENGTH parameter on call socket interface
 on EZACIC04 [524](#)
 on EZACIC05 [525](#)
 on EZACIC14 [532](#)
 on EZACIC15 [533](#)
length parameter on TCP/UDP/IP (Pascal) procedure
 on MonQuery [707](#)
 on PingRequest [707](#)
 on UdpSend [727](#)
level parameter on C socket calls
 on getibmssockopt() [110](#)
 on getsockopt() [119](#)
 on setibmssockopt() [150](#)
 on setsockopt() [153](#)
libraries
 Data Set [19](#)
 sockets and pascal API [92](#)
license, patent, and copyright information [801](#)
listen () [131](#)
LISTEN (call) [464](#)
LISTEN (macro) [287](#)
Listening (Pascal) [689](#)
lna parameter on inet_makeaddr(), C socket call [127](#)
LocalSocket [691](#)

LocalSocket parameter on TCP/UDP/IP (Pascal), on UdpOpen [725](#)
loopback, test address [19](#)

M

macro instruction interface for assembler programs [193](#)
macro instructions for assembler programs
 ACCEPT [202](#)
 BIND [205](#)
 BIND2ADDRSEL [208](#)
 CANCEL [211](#)
 CLOSE [213](#)
 CONNECT [215](#)
 GETCLIENTID [228](#)
 GETDHOSTBYNAME [232](#)
 GETHOSTID [234](#)
 GETHOSTNAME [236](#)
 GETIBMOPT [238](#)
 GETPEERNAME [245](#)
 GETSOCKNAME [248](#)
 GETSOCKOPT [251](#)
 GIVESOCKET [268](#)
 GLOBAL [270](#)
 INET6_IS_SRCADDR [271](#)
 INITAPI [274](#)
 IOCTL [278](#)
 LISTEN [287](#)
 READV [295](#)
 RECV [297](#)
 RECVFROM [301](#)
 RCVMSG [305](#)
 SELECT [309](#)
 SELECTEX [313](#)
 SEND [318](#)
 SENDMSG [320](#)
 SENDTO [324](#)
 SETSOCKOPT [327](#)
 SHUTDOWN [345](#)
 SOCKET [347](#)
 TAKESOCKET [350](#)
 TASK [352](#)
 TERMAPI [353](#)
 WRITE [354](#)
 WRITEV [356](#)
macro interface sample Assembler language programs [358](#)
mainframe
 education xxx
MANIFEST.H header file [92](#)
maxdesc() [132](#)
maximum number of sockets [10](#)
MAXSNO parameter on call interface, INITAPI call [454](#)
MAXSNO parameter on macro interface, INITAPI call [277](#)
MAXSOC parameter on call socket interface
 on INITAPI [454](#)
 on SELECT [483](#)
 on SELECTEX [488](#)
MAXSOC parameter on macro socket interface
 on INITAPI [276](#)
 on SELECT [312](#)
 on SELECTEX [316](#)
MISCSERV (miscellaneous server) [19](#)
monitor procedures [696](#)
MonQuery [706](#)

Motorola-style byte ordering [9](#)
msg parameter on C socket calls
 on recvmsg() [139](#)
 on send() [145](#)
 on sendmsg() [145](#)
 on sendto() [147](#)
MSG parameter on call socket interface
 on RECVMSG [479](#)
 on SENDMSG [493](#)
MSG parameter on macro call interface
 on RECVMSG [306](#), [321](#)
multicast programs [63](#)

