

z/OS Communications Server



IP Application Programming Interface Guide

Version 1 Release 6

z/OS Communications Server



IP Application Programming Interface Guide

Version 1 Release 6

Note:

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 827.

Fifth Edition (September 2004)

This edition applies to Version 1 Release 6 of z/OS (5694-A01) and Version 1 Release 6 of z/OS.e (5655-G52) and to all subsequent releases and modifications until otherwise indicated in new editions.

IBM welcomes your comments. You may send your comments to the following address.

International Business Machines Corporation
Attn: z/OS Communications Server Information Development
Department AKCA, Building 501
P.O. Box 12195, 3039 Cornwallis Road
Research Triangle Park, North Carolina 27709-2195

You can send us comments electronically by using one of the following methods:

Fax (USA and Canada):

1+919-254-4028

Internet e-mail:

- comsvrcf@us.ibm.com

World Wide Web:

<http://www.ibm.com/servers/eserver/zseries/zos/webqs.html>

If you would like a reply, be sure to include your name, address, telephone number, or FAX number. Make sure to include the following in your comment or note:

- Title and order number of this document
- Page number or topic related to your comment

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1989, 2004. All rights reserved.

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

Contents

Figures	xiii
--------------------------	-------------

Tables	xvii
-------------------------	-------------

About this document	xix
--------------------------------------	------------

Who should read this document	xix
How this document is organized	xix
How to use this document	xx
Determining if a publication is current	xx
How to contact IBM service	xx
Conventions and terminology used in this document	xxi
Clarification of notes	xxi
How to read a syntax diagram.	xxi
Symbols and punctuation	xxi
Parameters	xxii
Syntax examples	xxii
Prerequisite and related information	xxiv
Required information	xxiv
Related information	xxiv
How to send your comments	xxviii

Summary of changes.	xxix
--------------------------------------	-------------

Part 1. Overview	1
-----------------------------------	----------

Chapter 1. Introducing TCP/IP concepts	3
---	----------

TCP/IP concepts	3
Understanding sockets concepts	4
Programming with sockets.	5
Selecting sockets	5
Socket libraries	6
Address families	9
Addressing sockets in an Internet domain	9

Chapter 2. Organizing a TCP/IP application program	15
---	-----------

Client and server socket programs.	15
Iterative server socket programs	16
Concurrent server socket programs	17
Call sequence in socket programs	17
Call sequence in stream socket sessions	17
Call sequence in datagram socket sessions	18
Blocking, nonblocking, and asynchronous socket calls	19
Testing a program using a miscellaneous server	21
Testing a local machine using a loopback address	22
Accessing required data sets.	22

Part 2. Designing programs	25
---	-----------

Chapter 3. Designing an iterative server program	27
---	-----------

Allocating sockets	27
Binding sockets	29
Binding with a known port number	29
Binding using socket call gethostbyname	30

Binding a socket to a specific port number	30
Listening for client connection requests	32
Accepting client connection requests	33
Transferring data between sockets	35
Closing a connection	35
Active and passive closing	35
Shutdown call	37
Linger option.	37
Chapter 4. Designing a concurrent server program.	39
Overview	39
Concurrent servers in native MVS environment	39
MVS subtasking considerations.	40
Access to shared storage areas	40
Data set access	43
Task and workload management	43
Security considerations	44
Reentrant code	44
Understanding the structure of a concurrent server program	44
Selecting requests	45
Client connection requests	51
Passing sockets	51
Transferring data between sockets	55
Closing a concurrent server program	55
Chapter 5. Designing a client program	57
Allocating a socket	57
Connecting to a server.	57
Transferring data between sockets	59
Closing a client program	59
Chapter 6. Designing a program to use datagram sockets	61
Datagram socket characteristics.	61
Understanding datagram socket program structure	61
Allocating a socket	62
Binding sockets to port numbers	62
Streamline data transfer using connect call	62
Transferring data between sockets	62
Chapter 7. Transferring data between sockets	63
Overview	63
Streams and messages	63
Data representation.	67
Using send() and recv() calls.	68
The conversation	68
Using socket calls in a network application.	69
Reading and writing data from and to a socket	70
Using sendto() and recvfrom() calls	71
Chapter 8. Designing IPv6 programs	73
Chapter 9. Designing multicast programs.	75
IPv4 multicast options.	77
IP_ADD_MEMBERSHIP and IP_DROP_MEMBERSHIP.	77
IP_MULTICAST_IF.	81
IP_MULTICAST_LOOP	82
IP_MULTICAST_TTL	83
IPv6 multicast options.	84
IPV6_JOIN_GROUP and IPV6_LEAVE_GROUP	84

IPV6_MULTICAST_IF	88
IPV6_MULTICAST_LOOP	90
IPV6_MULTICAST_HOPS	91

Part 3. Application program interfaces 93

Chapter 10. C Socket application programming interface (API). 95

Compiler restrictions	95
Compiling and linking C applications	96
Compatibility considerations	96
Non-reentrant modules	96
Reentrant modules	99
Compiler messages	103
Program abends	103
C socket implementation	104
C socket header files	105
Manifest.h	105
Prototyping	105
C structures	106
Error messages and return codes	107
C socket calls	107
accept()	108
bind()	110
close()	114
connect()	115
endhostent()	118
endnetent()	119
endprotoent()	120
endservent()	121
fcntl()	122
getclientid()	124
getdtablesize()	125
gethostbyaddr()	126
gethostbyname()	127
gethostent()	128
gethostid()	129
gethostname()	130
getibmopt()	131
getibmsockopt()	133
getnetbyaddr()	134
getnetbyname()	135
getnetent()	136
getpeername()	137
getprotobyname()	138
getprotobynumber()	139
getprotoent()	140
getservbyname()	141
getservbyport()	142
getservent()	143
getsockname()	144
getsockopt()	145
givesocket()	151
htonl()	153
htons()	154
inet_addr()	155
inet_lnaof()	156
inet_makeaddr()	157
inet_netof()	158
inet_network()	159
inet_ntoa()	160

ioctl()	161
listen()	163
maxdesc()	164
ntohl()	166
ntohs()	167
read()	168
readv()	169
recv()	171
recvfrom()	173
recvmsg()	175
select()	177
selectex()	181
send()	183
sendmsg()	185
sendto()	187
sethostent()	189
setibmopt()	190
setibmssockopt()	192
setnetent()	194
setprotoent()	195
setservent()	196
setsockopt()	197
shutdown()	201
sock_debug()	202
sock_do_teststor()	203
socket()	204
takesocket()	207
tcperror()	208
write()	209
writew()	210
Sample C socket programs	212
Executing TCPS and TCPC modules	212
Executing UDPS and UDPC modules	212
C socket TCP client (TCPC)	212
C socket TCP server (TCPS)	215
C socket UDP server (UDPS)	218
C socket UDP client (UDPC)	221
Chapter 11. Using the X/Open Transport Interface (XTI)	225
Software requirements	225
What is provided	225
How XTI works in the z/OS environment	225
Creating an application	226
Coding XTI calls	226
Initializing a transport endpoint	226
Establishing a connection	227
Transferring data	227
Releasing a connection	227
Disabling a connection	227
Managing events	228
Using utility calls	228
Using system calls	228
Compiling and linking XTI applications using cataloged procedures	228
Understanding XTI sample programs	230
XTI socket client sample program	230
XTI socket server sample program	239
Chapter 12. Using the macro application programming interface (API)	249
Environmental restrictions and programming requirements	249
Linkage conventions for the macro API	250

Input register information	250
Output register information	250
Compatibility considerations	251
Defining storage for the API macro	251
Understanding common parameter descriptions	253
Error messages and return codes	253
Characteristics of sockets	253
Task management and asynchronous function processing	255
How it works	255
Asynchronous exit environmental and programming considerations	257
Using an unsolicited event exit routine	258
Diagnosing problems in applications using the macro API	259
Macros for assembler programs	259
ACCEPT	259
BIND	262
CANCEL	265
CLOSE	267
CONNECT	268
FCNTL	272
FREEADDRINFO	274
GETADDRINFO	275
GETCLIENTID	282
GETHOSTBYADDR	284
GETHOSTBYNAME	286
GETHOSTID	289
GETHOSTNAME	290
GETIBMOPT	292
GETNAMEINFO	294
GETPEERNAME	298
GETSOCKNAME	300
GETSOCKOPT	303
GIVESOCKET	313
GLOBAL	315
INITAPI	316
IOCTL	319
LISTEN	325
NTOP	327
PTON	329
READ	330
READV	333
RECV	334
RECVFROM	337
RECVMSG	341
SELECT	344
SELECTEX	349
SEND	352
SENDMSG	354
SENDTO	358
SETSOCKOPT	361
SHUTDOWN	371
SOCKET	373
TAKESOCKET	376
TASK	378
TERMAPI	379
WRITE	379
WRITEV	381
Macro interface assembler language sample programs	383
EZASOKAS sample server program for IPv4	384
EZASOKAC sample client program for IPv4	394
EZASO6AS sample server program for IPv6	404
EZASO6AC sample client program for IPv6	417

Chapter 13. Using the CALL instruction application programming interface (API)	429
Environmental restrictions and programming requirements	429
Linkage conventions for the CALL instruction API	430
Output register information	430
Compatibility considerations	430
CALL instruction application programming interface (API)	431
Understanding COBOL, Assembler, and PL/I call formats	431
COBOL language call format	431
Assembler language call format	431
PL/I language call format	432
Converting parameter descriptions	432
Diagnosing problems in applications using the CALL instruction API	433
Error messages and return codes	433
Code CALL instructions	433
ACCEPT	433
BIND	436
CLOSE	438
CONNECT	440
FCNTL	443
FREEADDRINFO	445
GETADDRINFO	446
GETCLIENTID	454
GETHOSTBYADDR	455
GETHOSTBYNAME	458
GETHOSTID	460
GETHOSTNAME	461
GETIBMOPT	462
GETNAMEINFO	464
GETPEERNAME	469
GETSOCKNAME	471
GETSOCKOPT	473
GIVESOCKET	482
INITAPI	485
IOCTL	487
LISTEN	493
NTOP	495
PTON	497
READ	499
READV	501
RECV	503
RECVFROM	505
RCVMSG	508
SELECT	512
SELECTEX	516
SEND	520
SENDMSG	522
SENDTO	526
SETSOCKOPT	530
SHUTDOWN	539
SOCKET	541
TAKESOCKET	543
TERMAPI	544
WRITE	545
WRITEV	547
Using data translation programs for socket call interface	548
Data translation	548
Bit string processing	548
Call interface sample programs	566
Sample code for IPv4 server program	566
Sample program for IPv4 client program	570
Sample code for IPv6 server program	573

Sample program for IPv6 client program	579
Common variables used in PL/I sample programs	583
COBOL call interface sample IPv6 server program	590
COBOL call interface sample IPv6 client program	604
Chapter 14. REXX socket application programming interface (API)	615
REXX socket initialization	615
REXX socket programming hints and tips	615
Compatibility considerations	616
Coding the socket built-in function	616
Socket	618
Error messages and return codes	619
Coding calls to process socket sets	619
Initialize	620
Socketsetlist	621
Socketset	622
Socketsetstatus	623
Terminate	624
Coding calls to initialize, change, and close sockets.	625
Accept.	626
Bind	627
Close	629
Connect	630
Givesocket	631
Listen	632
Shutdown	633
Socket	634
Takesocket	636
Coding calls to exchange data for sockets	637
Read	638
Recv	639
Recvfrom.	640
Send	642
Sendto.	643
Write	645
Coding calls to resolve names for REXX sockets.	646
Getaddrinfo	647
Getclientid	651
Getdomainname	652
Gethostbyaddr	653
Gethostbyname.	654
Gethostid.	655
Gethostname	656
Getnameinfo	657
Getpeername	659
Getprotobyname	660
Getprotobyname	661
Getservbyname.	662
Getservbyport	663
Getsockname	664
Resolve	665
Coding calls to manage configuration, options, and modes	666
Fcntl	667
Getsockopt	668
Ioctl	677
Select	679
Setsockopt	680
Version	689
REXX socket sample programs	690
The REXX-EXEC RSCIENT sample program for IPv4	690
The REXX-EXEC RSSERVER sample program for IPv4	694