N

name parameter on C socket calls
 on bind() [96](#)
 on connect() [100](#)
 on gethostbyname() [106](#)
 on gethostname() [109](#)
 on getnetbyname() [112](#)
 on getpeername() [114](#)
 on getprotobyname() [114](#)
 on getservbyname() [116](#)
 on getsockname() [119](#)
 on recvfrom() [138](#)
NAME parameter on call socket interface
 on ACCEPT [394](#)
 on BIND [396](#)
 on BIND2ADDRSEL [399](#)
 on CONNECT [403](#)
 on GETHOSTBYNAME [419](#)
 on GETHOSTNAME [422](#)
 on GETPEERNAME [429](#)
 on GETSOCKNAME [431](#)
 on INET6_IS_SRCADDR [451](#)
 on RECVFROM [476](#)
NAME parameter on macro interface
 on ACCEPT [204](#)
 on BIND [207](#)
 on BIND2ADDRSEL [210](#)
 on CONNECT [216](#)
 on GETHOSTBYNAME [233](#)
 on GETHOSTNAME [237](#)
 on GETPEERNAME [246](#)
 on GETSOCKNAME [249](#)
 on INET6_IS_SRCADDR [272](#)
 on RECVFROM [303](#)
 on SENDTO [326](#)
Name parameter on TCP/UDP/IP (Pascal) procedure
 on GetHostNumber [702](#)
 on GetHostResol [702](#)
 on GetHostString [703](#)
 on IsLocalHost [706](#)
namelen parameter on C socket calls
 on bind() [96](#)
 on connect() [100](#)
 on gethostname() [109](#)
 on getpeername() [114](#)
 on recvfrom() [138](#)
NAMELEN parameter on call socket interface
 on GETHOSTBYNAME [419](#)
 on GETHOSTNAME [421](#)
NAMELEN parameter on macro socket interface

- NAMELEN parameter on macro socket interface (*continued*)
 - on GETHOSTBYNAME [233](#)
 - on GETHOSTNAME [237](#)
- navigation
 - keyboard [799](#)
- NBYTE parameter on call socket interface
 - on READ [470](#)
 - on RECV [473](#)
 - on RECVFROM [476](#)
 - on SEND [490](#)
 - on SENDTO [496](#)
 - on WRITE [520](#)
- NBYTE parameter on macro socket interface
 - on RECV [299](#)
 - on RECVFROM [302](#)
 - on SEND [319](#)
 - on SENDTO [325](#)
 - on WRITE [355](#)
- net parameter on C socket call
 - on getnetbyaddr() [111](#)
 - on inet_makeaddr() [127](#)
- NETSTAT command [690](#)
- network concentrator function [121](#)
- NewNameOfTCP parameter on TCP/UDP/IP (Pascal), on TcpNameChange [720](#)
- nfds parameter on C socket calls
 - on select() [140](#)
 - on selectex() [143](#)
- non-reentrant modules, compiling and link-editing [86](#)
- NONEXISTENT (Pascal) [689](#)
- Note parameter on TCP/UDP/IP (Pascal), on GetNextNote [704](#)
- Notification parameter on TCP/UDP/IP (Pascal), on SayNoeEn [713](#)
- notification record (Pascal) [691](#)
- Notifications parameter on TCP/UDP/IP (Pascal)
 - on Handle [705](#)
 - on Unhandle [727](#)
- NotificationTag (Pascal) [693](#)
- NS parameter on macro interface
 - on ACCEPT [205](#)
 - on SOCKET [349](#)
 - on TAKESOCKET [352](#)
- ntohl() [133](#)
- ntohs() [133](#)
- number of sockets, maximum [10](#)
- NumPackets parameter on TCP/UDP/IP (Pascal), on RawIpSend [711](#)

O

- obey file [6](#)
- onoff parameter on C socket calls
 - on sock_debug() [157](#)
 - on sock_do_teststor() [157](#)
- OPEN (Pascal) [689](#)
- OpenAttemptTimeout [690](#)
- OptionName parameter on TCP/UDP/IP (Pascal), on TcpOption [722](#)
- options, getsockopt(), C socket call [120](#), [122](#), [153](#)
- OptionValue parameter on TCP/UDP/IP (Pascal), on TcpOption [722](#)
- optlen parameter on C socket calls
 - on getibmssockopt() [111](#)

- optlen parameter on C socket calls (*continued*)
 - on getsockopt() [119](#)
 - on setibmssockopt() [150](#)
 - on setsockopt() [153](#)
- optname parameter on C socket calls
 - on getibmssockopt() [110](#)
 - on getsockopt() [119](#)
 - on setibmssockopt() [150](#)
 - on setsockopt() [153](#)
- optval parameter on C socket calls
 - on getibmssockopt() [110](#)
 - on getsockopt() [119](#)
 - on setibmssockopt() [150](#)
 - on setsockopt() [153](#)
- organizing TCP/IP application program [13](#)
- OUT-BUFFER parameter on call interface, on EZACIC04 [524](#)
- OUT-BUFFER parameter on call interface, on EZACIC14 [532](#)
- OUT-BUFFER parameter on call interface, on EZACIC15 [533](#)

P

- parameters common, macro interface
 - 'value' [197](#)
 - (reg) [197](#)
 - *indaddr [197](#)
 - address [197](#)
- Pascal
 - assembler calls [697](#)
 - asynchronous communication [687](#)
 - Compiler, IBM VS Pascal and Library [688](#)
 - connection information record [690](#)
 - connection state type [689](#)
 - data structures [689](#)
 - file specification record [695](#)
 - include files
 - tcperrno.h [91](#)
 - notification record [691](#)
 - notifications [696](#)
 - procedure call usage [696](#)
 - return codes [697](#)
 - software requirements [688](#)
 - TCP/UDP/IP [687](#)
- Pascal procedure calls
 - AddUserNote [699](#)
 - BeginTcpIp [700](#)
 - ClearTimer [700](#)
 - CreateTimer [701](#)
 - DestroyTimer [701](#)
 - EndTcpIp [701](#)
 - GetHostNumber [702](#)
 - GetHostResol [702](#)
 - GetHostString [703](#)
 - GetIdentity [703](#)
 - GetNextNote [704](#)
 - GetSmsg [704](#)
 - Handle [705](#)
 - IsLocalAddress [705](#)
 - IsLocalHost [706](#)
 - MonQuery [706](#)
 - PingRequest [707](#)
 - RawIpClose [708](#)
 - RawIpOpen [708](#)
 - RawIpReceive [709](#)
 - RawIpSend [710](#)

Pascal procedure calls (*continued*)

- [ReadXlateTable 711](#)
- [SayCalRe 712](#)
- [SayConSt 712](#)
- [SayIntAd 712](#)
- [SayIntNum 713](#)
- [SayNotEn 713](#)
- [SayPorTy 713](#)
- [SayProTy 714](#)
- [SetTimer 714](#)
- [TcpAbort 714](#)
- [TcpClose 715](#)
- [TcpFReceive, TcpReceive, TcpWaitReceive 716](#)
- [TcpFSend, TcpSend, TcpWaitSend 718](#)
- [TcpNameChange 720](#)
- [TcpOpen, TcpWaitOpen 720](#)
- [TcpOption 722](#)
- [TcpStatus 723](#)
- [UdpClose 724](#)
- [UdpNReceive 724](#)
- [UdpOpen 725](#)
- [UdpReceive 726](#)
- [UdpSend 726](#)
- [Unhandle 727](#)
- Pascal sockets [7](#)
- pending activity [38](#)
- pending exception [41](#)
- pending read [41](#)
- performance [6](#)
- perror(), UNIX function [161](#)
- PING [696](#)
- PingRequest [707](#)
- PINGresponse (Pascal) [694](#)
- PL/I programs, required statement [391](#)
- port parameter on getservbyport – C socket call [117](#)
- Port parameter on TCP/UDP/IP (Pascal), on SayPorTy [713](#)
- ports, well known [8](#)
- POSIX standard
 - using z/OS UNIX C sockets API with [7](#)
- prerequisite information [xxx](#)
- program variable definitions, call interface
 - assembler definition [392](#)
 - COBOL PIC [392](#)
 - PL/I declare [392](#)
 - VS COBOL II PIC [392](#)
- programming with sockets [5](#)
- programs
 - IPv6 [61](#)
 - multicast [63](#)
- programs, client and server [13](#)
- proto parameter on C socket calls
 - on getprotobynumber() [115](#)
 - on getservbyname() [116](#)
 - on getservbyport() [117](#)
- PROTO parameter on call interface, on SOCKET [516](#)
- PROTO parameter on macro interface, on SOCKET [349](#)
- Protocol (Pascal) [692](#)
- protocol parameter on C socket call, on socket() [158](#)
- Protocol parameter on TCP/UDP/IP (Pascal), on SayProTy [714](#)
- ProtocolNo parameter on TCP/UDP/IP (Pascal) procedure
 - on RawIpClose [708](#)
 - on RawIpOpen [709](#)

ProtocolNo parameter on TCP/UDP/IP (Pascal) procedure (*continued*)