The REXX-EXEC R6CLIENT sample program for IPv6	699
The REXX-EXEC R6SERVER sample program for IPv6	703

Chapter 15. Pascal application programming interface (API) 711

Steps for procedure calls.	711
Software requirements	712
Pascal API header files	712
Compatibility considerations	712
Data structures	713
Connection state	713
Connection information record	714
Notification record	715
File specification record	720
Using procedure calls	721
Notifications.	721
TCP initialization procedures	721
TCP termination procedure.	721
TCP communication procedures	722
PING interface	722
Monitor procedures	722
UDP communication procedures	722
Raw IP interface	722
Timer routines	722
Host lookup routines.	722
Assembler calls.	722
Other routines	722
Pascal return codes	723
Procedure calls	724
AddUserNote	725
BeginTcpIp	725
ClearTimer	726
CreateTimer	726
DestroyTimer	726
EndTcpIp.	726
GetHostNumber	727
GetHostResol	727
GetHostString	728
GetIdentity	728
GetNextNote	729
GetSmsg	730
Handle	730
IsLocalAddress.	730
IsLocalHost	731
MonQuery	732
PingRequest.	733
RawIpClose	733
RawIpOpen	734
RawIpReceive	735
RawIpSend	735
ReadXlateTable.	736
SayCalRe	737
SayConSt.	737
SayIntAd	738
SayIntNum	738
SayNotEn	739
SayPorTy	739
SayProTy	739
SetTimer	740
TcpAbort	740
TcpClose	741
TcpFReceive, TcpReceive, and TcpWaitReceive	741

TcpFSend, TcpSend, and TcpWaitSend	743
TcpNameChange	746
TcpOpen and TcpWaitOpen.	746
TcpOption	748
TcpStatus.	749
UdpClose	750
UdpNReceive	750
UdpOpen	751
UdpReceive	752
UdpSend	753
Unhandle.	754
Sample Pascal program	754
Building the sample Pascal API module	754
Running the sample module	754
Sample Pascal application program	755
Part 4. Appendixes	763
Appendix A. Multitasking C socket sample program.	765
Server sample program in C	765
The subtask sample program in C	772
The client sample program in C	775
Appendix B. Return codes.	781
System error codes for socket calls	781
Sockets return codes (ERRNOs)	781
z/OS UNIX return codes	790
Additional return codes	791
Sockets extended ERRNOs	791
User abend U4093.	794
Appendix C. Address family cross reference	797
Appendix D. GETSOCKOPT/SETSOCKOPT command values	803
Appendix E. Abbreviations and acronyms	805
Appendix F. Related protocol specifications (RFCs)	811
Internet Drafts	820
Appendix G. Information APARs	821
Information APARs for IP documents	821
Information APARs for SNA documents	822
Other information APARs	823
Appendix H. Accessibility	825
Using assistive technologies	825
Keyboard navigation of the user interface	825
z/OS information	825
Notices	827
Programming Interface Information	835
Trademarks	836
Bibliography.	839
z/OS Communications Server information	839
z/OS Communications Server library	839

Index	845
Communicating Your Comments to IBM	861

Figures

1. The TCP/IP protocol stack	3
2. Socket concept	4
3. TCP/IP networking API relationship on z/OS	7
4. The port concept.	10
5. Port number assignments	10
6. Iterative server main logic	16
7. A typical stream socket session	18
8. A typical datagram socket session	19
9. Socket call variables.	28
10. MVS TCP/IP socket descriptor table	29
11. An application using the bind() call	30
12. A bind() call using gethostbyname()	30
13. Variables used for the BIND call	31
14. Variables used by the listen call	32
15. Variables used by the ACCEPT call	34
16. Socket states	35
17. Closing sockets	36
18. Serialized access to a shared storage area	41
19. Synchronized use of a common service task.	42
20. Concurrent server in an MVS address space.	45
21. To set/test bits for SELECT calls	48
22. An application using the select() call	50
23. Accepting a client connection	51
24. Giving a socket to a subtask	54
25. Taking sockets from the main process.	55
26. Finding the IP address of a server host using gethostbyname()	58
27. Layout of a message between a TPI client and a TPI server	64
28. Transaction request message segment	64
29. The TCP buffer flush technique	65
30. Big or little endian byte order for a 2-byte integer.	67
31. An application using the send() and recv() calls	69
32. An application using the sendto() and recvfrom() Calls	72
33. IP_ADD_MEMBERSHIP and IP_DROP_MEMBERSHIP	78
34. IP_MULTICAST_IF	81
35. IP_MULTICAST_LOOP	82
36. IP_MULTICAST_TTL	83
37. IPV6_JOIN_GROUP and IPV6_LEAVE_GROUP	85
38. IPV6_MULTICAST_IF	89
39. IPV6_MULTICAST_LOOP.	90
40. IPV6_MULTICAST_HOPS.	91
41. Sample JCL for compiling non-reentrant modules	98
42. Sample JCL for linking non-reentrant modules	99
43. Sample JCL for running non-reentrant modules	99
44. Sample JCL for compiling reentrant modules	101
45. Sample JCL for prelinking and linking reentrant modules.	102
46. Sample JCL for running the reentrant program	103
47. C socket TCP client sample	213
48. C socket TCP server sample.	216
49. C socket UDP server sample	219
50. C socket UDP client sample.	222
51. Using XTI with TCP/IP	226
52. Sample client code for XTI	231
53. Sample server code for XTI	240
54. ECB input parameter	256
55. User token setting	256

56.	HOSTENT structure returned by the GETHOSTBYADDR macro	286
57.	HOSTENT structure returned by the GETHOSTBYNAME macro	288
58.	NUM_IMAGES field settings	293
59.	Interface request structure (IFREQ) for IOCTL macro	321
60.	Assembler language example for SIOCGIFCONF	322
61.	EZASOKAS sample server program for IPv4	385
62.	EZASOKAC sample client program for IPv4	395
63.	EZASO6AS sample server program for IPv6	405
64.	EZASO6AC sample client program for IPv6	418
65.	Storage definition statement examples	432
66.	ACCEPT call instructions example	434
67.	BIND call instruction example	437
68.	CLOSE call instruction example	439
69.	CONNECT call instruction example	441
70.	FCNTL call instruction example	444
71.	FREEADDRINFO call instruction example	445
72.	GETADDRINFO call instruction example	447
73.	GETCLIENTID call instruction example.	454
74.	GETHOSTBYADDR call instruction example	456
75.	HOSTENT structure returned by the GETHOSTBYADDR call	457
76.	GETHOSTBYNAME call instruction example	458
77.	HOSTENT structure returned by the GETHOSTBYNAME call.	459
78.	GETHOSTID call instruction example	460
79.	GETHOSTNAME call instruction example	461
80.	GETIBMOPT call instruction example	463
81.	Example of name field	464
82.	GETNAMEINFO call instruction example	466
83.	GETPEERNAME call instruction example	470
84.	GETSOCKNAME call instruction example	472
85.	GETSOCKOPT call instruction example.	474
86.	GIVESOCKET call instruction example	484
87.	INITAPI call instruction example	486
88.	IOCTL call instruction example	488
89.	COBOL language example for SIOCGHOMEIF6	490
90.	Interface request structure (IFREQ) for the IOCTL call	491
91.	COBOL language example for SIOCGIFNAMEINDEX	492
92.	COBOL II example for SIOCGIFCONF	493
93.	LISTEN call instruction example	494
94.	NTOP call instruction example.	496
95.	PTON call instruction example.	498
96.	READ call instruction example.	500
97.	READV call instruction example	502
98.	RECV call instruction example.	504
99.	RECVFROM call instruction example	506
100.	RECVMMSG call instruction example	509
101.	SELECT call instruction example	514
102.	SELECTEX call instruction example	518
103.	SEND call instruction example.	521
104.	SENDMSG call instruction example	523
105.	SENDTO call instruction example.	528
106.	SETSOCKOPT call instruction example	530
107.	SHUTDOWN call instruction example	540
108.	SOCKET call instruction example	541
109.	TAKESOCKET call instruction example	543
110.	TERMAPI call instruction example	545
111.	WRITE call instruction example	546
112.	WRITEV call instruction example	547
113.	EZACIC04 call instruction example	550
114.	EZACIC05 call instruction example	551
115.	EZACIC06 call instruction example	552
116.	EZAZIC08 call instruction example	555

117.	EZACIC09 call instruction example	558
118.	EZACIC14 EBCDIC-to-ASCII table	562
119.	EZACIC14 call instruction example	562
120.	EZACIC15 ASCII-to-EBCDIC table	564
121.	EZACIC15 call instruction example	564
122.	EZASOKPS PL/1 sample server program for IPv4	567
123.	EZASOKPC PL/1 sample client program for IPv4	571
124.	EZASO6PS PL/1 sample server program for IPv6	574
125.	EZASO6PC PL/1 sample client program for IPv6	580
126.	CBLOCK PL/1 common variables	584
127.	EZASO6CS COBOL call interface sample IPv6 server program	592
128.	EZASO6CC COBOL call interface sample IPv6 client program	605
129.	REXX-EXEC RSCLIENT sample program for IPv4	691
130.	REXX-EXEC RSSERVER sample program for IPv4	695
131.	REXX-EXEC R6CLIENT sample program for IPv6	700
132.	REXX-EXEC R6SERVER sample program for IPv6	705
133.	Pascal declaration of connection state type	713
134.	Pascal declaration of connection information record	714
135.	Pascal declaration of socket type	715
136.	Notification record	716
137.	Pascal declaration of file specification record	721
138.	Sample calling sequence	725
139.	BeginTcpIp example	725
140.	ClearTimer example	726
141.	Create timer example	726
142.	Destroy timer example	726
143.	EndTcpIp example	727
144.	GetHostNumber example	727
145.	GetHostResol example	728
146.	GetHostString example	728
147.	GetIdentity example	729
148.	GetNextNote example	729
149.	GetSmsg example	730
150.	Handle example	730
151.	IsLocalAddress example	731
152.	IsLocalHost example	731
153.	MonQuery example	732
154.	Monitor query record	732
155.	PingRequest example	733
156.	RawIpClose example	734
157.	RawIpOpen example	734
158.	RawIpReceive example	735
159.	RawIpSend example	736
160.	ReadXlateTable example	737
161.	SayCalRe example	737
162.	SayConSt example	738
163.	SayIntAd example	738
164.	SayIntNum example	738
165.	SayNotEn example	739
166.	SayPorTy example	739
167.	SayProTy example	739
168.	SetTimer example	740
169.	TcpAbort example	740
170.	TcpClose example	741
171.	TcpFReceive example	742
172.	TcpReceive example	742
173.	TcpWaitReceive example	742
174.	TcpFSend example	744
175.	TcpSend example	744
176.	TcpWaitSend example	744
177.	TcpNameChange example	746