- on RawIpReceive [709](#)
- on RawIpSend [710](#)
- prototyping [93](#)
- PushFlag parameter on TCP/UDP/IP (Pascal), on TcpFSend, TcpSend, TcpWaitSend [719](#)

Q

- QueryRecord parameter on TCP/UDP/IP (Pascal), on MonQuery [707](#)

R

- Raw Ip Interface [696](#)
- raw sockets [5](#)
- RawIpClose (Pascal) [708](#)
- RawIpOpen (Pascal) [708](#)
- RAWIPpacketsDELIVERED (Pascal) [694](#)
- RawIpReceive (Pascal) [709](#)
- RawIpSend (Pascal) [710](#)
- RAWIPspaceAVAILABLE (Pascal) [694](#)
- READ (call) [469](#)
- READ (macro) [293](#)
- read() [134](#)
- readfds parameter on C socket calls
 - on select() [141](#)
 - on selectex() [143](#)
- READV (call) [470](#)
- READV (macro) [295](#)
- readv() [135](#)
- ReadXlateTable [711](#)
- RECEIVINGonly (Pascal) [689](#)
- RECV (call) [472](#)
- RECV (macro) [297](#)
- recv() [136](#)
- RECVFROM (call) [474](#)
- RECVFROM (macro) [301](#)
- recvfrom() [137](#)
- RECVMMSG (call) [477](#)
- RECVMMSG (macro) [305](#)
- recvmsg() [138](#)
- reentrant modules, compiling and link-editing [88](#)
- REQARG and RETARG parameter on call socket interface
 - on FCNTL [405](#)
 - on IOCTL [462](#)
- REQARG parameter on macro socket interface
 - on FCNTL [219](#)
 - on IOCTL [284](#)
- RESOLVE_VIA_LOOKUP, on C socket call [102](#), [106](#), [107](#)
- RESOURCESavailable (Pascal) [694](#)
- Result parameter on TCP/UDP/IP (Pascal), on GetIdentity [704](#)
- RETARG parameter on call interface, on IOCTL [463](#)
- RETARG parameter on macro interface, IOCTL call [284](#)
- RETCODE parameter on call socket interface
 - on ACCEPT [395](#)
 - on BIND [397](#)
 - on BIND2ADDRSEL [400](#)
 - on CLOSE [401](#)
 - on CONNECT [404](#)
 - on EZACIC06 [526](#)
 - on FCNTL [405](#)

RETcode parameter on call socket interface (*continued*)

- on GETCLIENTID [415](#)
- on GETHOSTBYADDR [416](#)
- on GETHOSTBYNAME [419](#)
- on GETHOSTID [421](#)
- on GETHOSTNAME [422](#)
- on GETIBMOPT [424](#)
- on GETPEERNAME [430](#)
- on GETSOCKNAME [432](#)
- on GETSOCKOPT [433](#)
- on GIVESOCKET [450](#)
- on INET6_IS_SRCADDR [452](#)
- on INITAPI [455](#)
- on IOCTL [463](#)
- on LISTEN [465](#)
- on READ [470](#)
- on READV [472](#)
- on RECV [474](#)
- on RECVFROM [477](#)
- on RECVMSG [480](#)
- on SELECT [484](#)
- on SELECTEX [489](#)
- on SEND [491](#)
- on SENDMSG [494](#)
- on SENDTO [497](#)
- on SETSOCKOPT [499](#)
- on SHUTDOWN [515](#)
- on SOCKET [517](#)
- on TAKESOCKET [518](#)
- on WRITE [520](#)
- on WRITEV [522](#)

RETcode parameter on macro socket interface

- on ACCEPT [205](#)
- on BIND [208](#)
- on BIND2ADDRSEL [210](#)
- on CANCEL [212](#)
- on CLOSE [214](#)
- on CONNECT [217](#)
- on Fcntl [220, 221, 227, 407, 414](#)
- on GETCLIENTID [229](#)
- on GETHOSTBYADDR [230](#)
- on GETHOSTID [235](#)
- on GETHOSTNAME [237](#)
- on GETIBMOPT [241](#)
- on GETPEERNAME [247](#)
- on GETSOCKNAME [250](#)
- on GETSOCKOPT [253, 330](#)
- on GIVESOCKET [270](#)
- on GRTHOSTBYNAME [233](#)
- on INET6_IS_SRCADDR [274](#)
- on INITAPI [277](#)
- on IOCTL [286](#)
- on LISTEN [288](#)
- on READV [297](#)
- on RECV [300](#)
- on RECVFROM [304](#)
- on RECVMSG [308](#)
- on SELECT [312](#)
- on SELECTEX [316](#)
- on SEND [320](#)
- on SENDMSG [323](#)
- on SENDTO [327](#)
- on SHUTDOWN [346](#)
- on TAKESOCKET [352](#)

RETcode parameter on macro socket interface (*continued*)

- on WRITE [356](#)
- on WRITEV [358](#)
- return codes
 - C sockets [94, 743](#)
 - call interface [393](#)
 - macro and call interface [754](#)
 - macro interface [197](#)
 - socket [743](#)
- return values, code dependency [91](#)
- ReturnCode parameter on TCP/UDP/IP (Pascal) procedure
 - on AddUserNote [700](#)
 - on BeginTcpip [700](#)
 - on GetNextNote [704](#)
 - on Handle [705](#)
 - on IsLocalAddress [705](#)
 - on MonQuery [706](#)
 - on Pingrequest [708](#)
 - on RawIpClose [708](#)
 - on RawIpOpen [709](#)
 - on RawIpReceive [710](#)
 - on RawIpSend [711](#)
 - on ReadXlateTable [711](#)
 - on TcpAbort [715](#)
 - on TcpClose [715](#)
 - on TcpFReceive, TcpReceive, TcpWaitReceive [717](#)
 - on TcpFSend, TcpSend, TcpWaitSend [719](#)
 - on TcpOpen, TcpWaitOpen [721](#)
 - on TcpOption [722](#)
 - on TcpStatus [723](#)
 - on UdpClose [724](#)
 - on UdpNReceive [725](#)
 - on UdpOpen [725](#)
 - on UdpReceive [726](#)
 - on UdpSend [727](#)
 - on Unhandle [728](#)

REXX sockets

- general description [7](#)

RFC (request for comments)

- accessing online [xxxii](#)

RRETMsk parameter on call interface, on SELECT [484](#)

RSNDMSK parameter on call interface, on SELECT [483](#)

RTL functions

- and return codes [91](#)
- built-in [91](#)

S

S, defines socket descriptor on C socket call

- on accept() [94](#)
- on bind() [96](#)
- on close() [99](#)
- on connect() [99](#)
- on fcntl() [103](#)
- on getibmssockopt() [110](#)
- on getpeername() [114](#)
- on getsockname() [116](#)
- on getsockopt() [119](#)
- on ioctl() [129](#)
- on listen() [131](#)
- on read() [134](#)
- on readv() [135](#)
- on recv() [136](#)
- on recvfrom() [138](#)

S, defines socket descriptor on C socket call (*continued*)

- on `recvmsg()` [139](#)
- on `send()` [144](#)
- on `sendmsg()` [145](#)
- on `sendto()` [147](#)
- on `setibmssockopt()` [150](#)
- on `setsockopt()` [153](#)
- on `shutdown()` [156](#)
- on `tcperror()` [161](#)
- on `write()` [162](#)
- on `writv()` [163](#)

S, defines socket descriptor on macro interface

- on `ACCEPT` [204](#)
- on `BIND` [206](#)
- on `BIND2ADDRSEL` [210](#)
- on `CLOSE` [214](#)
- on `CONNECT` [216](#)
- on `FCNTL` [219](#), [221–223](#), [242](#), [406](#), [408](#), [409](#), [425](#)
- on `GETPEERNAME` [246](#)
- on `GETSOCKNAME` [249](#)
- on `GETSOCKOPT` [253](#), [330](#)
- on `GIVESOCKET` [269](#)
- on `IOCTL` [280](#)
- on `LISTEN` [288](#)
- on `READV` [296](#)
- on `RECV` [299](#)
- on `RECVFROM` [302](#)
- on `RCVMSG` [306](#), [321](#)
- on `SEND` [319](#)
- on `SENDTO` [325](#)
- on `SHUTDOWN` [346](#)
- on `WRITE` [355](#)
- on `WRITEV` [357](#)