178.	TcpOpen example	746
179.	TcpWaitOpen example	746
180.	TcpOption example	748
181.	TcpStatus example	749
182.	UdpClose example.	750
183.	UdpNReceive example	751
184.	UdpOpen example.	752
185.	UdpReceive example	752
186.	UdpSend example	753
187.	Unhandle example.	754
188.	Sample Pascal API with receive option	755
189.	Sample Pascal API with send option	755
190.	Sample Pascal application program	756
191.	MTCSRVR C socket server program sample	766
192.	MTCCSUB C socket server program sample	773
193.	MTCCCLNT C socket server program sample	776
194.	Example of abend U4093.	795

Tables

1.	Socket programming interface actions	20
2.	TCP/IP data sets and applications	22
3.	Effect of shutdown socket call	37
4.	First fullword passed in a bit string select()	46
5.	Second fullword passed in a bit string using select()	47
6.	C/C++/390 R4 compiler messages	103
7.	C structures in assembler language format	106
8.	Initializing a call	227
9.	Establishing a connection	227
10.	Transferring data	227
11.	Releasing a connection	227
12.	Disabling a connection	227
13.	Managing events	228
14.	Using utilities	228
15.	System function calls	228
16.	OPTNAME options for GETSOCKOPT and SETSOCKOPT	305
17.	IOCTL macro arguments	324
18.	OPTNAME options for GETSOCKOPT and SETSOCKOPT	364
19.	OPTNAME options for GETSOCKOPT and SETSOCKOPT	475
20.	IOCTL call arguments	492
21.	OPTNAME options for GETSOCKOPT and SETSOCKOPT	532
22.	OPTNAME options for GETSOCKOPT and SETSOCKOPT	669
23.	OPTNAME options for GETSOCKOPT and SETSOCKOPT	681
24.	TCP connection states	714
25.	Pascal language return codes	723
26.	Sockets ERRNOs	781
27.	Sockets extended ERRNOs	791
28.	C socket address families cross reference	797
29.	MACRO, CALL, REXX, socket address families cross reference	799
30.	MACRO, CALL, REXX, exceptions	801
31.	GETSOCKOPT/SETSOCKOPT command values for Macro, Assembler, and PL/I.	803
32.	GETSOCKOPT/SETSOCKOPT optname value for C programs	803
33.	IP information APARs for z/OS Communications Server	821
34.	SNA information APARs for z/OS Communications Server	822
35.	Non-document information APARs	823

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.

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

This document supports z/OS.e.

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 book, 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 25 describes ways to design various types of programs.

Part 3, "Application program interfaces," on page 93 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)

"Part 4. 'Appendices' " provides additional information for this document.

"Notices" contains notices and trademarks used in this document.

"Bibliography" 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.

Determining if a publication is current

As needed, IBM updates its publications with new and changed information. For a given publication, updates to the hardcopy and associated BookManager® softcopy are usually available at the same time. Sometimes, however, the updates to hardcopy and softcopy are available at different times. The following information describes how to determine if you are looking at the most current copy of a publication:

- At the end of a publication's order number there is a dash followed by two digits, often referred to as the dash level. A publication with a higher dash level is more current than one with a lower dash level. For example, in the publication order number GC28-1747-07, the dash level 07 means that the publication is more current than previous levels, such as 05 or 04.
- If a hardcopy publication and a softcopy publication have the same dash level, it is possible that the softcopy publication is more current than the hardcopy publication. Check the dates shown in the Summary of Changes. The softcopy publication might have a more recently dated Summary of Changes than the hardcopy publication.
- To compare softcopy publications, you can check the last two characters of the publication's filename (also called the book name). The higher the number, the more recent the publication. Also, next to the publication titles in the CD-ROM booklet and the readme files, there is an asterisk (*) that indicates whether a publication is new or changed.

How to contact IBM service

For immediate assistance, visit this Web site:

<http://www.software.ibm.com/network/commserver/support/>

Most problems can be resolved at this Web site, where you can submit questions and problem reports electronically, as well as access a variety of diagnosis information.

For telephone assistance in problem diagnosis and resolution (in the United States or Puerto Rico), call the IBM Software Support Center anytime (1-800-IBM-SERV). You will receive a return call within 8 business hours (Monday – Friday, 8:00 a.m. – 5:00 p.m., local customer time).

Outside of the United States or Puerto Rico, contact your local IBM representative or your authorized IBM supplier.

If you would like to provide feedback on this publication, see “Communicating Your Comments to IBM” on page 861.

Conventions and terminology used in this document

For definitions of the terms and abbreviations used in this document, you can view the latest IBM terminology at the IBM Terminology Web site.

Clarification of notes

Information traditionally qualified as **Notes** is further qualified as follows:

Note Supplemental detail

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

Guideline

Customary way to perform a procedure; stronger request than recommendation

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

How to read a syntax diagram

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

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.

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.

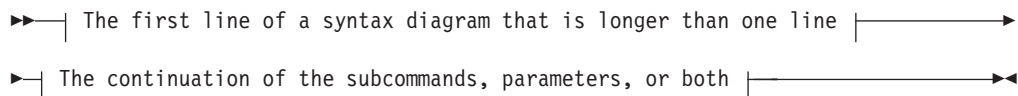
Syntax examples

In the following example, the USER command is a keyword. The required variable parameter is *user_id*, and the optional variable parameter is *password*. 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.

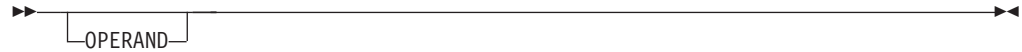
Choose one required item from a stack

If there is more than one mutually exclusive required operand or value to choose from, they are stacked vertically.



Optional values

Optional operands and values appear below the main path line.



You can choose not to code optional operands and values.

Choose one optional operand from a stack

If there is more than one mutually exclusive optional operand or value to choose from, they are stacked vertically below the main path line.



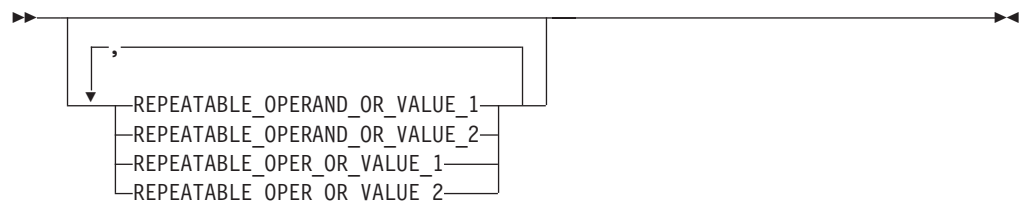
Repeating an operand

An arrow returning to the left above an operand or value on the main path line means that the operand or value can be repeated. The comma means that each operand or value must be separated from the next by a comma. If no comma appears in the returning arrow, the operand or value must be separated from the next by a blank.



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.

▶▶

DEFAULT
OPERAND

—▶▶

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 |—▶▶

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 “z/OS Communications Server information” on page 839, 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.

Related information

This section contains subsections on:

- “Softcopy information” on page xxv
- “Other documents” on page xxv
- “Redbooks” on page xxvi
- “Where to find related information on the Internet” on page xxvi

- “Accessing z/OS licensed documents on the Internet” on page xxvii
- “Using LookAt to look up message explanations” on page xxviii

Softcopy information

Softcopy publications are available in the following collections:

Titles	Order Number	Description
<i>z/OS V1R6 Collection</i>	SK3T-4269	This is the CD collection shipped with the z/OS product. It includes the libraries for z/OS V1R6, in both BookManager and PDF formats.
<i>z/OS Software Products Collection</i>	SK3T-4270	This CD includes, in both BookManager and PDF formats, the libraries of z/OS software products that run on z/OS but are not elements and features, as well as the <i>Getting Started with Parallel Sysplex</i> [®] bookshelf.
<i>z/OS V1R6 and Software Products DVD Collection</i>	SK3T-4271	This collection includes the libraries of z/OS (the element and feature libraries) and the libraries for z/OS software products in both BookManager and PDF format. This collection combines SK3T-4269 and SK3T-4270.
<i>z/OS Licensed Product Library</i>	SK3T-4307	This CD includes the licensed documents in both BookManager and PDF format.
<i>System Center Publication IBM S/390[®] Redbooks[™] Collection</i>	SK2T-2177	This collection contains over 300 ITSO redbooks that apply to the S/390 platform and to host networking arranged into subject bookshelves.

Other documents

For information about z/OS products, refer to *z/OS Information Roadmap* (SA22-7500). The Roadmap describes what level of documents are supplied with each release of z/OS Communications Server, as well as describing each z/OS publication.

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 may be helpful to readers.

Title	Number
<i>z/OS Integrated Security Services Firewall Technologies</i>	SC24-5922
<i>S/390: OSA-Express Customer's Guide and Reference</i>	SA22-7403
<i>z/OS JES2 Initialization and Tuning Guide</i>	SA22-7532
<i>z/OS MVS Diagnosis: Procedures</i>	GA22-7587
<i>z/OS MVS Diagnosis: Reference</i>	GA22-7588
<i>z/OS MVS Diagnosis: Tools and Service Aids</i>	GA22-7589
<i>z/OS Integrated Security Services LDAP Client Programming</i>	SC24-5924
<i>z/OS Integrated Security Services LDAP Server Administration and Use</i>	SC24-5923
<i>Understanding LDAP</i>	SG24-4986
<i>z/OS UNIX System Services Programming: Assembler Callable Services Reference</i>	SA22-7803
<i>z/OS UNIX System Services Command Reference</i>	SA22-7802
<i>z/OS UNIX System Services User's Guide</i>	SA22-7801

Title	Number
<i>z/OS UNIX System Services Planning</i>	GA22-7800
<i>z/OS MVS Using the Subsystem Interface</i>	SA22-7642
<i>z/OS C/C++ Run-Time Library Reference</i>	SA22-7821
<i>z/OS Program Directory</i>	GI10-0670
<i>DNS and BIND</i> , Fourth Edition, O'Reilly and Associates, 2001	ISBN 0-596-00158-4
<i>Routing in the Internet</i> , Christian Huitema (Prentice Hall PTR, 1995)	ISBN 0-13-132192-7
<i>sendmail</i> , Bryan Costales and Eric Allman, O'Reilly and Associates, 2002	ISBN 1-56592-839-3
<i>TCP/IP Tutorial and Technical Overview</i>	GG24-3376
<i>TCP/IP Illustrated, Volume I: The Protocols</i> , W. Richard Stevens, Addison-Wesley Publishing, 1994	ISBN 0-201-63346-9
<i>TCP/IP Illustrated, Volume II: The Implementation</i> , Gary R. Wright and W. Richard Stevens, Addison-Wesley Publishing, 1995	ISBN 0-201-63354-X
<i>TCP/IP Illustrated, Volume III</i> , W. Richard Stevens, Addison-Wesley Publishing, 1995	ISBN 0-201-63495-3
<i>z/OS Cryptographic Service System Secure Sockets Layer Programming</i>	SC24-5901
<i>SNA Formats</i>	GA27-3136

Redbooks

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

Title	Number
<i>TCP/IP Tutorial and Technical Overview</i>	GG24-3376
<i>SNA and TCP/IP Integration</i>	SG24-5291
<i>IBM Communications Server for OS/390® V2R10 TCP/IP Implementation Guide: Volume 1: Configuration and Routing</i>	SG24-5227
<i>IBM Communications Server for OS/390 V2R10 TCP/IP Implementation Guide: Volume 2: UNIX Applications</i>	SG24-5228
<i>IBM Communications Server for OS/390 V2R7 TCP/IP Implementation Guide: Volume 3: MVS Applications</i>	SG24-5229
<i>Secureway Communications Server for OS/390 V2R8 TCP/IP: Guide to Enhancements</i>	SG24-5631
<i>TCP/IP in a Sysplex</i>	SG24-5235
<i>Managing OS/390 TCP/IP with SNMP</i>	SG24-5866
<i>Security in OS/390-based TCP/IP Networks</i>	SG24-5383
<i>IP Network Design Guide</i>	SG24-2580
<i>Migrating Subarea Networks to an IP Infrastructure</i>	SG24-5957
<i>IBM Communication Controller Migration Guide</i>	SG24-6298

Where to find related information on the Internet

z/OS

- <http://www.ibm.com/servers/eserver/zseries/zos/>

z/OS Internet Library

- <http://www.ibm.com/servers/eserver/zseries/zos/bkserv/>

IBM Communications Server product

- <http://www.software.ibm.com/network/commserver/>

IBM Communications Server product support

- <http://www.software.ibm.com/network/commserver/support/>

IBM Systems Center publications

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

IBM Systems Center flashes

- <http://www-1.ibm.com/support/techdocs/atmastr.nsf>

RFCs

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

Internet drafts

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

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

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

USENET news groups:

comp.protocols.dns.bind

For BIND mailing lists, see:

- <http://www.isc.org/ml-archives/>
 - 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 (Note: This list may 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.