S, defines socket descriptor on socket call interface

- on `ACCEPT` [394](#)
- on `BIND` [396](#)
- on `BIND2ADDRSEL` [399](#)
- on `CLOSE` [401](#)
- on `CONNECT` [403](#)
- on `FCNTL` [405](#)
- on `GETPEERNAME` [429](#)
- on `GETSOCKNAME` [431](#)
- on `GETSOCKOPT` [433](#)
- on `GIVESOCKET` [449](#)
- on `IOCTL` [456](#)
- on `LISTEN` [465](#)
- on `READ` [470](#)
- on `READV` [471](#)
- on `RECV` [473](#)
- on `RECVFROM` [475](#)
- on `RCVMSG` [479](#)
- on `SEND` [490](#)
- on `SENDMSG` [493](#)
- on `SENDTO` [496](#)
- on `SETSOCKOPT` [498](#)
- on `SHUTDOWN` [514](#)
- on `WRITE` [520](#)
- on `WRITEV` [522](#)

sample programs

C socket

- TCP client [165](#)
- TCP server [166](#)
- UDP client [170](#)
- UDP server [168](#)

sample programs (*continued*)

call interface

- `CBLOCK`, PL/I [546](#)
- client, PL/I [537](#)
- server, PL/I [534](#)

IUCV sockets

- client, C language [44](#), [740](#)
- server, C language [44](#), [733](#)
- subtask, C language [44](#), [738](#)

macro interface

- client, assembler language [365](#)
- server, assembler language [358](#)

TCP/UDP/IP Pascal [728](#)

`SayCalRe` [712](#)

`SayConSt` [712](#)

`SayIntAd` [712](#)

`SayIntNum` [713](#)

`SayNotEn` [713](#)

`SayPorTy` [713](#)

`SayProTy` [714](#)

`SCEERUN` [91](#)

`SELECT` (call) [481](#)

`SELECT` (macro) [309](#)

select mask [38](#)

select, server, socket call, general [37](#)

`select()` [140](#)

`SELECTEX` (call) [484](#)

`SELECTEX` (macro) [313](#)

`selectex()` [143](#)

selecting sockets [5](#)

`SEND` (call) [489](#)

`SEND` (macro) [318](#)

`send()` [144](#)

`SENDINGonly` [689](#)

`SENDMSG` (call) [491](#)

`SENDMSG` (macro) [320](#)

`sendmsg()` [145](#)

`SENDTO` (call) [495](#)

`SENDTO` (macro) [324](#)

`sendto()` [147](#)

server

- `allocate()` [23](#)

- `select()` [37](#)

`sethostent()` [148](#)

`setibmopt()` [149](#)

`setibmssockopt()` [150](#)

`setnetent()` [151](#)

`setprotoent()` [152](#)

`setservent()` [152](#)

`SETSOCKOPT` (call) [497](#)

`SETSOCKOPT` (macro) [327](#)

`setsockopt()` [152](#)

`SetTimer` [714](#)

shortcut keys [799](#)

shouldwait parameter on TCP/UDP/IP (Pascal), on

`GetNextNote` [704](#)

`SHUTDOWN` (call) [513](#)

`SHUTDOWN` (macro) [345](#)

`shutdown()` [156](#)

`smmsg` parameter on TCP/UDP/IP (Pascal), on `GetSmsg` [704](#)

`SMSGreceived` (Pascal) [694](#)

`SO_BULKMODE`, on C socket calls. [110](#)

`SO_NONBLOCKLOCAL`, on C socket calls. [110](#)

`Sock_debug()` [157](#)