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

Accessing z/OS licensed documents on the Internet

z/OS licensed documentation is available on the Internet in PDF format at the IBM Resource Link™ Web site at:

<http://www.ibm.com/servers/resourcelink>

Licensed documents are available only to customers with a z/OS license. Access to these documents requires an IBM Resource Link user ID and password, and a key code. With your z/OS order you received a Memo to Licensees, (GI10-0671), that includes this key code.

To obtain your IBM Resource Link user ID and password, log on to:

<http://www.ibm.com/servers/resourcelink>

To register for access to the z/OS licensed documents:

1. Sign in to Resource Link using your Resource Link user ID and password.
2. Select **User Profiles** located on the left-hand navigation bar.

Note: You cannot access the z/OS licensed documents unless you have registered for access to them and received an e-mail confirmation informing you that your request has been processed.

Printed licensed documents are not available from IBM.

You can use the PDF format on either **z/OS Licensed Product Library CD-ROM** or IBM Resource Link to print licensed documents.

Using LookAt to look up message explanations

LookAt is an online facility that allows you to look up explanations for most messages you encounter, as well as for some system abends and codes. Using LookAt to find information is faster than a conventional search because in most cases LookAt goes directly to the message explanation.

You can access LookAt from the Internet at:

<http://www.ibm.com/eserver/zseries/zos/bkserv/lookat/>

or from anywhere in z/OS where you can access a TSO/E command line (for example, TSO/E prompt, ISPF, z/OS UNIX System Services running OMVS). You can also download code from the *z/OS Collection* (SK3T-4269) and the LookAt Web site that will allow you to access LookAt from a handheld computer (Palm Pilot VIIx suggested).

To use LookAt as a TSO/E command, you must have LookAt installed on your host system. You can obtain the LookAt code for TSO/E from a disk on your *z/OS Collection* (SK3T-4269) or from the **News** section on the LookAt Web site.

Some messages have information in more than one document. For those messages, LookAt displays a list of documents in which the message appears.

How to send your comments

Your feedback is important in helping to provide the most accurate and high-quality information. If you have any comments about this document or any other z/OS Communications Server documentation:

- Go to the z/OS contact page at:
<http://www.ibm.com/servers/eserver/zseries/zos/webqs.html>
There you will find the feedback page where you can enter and submit your comments.
- Send your comments by e-mail to comsvrcf@us.ibm.com. Be sure to include the name of the document, the part number of the document, the version of z/OS Communications Server, and, if applicable, the specific location of the text you are commenting on (for example, a section number, a page number or a table number).

Summary of changes

Summary of changes for SC31-8788-04 z/OS Version 1 Release 6

This document contains information previously presented in SC31-8788-03, which supports z/OS Version 1 Release 5. 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.

New Information

- Commands and select examples are enabled for z/OS library center advanced searches.

Changed information

- Updated getsockopt() information, see “getsockopt()” on page 145.

This document contains terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations are indicated by a vertical line to the left of the change.

Starting with z/OS V1R4, you may notice changes in the style and structure of some content in this document—for example, headings that use uppercase for the first letter of initial words only, and procedures that have a different look and format. The changes are ongoing improvements to the consistency and retrievability of information in our documents.

Summary of changes for SC31-8788-03 z/OS Version 1 Release 5

This document contains information previously presented in SC31-8788-02, which supports z/OS Version 1 Release 4. 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.

Changed information

- EZASMI now supports MAXSOC values up to 65535 and socket numbers up to 65534. See “Maximum number of sockets” on page 11 and “INITAPI” on page 316.
- The shutdown() flow has been altered to abort a TCP connection if there is data on the receive queue. See “Shutdown call” on page 37.
- Inbound processing has been altered to abort a TCP connection when new data arrives and the socket has been shutdown for read. See “Shutdown call” on page 37.
- The description for EZASMI TYPE=BIND has been modified. See “BIND” on page 262.

This document contains terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations are indicated by a vertical line to the left of the change.

Starting with z/OS V1R4, you may notice changes in the style and structure of some content in this document—for example, headings that use uppercase for the first letter of initial words only, and procedures that have a different look and format. The changes are ongoing improvements to the consistency and retrievability of information in our documents.

Summary of changes for SC31-8788-02 z/OS Version 1 Release 4

This document contains information previously presented in SC31-8788-01, which supports z/OS Version 1 Release 2. 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.

New information

- New chapter, see Chapter 8, “Designing IPv6 programs,” on page 73.
- New chapter, see Chapter 9, “Designing multicast programs,” on page 75.
- New appendix, see Appendix D, “GETSOCKOPT/SETSOCKOPT command values,” on page 803.
- The following areas contain new information pertaining to IPv6 support and addressing changes from 32 bit to 128 bit for many API calls.
 - Macro APIs. See Chapter 12, “Using the macro application programming interface (API),” on page 249.
 - CALL instruction APIs. See Chapter 13, “Using the CALL instruction application programming interface (API),” on page 429.
 - REXX socket APIs. See Chapter 14, “REXX socket application programming interface (API),” on page 615.
 - Sockets ERRNOs. See Appendix B, “Return codes,” on page 781.

An appendix with z/OS product accessibility information has been added.

Changed information

- AF_INET is changed to AF_INET or AF_INET6, see Appendix B, “Return codes,” on page 781.
- IP_DROP_MULTICAST is changed to IP_DROP_MEMBERSHIP, see “setsockopt()” on page 197.

Deleted information

- BULKMODE support was removed from GETIBMSOCKOPT and SETIBMSOCKOPT.

This document contains terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations are indicated by a vertical line to the left of the change.

Starting with z/OS V1R4, you may notice changes in the style and structure of some content in this document—for example, headings that use uppercase for the first letter of initial words only, and procedures that have a different look and

format. The changes are ongoing improvements to the consistency and retrievability of information in our documents.

This document supports z/OS.e.

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. Refer to *z/OS Communications Server: IPv6 Network and Application 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, refer to Appendix C, “Address family cross reference,” on page 797.

Chapter 1. Introducing TCP/IP concepts

This chapter explains basic TCP/IP concepts and sockets programming issues, including the following:

- TCP/IP concepts
- Understanding sockets concepts

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 illustrates the TCP/IP protocol stack.

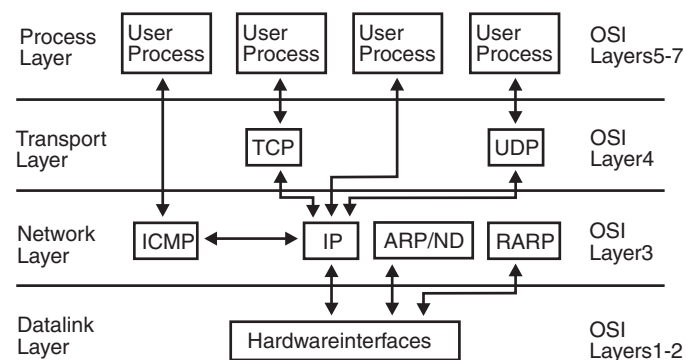


Figure 1. The TCP/IP protocol stack

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 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 all three 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 illustrates the concept of a socket.

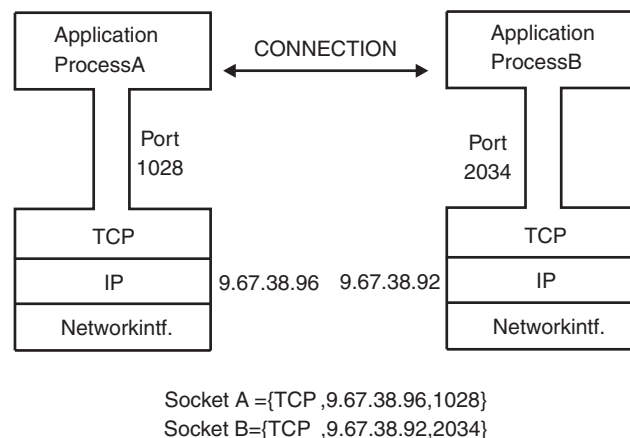


Figure 2. Socket concept

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 halfword (2-byte integer) that acts as an index to a table of sockets currently allocated to a given process. A socket descriptor represents the socket, but is not the socket itself.

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.

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).¹ Transaction-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.
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, corrupted, 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.

Socket libraries

Figure 3 on page 7 illustrates the TCP/IP networking API relationship on z/OS.

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.

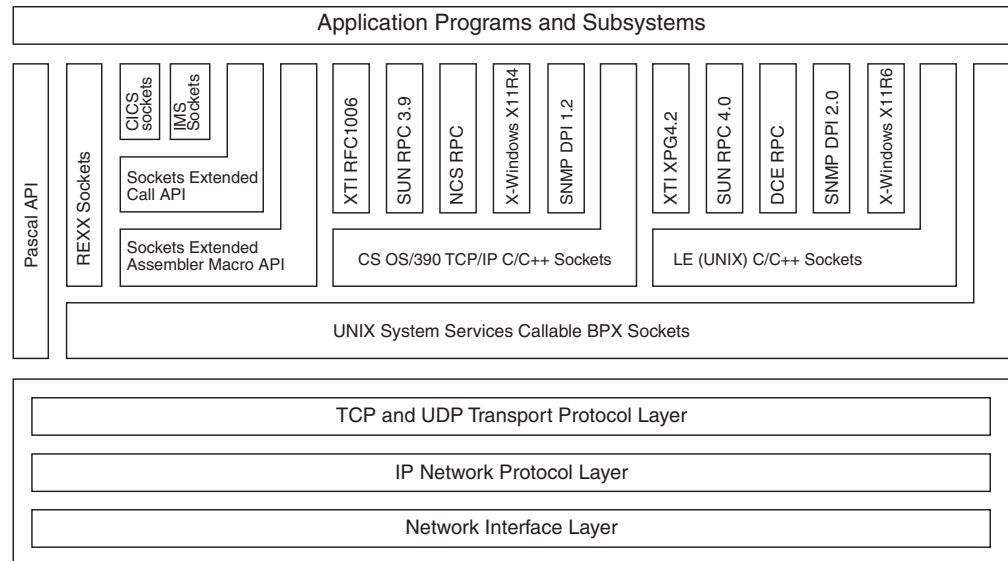


Figure 3. TCP/IP networking API relationship on z/OS

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 CS)
- UNIX (implemented by OS/390 UNIX System Services (Language Environment[®]))

Native TCP/IP

A non-UNIX socket program can only use 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 directly with the LFS. 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, refer to the *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, refer to the *z/OS Communications Server: IP CICS Sockets Guide*.

CS OS/390 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 is 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, refer to the *z/OS C/C++ Run-Time 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, refer to the *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

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 “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.

Addressing sockets in an Internet domain

This section 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. Refer to *z/OS Communications Server: IPv6 Network and Application 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, SMTP, 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 CS 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, SMTP, and Telnet. Data set *hlq.ETC.SERVICES* is described in *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 shows the port concept.

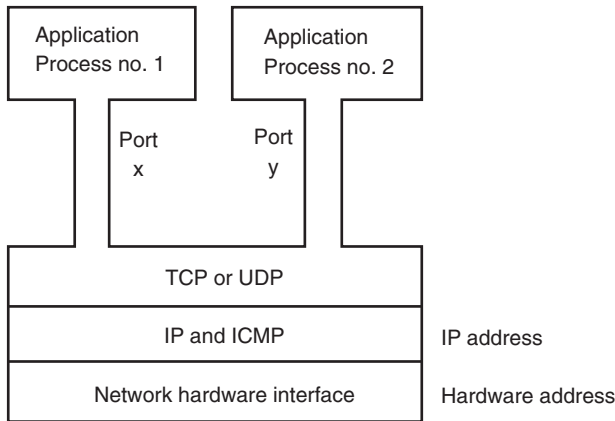


Figure 4. The port concept

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 to 255. You can find a list of these port numbers in Assigned Numbers, RFC 1700. In addition, port numbers in the range of 256 to 1023 are reserved for other well-known services. Port numbers in the range of 1024 to 5000 are used by TCP/IP when TCP/IP automatically assigns port numbers to client programs that do not use a specific port number. Your server applications should use port numbers above 5000.

Figure 5 shows port number assignments.

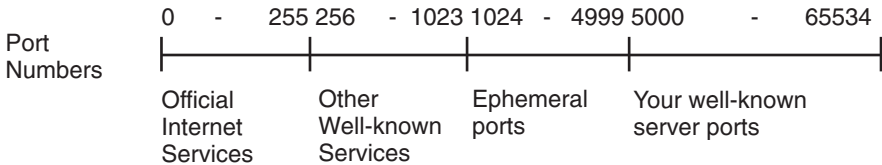


Figure 5. Port number assignments

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 108, “`bind()`” on page 110, “`htonl()`” on page 153, “`htons()`” on page 154, “`ntohl()`” on page 166 and “`ntohs()`” on page 167 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 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 65535 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 than 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 UNIX System Services process can have open. Each address space is usually a 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, refer to 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 comprises 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

Refer to *z/OS Communications Server: IPv6 Network and Application 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 chapter explains how to organize a TCP/IP application program. All examples within this chapter 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.

The client and server distribution model is structured on the roles of master and slave; the client acts as the master and requests service from the server (the slave). The server responds to the request from the client. This model implies a one-to-many relationship; the server typically serves multiple clients, while each client deals with a single server.

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.

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 shows the iterative server main logic.

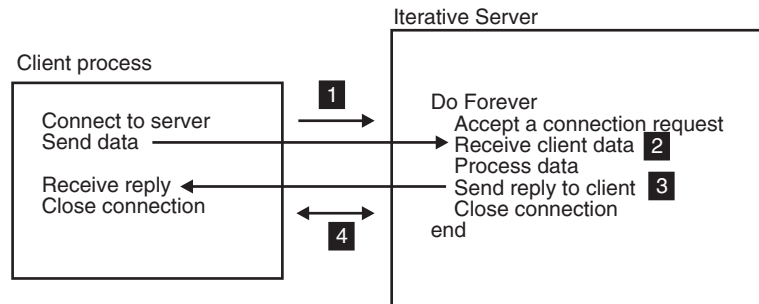


Figure 6. Iterative server main logic

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.

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.

Call sequence in socket programs

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

Call sequence in stream socket sessions

This section 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 212 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 18 shows the general sequence of calls for most socket routines using stream sockets.

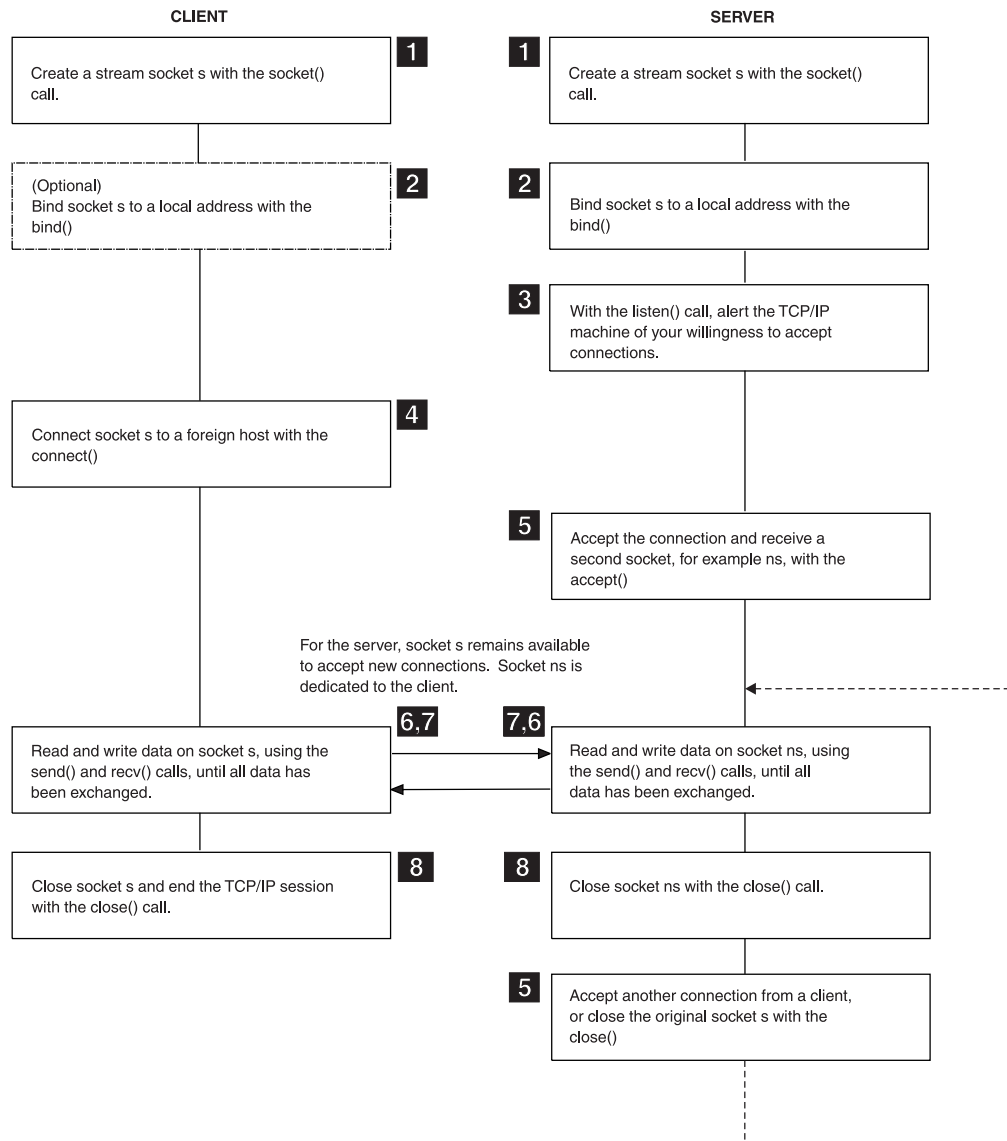


Figure 7. A typical stream socket session

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 shows the general sequence of calls for socket routines using datagram sockets.

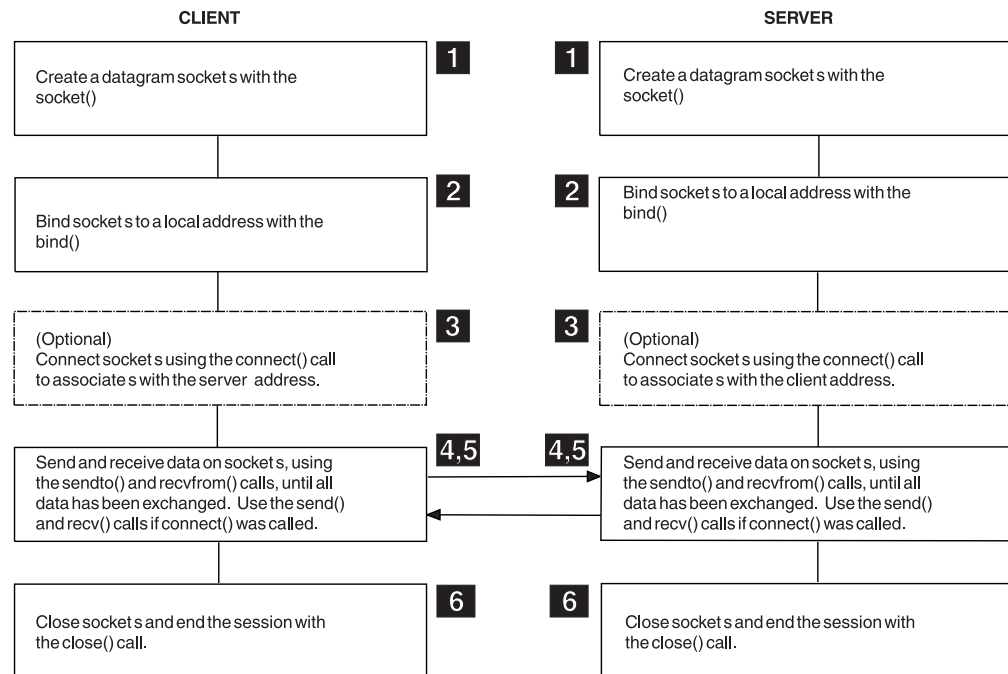


Figure 8. A typical datagram socket session

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 full word (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 (`F'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 255.

Table 1 lists the actions taken by the socket programming interface.

Table 1. Socket programming interface actions

Call type	Socket state	blocking	Nonblocking
Types of read() 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 write() 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>)
accept() 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>)
connect() 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.

Testing a program using a miscellaneous server

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

Tool Server description

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

Discard
Specify Port 9 when you want MISC SERV to discard the data.

Character Generator

Specify Port 19 when you want MISC SERV 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 *z/OS Communications Server: IP Configuration Reference*.

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 127.0.0.0 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 127.0.0.0 (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.

Accessing required data sets

Table 2 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
hlq.SEZABPDM	Kerberos B-plus tree database operations
hlq.SEZACMAC	Client Pascal macros, C headers, and assembler macros
hlq.SEZACMTX	Sockets and Pascal API
hlq.SEZADES	Kerberos encryption function, US Kerberos only
hlq.SEZADPIL	SNMP DPI [®]
hlq.SEZAKDB	Kerberos database administration
hlq.SEZAKRB	Kerberos ticket authentications
hlq.SEZALIBN	NCS
hlq.SEZAOLDX	X Release 10 compatibility routines
hlq.SEZANMAC	C headers and assembler macros for z/OS UNIX and TCP/IP Services APIs
hlq.SEZARNT1	Sockets, X11, and PEXlib (reentrant)
hlq.SEZARNT2	Athena widget (reentrant)
hlq.SEZARNT3	Motif widget (reentrant)
hlq.SEZARPCL	Remote procedure calls
hlq.SEZAXAWL	Athena widget set

Table 2. TCP/IP data sets and applications (continued)

Data set	Application
<i>hlq</i> .SEZAXMLB	OSF/Motif-based widget set
<i>hlq</i> .SEZAXTLB	Xt Intrinsic
<i>hlq</i> .SEZAX11L	Xlib, Xmu, Xext, and Xau routines

Part 2. Designing programs

Chapter 3. Designing an iterative server program

All examples within this chapter 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).