- SOCK_DGRAM [6](#)
- Sock_do_teststor() [157](#)
- SOCK_RAW [6](#)
- SOCK_STREAM [95](#)
- SOCKET (call) [515](#)
- SOCKET (macro) [347](#)
- socket call syntax, C [93](#)
- socket definition [5](#)
- socket libraries
 - Native TCP/IP environment [7](#)
 - UNIX environment [7](#)
- socket return codes [743](#)
- socket service types
 - datagram socket [5](#)
 - raw socket [5](#)
 - stream socket [5](#)
- socket() [158](#)
- sockets
 - addresses [10](#), [91](#)
 - connected [16](#), [56](#)
 - data transfer [6](#)
 - domain parameter [23](#)
 - guidelines for using [6](#)
 - header files
 - MANIFEST.H [92](#)
 - SOCKET.H [131](#)
 - implementation [91](#)
 - interface
 - datagram [5](#)
 - raw [5](#)
 - stream [5](#)
 - transaction [5](#)
 - library [92](#)
 - performance [6](#)
 - protocol parameter [23](#)
 - TCP socket [15](#)
 - type parameter [23](#)
 - typical TCP socket session [15](#)
 - typical UDP socket session [16](#)
 - UDP socket [16](#)
 - unconnected [16](#)
- sockets concepts [4](#)
- Sockets Extended
 - definition of call instruction API [7](#)
 - definition of macro API [7](#)
- sockets programming [5](#)
- sockets, maximum number [10](#)
- SOCRCV parameter on call interface, TAKESOCKET call [518](#)
- SOCRCV parameter on macro interface, TAKESOCKET call [352](#)
- SOCTYPE parameter on call interface, on SOCKET [516](#)
- SOCTYPE parameter on macro interface, on SOCKET [348](#)
- softcopy information xxx
- software requirements, Pascal [688](#)
- SOL_SOCKET, on C socket calls. [110](#), [119](#), [153](#)
- state parameter on TCP/UDP/IP (Pascal), on SayConSt [712](#)
- storage definition, macro interface
 - STORAGE=CSECT on EZASMI [196](#)
 - STORAGE=DSECT on EZASMI [196](#), [270](#), [352](#)
- STORAGE parameter on macro interface
 - on GLOBAL call [271](#)
 - on TASK call [353](#)
- stream sockets [5](#)
- strerror() [91](#)

- structures, C [93](#)
- SUBTASK parameter on call interface, INITAPI call [454](#)
- SUBTASK parameter on macro interface, INITAPI call [276](#)
- Success parameter on TCP/UDP/IP (Pascal), on GetSmsg [705](#)
- summary of changes xxxv
- syntax diagram, how to read xxviii
- syntax, C socket call [93](#)
- system errors, printing [85](#)

T

- T parameter on TCP/UDP/IP (Pascal) procedure
 - on ClearTimer [701](#)
 - on CreateTimer [701](#)
 - on DestroyTimer [701](#)
 - on SetTimer [714](#)
- TableName parameter on TCP/UDP/IP (Pascal), on ReadXlateTable [711](#)
- TAKESOCKET (call) [517](#)
- TAKESOCKET (macro) [350](#)
- takesocket() [160](#)
- TASK (macro) [352](#)
- task management, macro calls [199](#)
- TCP
 - communication procedures (PASCAL) [696](#)
 - socket session [15](#)
- TCP/IP
 - online information xxxii
 - protocol specifications [779](#)
- TCP/IP concepts [3](#)
- TCP/UDP/IP API (pascal language) [687](#)
- TCP/UDP/IP initialization procedures (Pascal) [696](#)
- TCP/UDP/IP termination procedure (Pascal) [696](#)
- TcpAbort (Pascal) [714](#)
- TcpClose (Pascal) [715](#)
- tcperror() [161](#)
- TcpFReceive (Pascal) [716](#)
- TcpFSend (Pascal) [718](#)
- TcpIpServiceName parameter on TCP/UDP/IP (Pascal), on GetIdentity [704](#)
- TcpNameChange [720](#)
- TcpOpen (Pascal) [720](#)
- TcpOption (Pascal) [722](#)
- TcpReceive (Pascal) [716](#)
- TcpSend (Pascal) [718](#)
- tcpserver() [91](#)
- TcpStatus (Pascal) [723](#)
- TcpWaitOpen (Pascal) [720](#)
- TcpWaitReceive [716](#)
- TcpWaitSend (Pascal) [718](#)
- Technotes xxx
- TERMAPI (call) [519](#)
- TERMAPI (macro) [353](#)
- test address, loopback [19](#)
- test tools
 - Character Generator [19](#)
 - Discard [19](#)
 - Echo [19](#)
 - loopback address [19](#)
 - miscellaneous server (MISCSRV) [19](#)
- timeout parameter on C socket calls
 - on select() [141](#)
 - on selectex() [143](#)