- Allocating sockets
- Binding sockets
- Listening for client connection requests
- Accepting client connection requests
- Transferring data between sockets
- Closing a connection

Allocating sockets

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 204.

```
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 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 'EZASOKET' 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 29 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

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_ x, 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 30. For more information about binding, see “bind()” on page 110.

```

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 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 155.
- 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 154.

Binding using socket call `gethostbyname`

Figure 12 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 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 'EZASOCKET' using soket-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). Refer to *z/OS Communications Server: IPv6 Network and Application 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

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.

```
*-----*
* 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 'EZASOKET' using soket-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 rejected 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

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 is an example of 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 wishes 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.

```

*-----*
* 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 'EZASOCKET' 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:

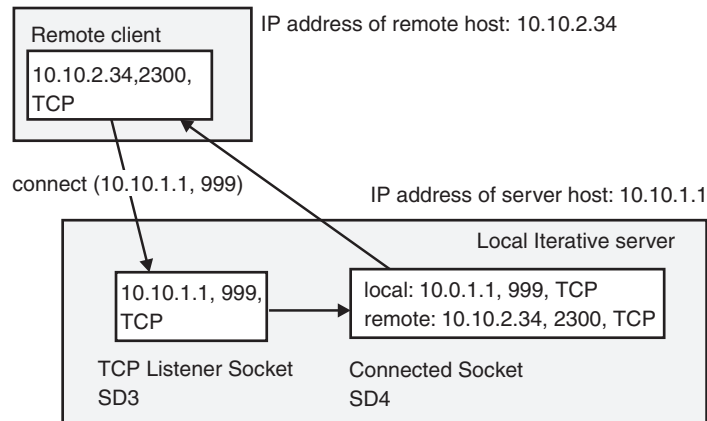
- The first socket descriptor represents the socket that was obtained, bound to the server port and (optionally) the IP address, and changed to the passive state using the listen() call.
- The accept() call returns a new socket descriptor, to represent a complete association:

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 35 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 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 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

See Chapter 7, “Transferring data between sockets,” on page 63.

Closing a connection

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 sections describe various ways to close a connection.

Active and passive closing

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 illustrates socket closing.

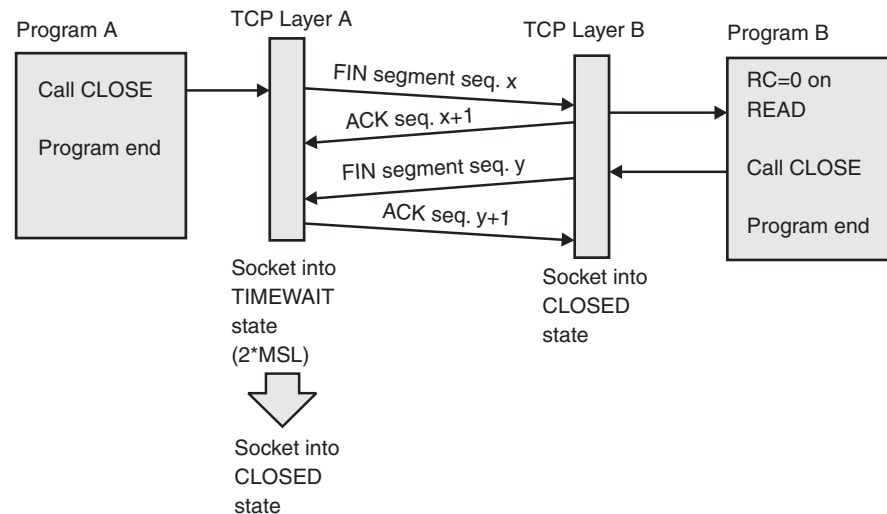


Figure 17. Closing sockets

In Figure 17, 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` (`FIN`ish). 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 one to four 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

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 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

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 full word 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 chapter describes concurrent server programs. All examples within this chapter 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). These programs include the following:

- An overview
- Concurrent servers in the native MVS environment
- MVS subtasking considerations
- Understanding the structure of a concurrent server program
- Selecting requests
- Client connection requests
- Transferring data between sockets
- Closing a concurrent server program

Overview

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 already 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.

For the MVS address space examples presented in this chapter, 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 both 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 chapter, we discuss implementation of a concurrent server in an MVS address space.

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

MVS subtasking considerations

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

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 41 illustrates access to a shared storage area.

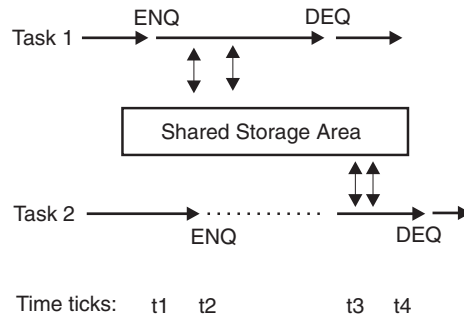


Figure 18. Serialized access to a shared storage area

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 42.

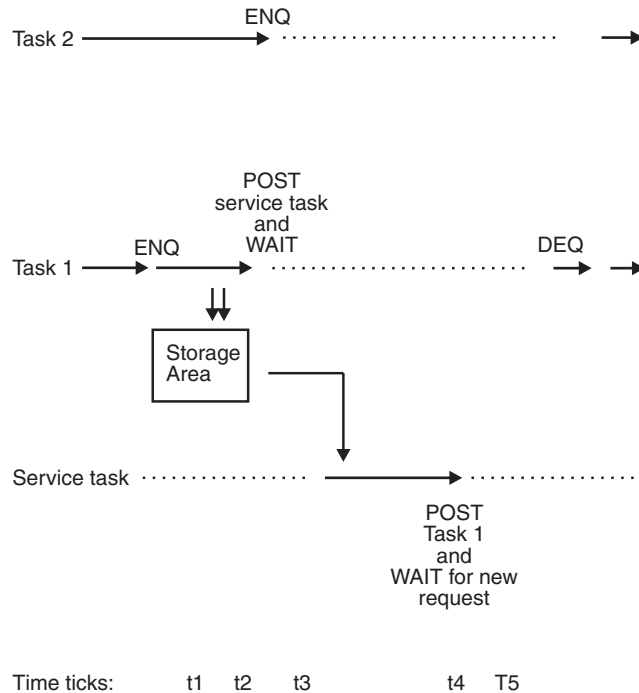


Figure 19. Synchronized use of a common service task

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, 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

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

When a program is started by MVS, it is executing as the main task of the address space in which it was started. In the examples in this chapter, 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 of the following:

- 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 45), 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

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

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 45 shows the basic logic in a multitasking concurrent server.

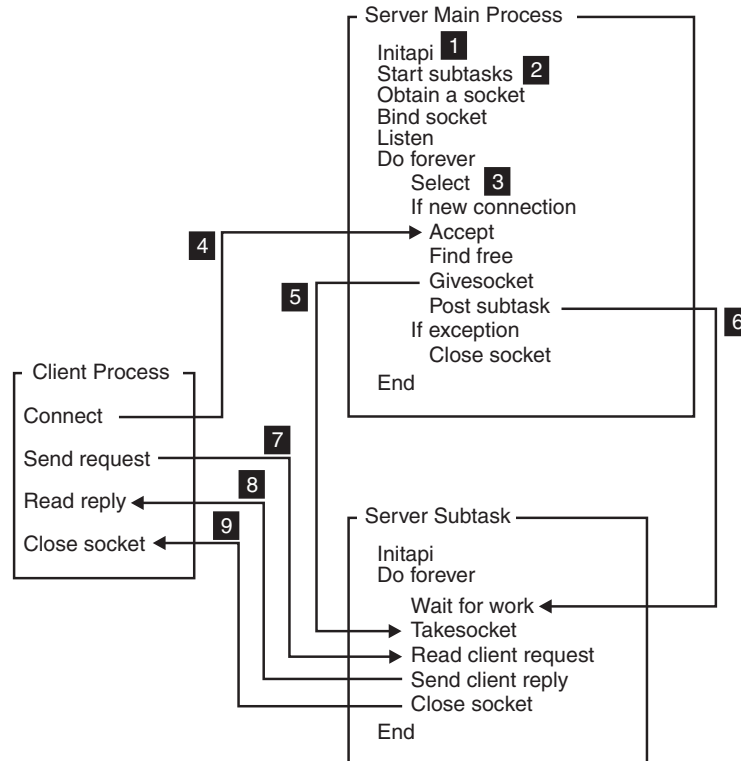


Figure 20. Concurrent server in an MVS address space

Selecting requests

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:

$\text{INT}(\text{highest socket descriptor} / 32) + 1$

Table 4 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.

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

To set and test these bits in an another way, use the following assembler macro, found in file *hlq.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
LH    15,&SD             Get Socket Descriptor
AGO   .SDOK
.SDREG ANOP
LR    15,&SD             Get Socket Descriptor

```

Figure 21. To set/test bits for SELECT calls (Part 1 of 2)

```

.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 (Part 2 of 2)

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
    ERETMASK=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.

```
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 177.

Client connection requests

As shown in Figure 20 on page 45, the listener socket is selected with a pending read. Then, a new connection request arrives, and the following socket() call must accept.

Figure 23 illustrates this type of connection request.

```

*-----*
* ACCEPT the connection from a client                                     *
*-----*
EZASMI  TYPE=ACCEPT,  *Accept new connection                          C
        S=TPIMSNO,    *On listener socket descriptor                  C
        NAME=SOCSTRUC, *Returned client socket structure              C
        ERRNO=ERRNO,   *Error number from EZASMI                     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
*
TPIMSNO  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

This section contains concepts and tasks information about passing sockets.

Common interface concepts

To help you better understand socket passing, the following sections 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 sections 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 already 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 765 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 *hlq.SEZAINST* partitioned data set. The member names are:

- MTCSRVR
- MTCCSUB
- MTCCCLNT

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 54:

```

*-----*
*Give socket to subtask
*-----*
MVC CLNNAME,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
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
CLNNAME 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 45, 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 55 illustrates this process.


```

*-----*
* Take socket from main process                                     *
*-----*
EZASMI TYPE=TAKESOCKET,      *Takesocket                               C
      CLIENT=TPIMCLNI,      *Main task client id structure          C
      SOCRECV=TPISSOD,      *Main task socket descriptor          C
      ERRNO=ERRNO,          C
      RET CODE=RETCODE,      C
      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, 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

See Chapter 7, “Transferring data between sockets,” on page 63.

Closing a concurrent server program

See Chapter 3, “Designing an iterative server program,” on page 27.

Chapter 5. Designing a client program

This chapter explains how to design a client program. All examples within this chapter 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).

- “Allocating a socket”
- “Connecting to a server”
- “Transferring data between sockets” on page 59.
- “Closing a client program” on page 59.

Allocating a socket

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” on page 27.

Connecting to a server

To connect to a server, the client must know the server name. This section 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 prior to 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 58:

```

/*-----*/
/* Find IP addresses of server host */
/*-----*/
servipaddr = DoSocket('Gethostbyname', tpi_server)
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

```

When you issue a connect call to an IP address currently not available, your connect call eventually 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

```

```
        end
    end
    i = i + 1
end
if connected then do
    say 'Connect failed, rc='sockrc
    exit(sockrc)
end
```

Transferring data between sockets

See Chapter 7, “Transferring data between sockets,” on page 63.

Closing a client program

See Chapter 3, “Designing an iterative server program,” on page 27.

Chapter 6. Designing a program to use datagram sockets

This chapter explains how to design a program to use datagram sockets. All examples within this chapter 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). Topics include:

- Datagram socket characteristics
- Understanding datagram socket program structure
- Allocating a socket
- Binding sockets to port numbers
- Streamline data transfer using connect call
- Transferring data between sockets

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.

Because implementation of such function is complicated, it is recommend that you 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 27.

Allocating a socket

See “Allocating sockets” on page 27.

Binding 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 sockets

See Chapter 7, “Transferring data between sockets,” on page 63.

Chapter 7. Transferring data between sockets

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

- Overview
- Streams and messages
- Data representation

AF_INET6 (IPv6) sockets can communicate with AF_INET (IPv4) sockets using mapped addresses. Refer to *z/OS Communications Server: IPv6 Network and Application Design Guide* for details.
- Using send() and recv() calls
- Using sendto() and recvfrom() calls

Overview

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 the *z/OS Communications Server: New Function Summary* for additional information.

Streams and messages

This section 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:

```
*-----*
* 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 binary halfword with the value encoded in network byte order, but you can implement it as a text field, as shown in Figure 28.

```
*-----*
* 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 shows the TCP buffer flush technique.

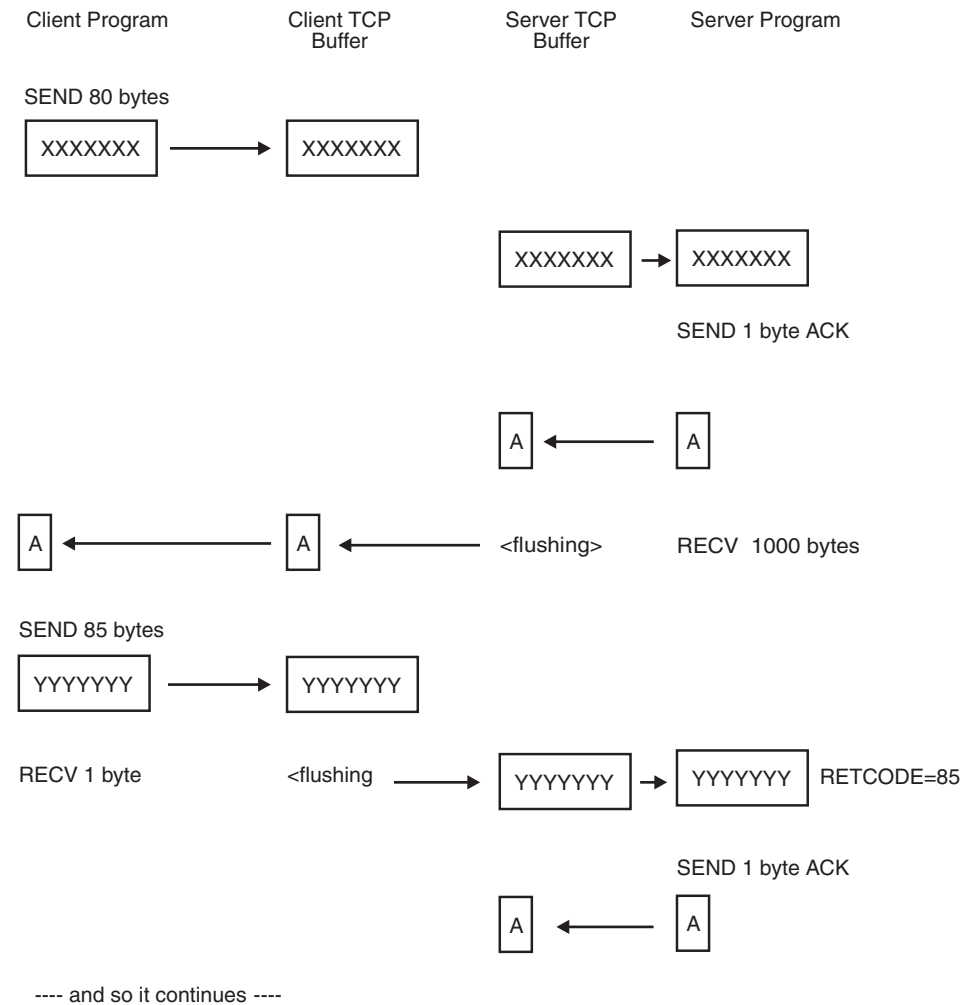


Figure 29. The TCP buffer flush technique

In Figure 29, 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
        recv-flag-peek
        recv-peek-len
        recv-peek-buffer
        errno
        retcode.
    if retcode < 0 then
        - process error -
    if retcode = 0 then
        - process client closed socket -
    if not TPI-identifier then
        - translate recv-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.

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.

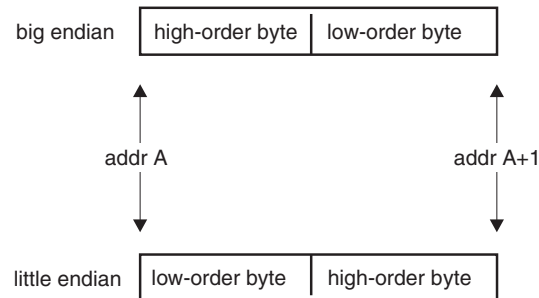


Figure 30. Big or little endian byte order for a 2-byte integer

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 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 only apply translation between ASCII and EBCDIC to the text fields in your message.

If you use binary integer fields in your messages, it is recommended that you 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 binary 2- and 4-byte 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 section provides information about sending and receiving calls.

The 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 illustrated here apply to all the APIs in this book.

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 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 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:

- “`read()`” on page 168
- “`readv()`” on page 169
- “`recv()`” on page 171
- “`send()`” on page 183
- “`write()`” on page 209
- “`writv()`” on page 210

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

read and write

These calls can only be used 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 'EZASOCKET' 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 'EZASOKET' 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 72:

```

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 173, “sendmsg()” on page 185, and “sendto()” on page 187 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 documents contain details on how to enable an IPv6 application:

- *z/OS Communications Server: IPv6 Network and Application 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 “Socket libraries” on page 6 for information on which APIs support IPv6.
- See Appendix C, “Address family cross reference,” on page 797 for information on 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 chapter 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 section 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 section.

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.

IPv4 multicast options

This section describes IPv4 multicast support of the following socket options:

- `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` and `IP_DROP_MEMBERSHIP` options are exceptions, as they are `SETSOCKOPT` options only.

`IP_ADD_MEMBERSHIP` and `IP_DROP_MEMBERSHIP`

Use `IP_ADD_MEMBERSHIP` to set the IPv4 multicast address and the local IPv4 interface address. This is accomplished by using the `SETSOCKOPT` API and specifying the address of the `IP_MREQ` structure containing these addresses. 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 socket. The stack chooses a default multicast interface if an interface of 0 is passed. The format of the `IP_MREQ` structure can be found in the `BPXYSOCK` macro. The assembler program example in Figure 33 illustrates this socket option using the `EZASMI`:

```

*****
*
*           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

```

Figure 33. IP_ADD_MEMBERSHIP and IP_DROP_MEMBERSHIP (Part 1 of 3)


```

*****
*
*      Issue BIND socket
*
*****
MVC  PORT(2),PORTS      Load port #
MVC  ADDRESS(4),ADDR    Load IP address
EZASMI TYPE=BIND,      ISSUE BIND MACRO
      S=S,              DATAGRAM
      NAME=NAME,        SOCKET ADDRESS STRUCTURE
      RETCODE=RETCODE,  (SPECIFY RETCODE FIELD)
      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

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

```

Figure 33. IP_ADD_MEMBERSHIP and IP_DROP_MEMBERSHIP (Part 2 of 3)

```

*****
*
*      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
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'
*
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
*
      CNOP 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 and interface
*
MY_MULTICAST_ADDRESS DC AL1(224),AL1(9),AL1(9),AL1(9)
                        Multicast address
MY_MULTICAST_INTERFACE DC AL1(204),AL1(59),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-BPXYS
IMR_MULTIADDR DS CL4          IP MULTICAST ADDR OF GROUP 01-BPXYS
IMR_INTERFACE DS CL4          LOCAL IP ADDR OF INTERFACE 01-BPXYS

```

Figure 33. IP_ADD_MEMBERSHIP and IP_DROP_MEMBERSHIP (Part 3 of 3)

In order to remove the host from the multicast host group you must issue a `SETSOCKOPT` with the `IP_DROP_MEMBERSHIP` option. This call is similar to the `IP_ADD_MEMBERSHIP` as it also uses the `IP_MREQ` structure to declare the IPv4 multicast address and the local IPv4 address interface. See also Figure 33 on page 78.

While the application is a member of the multicast host group, datagrams may be sent or received as required. By issuing the `NETSTAT DEVLINKS` command you may see the status of the interface.

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 34 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 34. `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 35 illustrates the use of IP_MULTICAST_LOOP 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
```

Figure 35. IP_MULTICAST_LOOP (Part 1 of 2)

```
*****
*
*      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 35. IP_MULTICAST_LOOP (Part 2 of 2)

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 36 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 36. IP_MULTICAST_TTL

IPv6 multicast options

To enable your application to support the IPv6 flavor 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.

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 37 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

```

Figure 37. IPV6_JOIN_GROUP and IPV6_LEAVE_GROUP (Part 1 of 3)

```

*****
*
*      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,RCCHECK      --> 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,RCCHECK      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,RCCHECK      --> CHECK IT

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

```

Figure 37. IPV6_JOIN_GROUP and IPV6_LEAVE_GROUP (Part 2 of 3)


```

*****
*
*      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'FF020101010101050505050505' IP ADDR TO BIND
PORTS      DC   H'11007'     PORT TO BIND
*
* My Multicast address and interface
*
MY_MULTICAST_ADDRESS DC XL16'FF020101010101050505050505' 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 37. IPV6_JOIN_GROUP and IPV6_LEAVE_GROUP (Part 3 of 3)

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 37 on page 85.

While the application is a member of the multicast host group, datagrams may be sent or received as required. By issuing the NETSTAT DEVLINKS command you may see the status of the interface.

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 38 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 38. 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 39 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=OPTVALON,    OPTION VALUE              X
      RETCODE=RETCODE,    (SPECIFY RETCODE FIELD)    X
      ERROR=ERROR        ABEND IF MACRO ERROR
BAL    R14,RCCHECK      --> 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,RCCHECK      --> CHECK IT
```

Figure 39. IPV6_MULTICAST_LOOP (Part 1 of 2)

```
*****
*
*      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=OPTVALOFF,  OPTION VALUE              X
      RETCODE=RETCODE,    (SPECIFY RETCODE FIELD)    X
      ERROR=ERROR        ABEND IF MACRO ERROR
BAL    R14,RCCHECK      --> 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,RCCHECK      --> CHECK IT
```

Figure 39. IPV6_MULTICAST_LOOP (Part 2 of 2)

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 40 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 40. IPV6_MULTICAST_HOPS

Part 3. Application program interfaces

Chapter 10. C Socket application programming interface (API)

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, refer to the *z/OS C/C++ Run-Time Library Reference*.

This chapter 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 C/C++ Programming Guide*.

Compiler restrictions

This section tells you how to move your application to the z/OS CS 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 CS.
- 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 CS.

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 section on 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 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 (refer to the *z/OS C/C++ Run-Time 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 CS 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 section 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.

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.

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=hlq.SEZACMAC,DISP=SHR
```

Note: *hlq* is the high-level qualifier of your TCP/IP system libraries.