- TIMEOUT parameter on call interface, on SELECT [483](#)
- TIMEOUT parameter on call socket interface
 - on SELECTEX [488](#)
- TIMEOUT parameter on macro interface, on SELECT [312](#)
- TIMEOUT parameter on macro socket interface
 - on SELECTEX [317](#)
- Timeout parameter on TCP/UDP/IP (Pascal), on PingRequest [707](#)
- Timer Routines [697](#)
- TIMERexpired (Pascal) [694](#)
- to parameter on C socket calls [147](#)
- tolen parameter on C socket calls [148](#)
- totdesc parameter on C socket calls [132](#)
- trademark information [804](#)
- transaction sockets [5](#)
- transferring data between sockets [51](#)
- TranslateTableSpec parameter on TCP/UDP/IP (Pascal), on ReadXlateTable [711](#)
- transport, protocol layer [5](#)
- TRYINGtoOPEN (Pascal) [689](#)
- type parameter on C socket call
 - on getnetbyaddr() [111](#)
 - on socket() [158](#)
- typical TCP socket session [15](#)
- typical UDP socket session [16](#)

U

- U4093, user abend [759](#)
- UDP
 - communication procedures [696](#)
 - socket session [16](#)
- UdpClose (Pascal) [724](#)
- UDPdatagramDELIVERED (Pascal) [694](#)
- UDPdatagramSPACEavailable (Pascal) [695](#)
- UdpNReceive [724](#)
- UdpOpen (Pascal) [725](#)
- UdpReceive (Pascal) [726](#)
- UdpSend (Pascal) [726](#)
- Unhandle (Pascal) [727](#)
- UnpackedBytes [690](#)
- unsolicited event exit [201](#)
- urgentflag parameter on TCP/UDP/IP (Pascal) procedure, on TcpFSend, TcpSend, TcpWaitSend [719](#)
- URGENTpending (Pascal) [695](#)
- use of HOSTENT structure interpreter, EZACIC08 [527](#)
- user abend U4093 [759](#)
- user interface
 - ISPF [799](#)
 - TSO/E [799](#)
- USERdefinedNOTIFICATION (Pascal) [695](#)
- userid parameter on TCP/UDP/IP (Pascal), on GetIdentity [703](#)
- using socket implementation [91](#)
- utility programs
 - EZACIC04 [523](#)
 - EZACIC05 [524](#)
 - EZACIC06 [525](#)
 - EZACIC08 [527](#)
 - EZACIC14 [532](#)
 - EZACIC15 [533](#)

V

- Versatile Message Transfer Protocol (VMTP) [5](#)
- VTAM, online information [xxxii](#)

W

- WRETMSK parameter on call interface, on SELECT [484](#)
- WRITE (call) [519](#)
- WRITE (macro) [354](#)
- write() [162](#)
- writetds parameter on C socket calls
 - on select() [141](#)
 - on selectex() [143](#)
- WRITEV (call) [521](#)
- WRITEV (macro) [356](#)
- writetv() [163](#)
- WSNDMSK parameter on call interface, on SELECT [484](#)

X

- X/Open Transport Interface (XTI)
 - fnctl() [176](#)
 - RFC1006 [173](#)
 - select() [176](#)
 - selectex() [176](#)
 - t_accept() [175](#)
 - t_bind() [174](#)
 - t_close() [175](#)
 - t_connect() [175](#)
 - t_error() [176](#)
 - t_getinfo() [176](#)
 - t_getstate() [176](#)
 - t_listen() [175](#)
 - t_look() [175](#)
 - t_open() [174](#)
 - t_rcv() [175](#)
 - t_rcvconnect() [175](#)
 - t_rcvdis() [175](#)
 - t_snd() [175](#)
 - t_snddis() [175](#)
 - t_unbind() [175](#)
- XPG4 standard
 - using z/OS UNIX C sockets API with [7](#)
- XTI call library [173](#)
- XTI management services [173](#)

Z

- z/OS Basic Skills Information Center [xxx](#)
- z/OS UNIX return codes [754](#)
- z/OS, documentation library listing [805](#)



Product Number: 5650-ZOS

SC27-3660-70