- 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=hlq.SEZACMTX,DISP=SHR
```

Figure 41 on page 98 shows a sample JCL to be used when compiling non-reentrant modules. Modify the lines to conform to the naming conventions of your site:

```

//COMMIT 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,
// CPARAM1=NORENT,
// CPARAM2='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=CCNDRVR,PARM=('&CPARM1','&CPARM2'),
// REGION=&CREGSIZ
//STEPLIB DD DSN=&LIBPRFX1..SCEERUN,DISP=SHR
// DD DSN=&COMPRFX..SCBCCMP,DISP=SHR
//SYMSGSGS DD DUMMY,DSN=&COMPRFX..SCBC3MSG(EDCMSGE),DISP=SHR
//SYSIN DD DSN=USER71.TEST.SRC(&INFILE),DISP=SHR
//USERLIB DD DSN=USER.MYPROG.H,DISP=SHR
//SYSLIB DD DSN=TCP.SEZACMAC,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 41. Sample JCL for compiling non-reentrant modules.

Figure 42 on page 99 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
//SYSLIB DD DSN=TCP.SEZACMTX,DISP=SHR
// DD DSN=CEE.OSV2R7.SCEELKED,DISP=SHR
// PEND
// EXEC EDCL,MEM=CTEST

```

Figure 42. Sample JCL for linking non-reentrant modules.

Figure 43 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 43. Sample JCL for running non-reentrant modules.

Note: For more information about compiling and linking, see *z/OS C/C++ Compiler and Run-Time Migration Guide for the Application Programmer*.

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=h1q.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=h1q.SEZARNT1,DISP=SHR
```

Guideline: The system administrator should have followed the instructions for program reentrancy in the section of the *z/OS C/C++ Programming Guide* 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=h1q.SEZACMTX,DISP=SHR
```

Notes:

1. If LE libraries are concatenated ahead of SEZACMTX, socket errors can occur because the link-edit uses the LE OS/390 UNIX socket library, not the TCP/IP library.
2. For more information about compiling and linking, see *z/OS C/C++ Compiler and Run-Time Migration Guide for the Application Programmer*.

Figure 44 on page 101 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,SO,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
//SYSMSGSD D DUMMY,DSN=&COMPRFX..SCBC3MSG(EDCMSGE),DISP=SHR
//SYSIN DD DSN=USER71.TEST.SRC(&INFILE),DISP=SHR
//USERLIB DD DSN=USER.MYPROG.H,DISP=SHR
//SYSLIB DD DSN=TCP.SEZACMAC,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 44. Sample JCL for compiling reentrant modules

Figure 45 on page 102 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 C/MVS,
//* OS390 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.
//* - TCPV34.SEZARNT1 IS THE REENTRANT C SOCKET LIBRARY.
//* IT IS USED ON THE PRE-LINK STEP.
//* - TCPV34.SEZACMTX IS THE GENERIC SOCKET LIBRARY.
//* IT IS USED ON THE LINK STEP TO RESOLVE OTHER C SOCKET
//* MODULES THAT DO NOT EXIST IN TCPV34.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='CEEL.OSV2R7',
// LIBPRFX2='CEE.OSV2R7'
//*-----
//* PRE-LINKEDIT STEP:
//*-----
//PLKED EXEC PGM=EDCPRLK,
// REGION=2048K,PARM='UPCASE'
//STEPLIB DD DSN=LIBPRFX1..SCEERUN,DISP=SHR
//SYSMSGSGS DD DSN=LIBPRFX2..SCEMSGP(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=LIBPRFX2..SCEELKED,DISP=SHR
// DD DSN=TCP.SEZACMTX,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSN=PLKED.SYSMOD,DISP=(OLD,DELETE)
//SYSMOD 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 45. Sample JCL for prelinking and linking reentrant modules

Figure 46 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 46. Sample JCL for running the reentrant program

Compiler messages

z/OS CS 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. Table 6 identifies some of the errors you might find.

Table 6. C/C++/390 R4 compiler messages

Message	Problem	Solution
CBC3022	Structure fields are not found.	Use #include <time.h>
CBC3050	Character assumed to be an integer. For example, if a function, <code>abc()</code> , actually returns a character, <code>*</code> , that is not declared, the compiler assumes the character to be an integer and finds a mismatch.	Be sure to declare the character explicitly. In our example, it would be <code>char *abc();</code>
CBC3221	The initializers used are not valid.	First define the array. Then initialize the array element assignments.
CBC3275	Compiler message indicates that <code>va_list</code> is not defined. This happens when <code>stdio.h</code> is included before <code>stdarg.h</code> .	Include <code>stdarg.h</code> before <code>stdio.h</code>
CBC3282	Function parameters must be prototyped.	Identify the variable type of all parameters.
CBC3296	Some header files are not in the search path.	You did not #include the necessary header files.
CBC3343	External function is used but not explicitly declared.	Declare the function as above.
CBC5034	Structure assignments are non-reentrant; module was compiled as reentrant.	Compile the module as NORENT (non-reentrant).

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. Do the following:

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 CS, 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 161 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 145 and “`setsockopt()`” on page 197 for details about the supported options.
- The IBM `fcntl()` call supports only a subset of the options available. See “`fcntl()`” on page 122 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 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*.

C socket header files

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

- `bsdtime.h`
- `bsdtocms.h`
- `bsdtypes.h`
- `fcntl.h`
- `if.h`
- `in.h`
- `inet.h`
- `ioctl.h`
- `manifest.h`
- `netdb.h`
- `rtrouteh.h`
- `socket.h`
- `tcpermo.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

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 CS, 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 7 shows the C structures used, and the corresponding assembler language syntax.

Table 7. C structures in assembler language format

C structure	Assembler language equivalent		
struct sockaddr_in { short sin_family; ushort sin_port; struct in_addr sin_addr; char sin_zero[8]; };	FAMILY	DS	H
	PORT	DS	H
	ADDR	DS	F
	ZERO	DC	XL8'00'
struct timeval { long tv_sec; long tv_usec; };	TVSEC	DS	F
	TVUSEC	DS	F
struct linger { int l_onoff; int l_linger; };	ONOFF	DS	F
	LINGER	DS	F
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; };	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
struct ifconf { int ifc_len; union { caddr_t ifcu_buf; struct ifreq *ifcu_req; } ifc_ifcu; };	IFCLEN	DS	F
	IGNORED	DS	F
struct clientid { int domain; char name[8]; char subtaskname[8]; char reserved[20]; };	DOMAIN	DS	F
	NAME	DS	CL8
	SUBTASK	DS	CL8
	RESERVED	DC	XL20'00'

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, “Return codes,” on page 781.

C socket calls

This section 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
<code>s</code>	The socket descriptor.
<code>addr</code>	The socket address of the connecting client that is filled by <code>accept()</code> before it returns. The format of <code>addr</code> is determined by the domain in which the client resides. <code>addr</code> is only filled in by <code>accept()</code> when both <code>addr</code> and <code>addrlen</code> are nonzero values.
<code>addrlen</code>	Must initially point to an integer that contains the size in bytes of the storage pointed to by <code>addr</code> . If <code>addr</code> is NULL, then <code>addrlen</code> 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 filled in. 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 <i>s</i> parameter is not a valid socket descriptor.
ENOBUFS	Indicates insufficient buffer space available to create the new socket.
EINVAL	The <i>s</i> parameter is not of type SOCK_STREAM.
EFAULT	Using <i>addr</i> and <i>addrlen</i> 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 <i>s</i> 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 wishes 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
<i>s</i>	Socket descriptor returned by a previous <code>socket()</code> call
<i>name</i>	Points to a <i>sockaddr</i> structure containing the name to be bound to <i>s</i>
<i>namelen</i>	Size of <i>name</i> in bytes

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 recommended 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

EBADF

The *s* parameter is not a valid socket descriptor.

EADDRNOTAVAIL

The address specified is not valid on this host. For example, the internet address does not specify a valid network interface.

EFAULT

The name or namelen parameter specified an address outside of the caller address space.

EAFNOSUPPORT

The address family is not supported (it is not AF_INET).

EADDRINUSE

The address is already in use. See the SO_REUSEADDR option described under “getsockopt()” on page 145 and the SO_REUSEADDR option described under the “setsockopt()” on page 197 for more information.

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.

Example

Following are examples of 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 115 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
<i>s</i>	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 <i>s</i> 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()`, `writv()`, `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
<code>s</code>	Socket descriptor
<code>name</code>	Points to a <i>socket address</i> structure containing the address of the socket to which connection will be attempted
<code>namelen</code>	Size of the <i>socket address</i> , in bytes, pointed to by <code>name</code>

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
EADDRNOTAVAIL	Calling host cannot reach the specified destination.
EAFNOSUPPORT	Address family is not supported.
EALREADY	Socket descriptor <i>s</i> is marked nonblocking, and a previous connection attempt is incomplete.
EBADF	The <i>s</i> parameter is not a valid socket descriptor.
ECONNREFUSED	The connection request was rejected by the destination host.
EFAULT	The <i>name</i> or <i>namelen</i> 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 110.

```
int s;
struct sockaddr_in servername;
struct sockaddr_iucv mvssservername;
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(&mvssservername, 0, sizeof(mvssservername));
mvssservername.siucv_family = AF_IUCV;
strncpy(mvssservername.siucv_nodeid, "          ", 8);
/* The field is 8 positions padded to the right with blanks */
strncpy(mvssservername.siucv_userid, "MVSUSER1 ", 8);
strncpy(mvssservername.siucv_name, "APPL      ", 8);
:
:
rc = connect(s, (struct sockaddr *) &mvssservername, sizeof(mvssservername));
```

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. Refer to *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. Refer to *z/OS Communications Server: IP Configuration Guide* for information on 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 *hlq*.ETC.SERVICES data set.

The *hlq*.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
<i>s</i>	Socket descriptor
<i>cmd</i>	Command to perform
<i>arg</i>	Data associated with <i>cmd</i>

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 are valid `fcntl()` commands:

Command	Description
F_SETFL	Sets the status flags of socket descriptor <i>s</i> . (One flag, <code>FNDELAY</code> , can be set.)
F_GETFL	Returns the status flags of socket descriptor <i>s</i> . (One flag, <code>FNDELAY</code> , can be queried.)
	The <code>FNDELAY</code> flag marks <i>s</i> as being in nonblocking mode. If data is not present on calls that can block [<code>read()</code> , <code>readv()</code> , and <code>recv()</code>] the call returns with -1, and <code>errno</code> is set to <code>EWOULDBLOCK</code> .

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 <i>s</i> parameter is not a valid socket descriptor.
EINVAL	The <i>arg</i> 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
<i>domain</i>	The value in <i>domain</i> must be AF_INET.
<i>clientid</i>	Points to a <i>clientid</i> 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. Refer to *z/OS Communications Server: IP Configuration Guide* for information on 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
<i>addr</i>	Points to an unsigned long value containing the address of the host
<i>addrlen</i>	Size of <i>addr</i> in bytes
<i>domain</i>	Address domain supported (<code>AF_INET</code>)

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
<i>h_name</i>	The address of the official name of the host
<i>h_aliases</i>	A pointer to a zero-terminated list of addresses pointing to alternate names for the host
<i>h_addrtype</i>	The type of host address returned; currently, always set to <code>AF_INET</code>
<i>h_length</i>	The length of the host address, in bytes
<i>h_addr</i>	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 `tcprerror()` call cannot be validated. The global variable `h_errno` identifies the specific error.

<code>h_errno</code>	Description
<code>HOST_NOT_FOUND</code>	The name specified is unknown, the address domain specified is not supported, or the address length specified is not valid.
<code>TRY_AGAIN</code>	Temporary error; information not currently accessible.
<code>NO_RECOVERY</code>	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. Refer to *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
<i>name</i>	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
<i>h_name</i>	The address of the official name of the host
<i>h_aliases</i>	A pointer to a zero-terminated list of addresses pointing to alternate names for the host
<i>h_addrtype</i>	The type of host address returned; currently, set to <code>AF_INET</code>
<i>h_length</i>	The length of the host address in bytes
<i>h_addr</i>	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
<code>HOST_NOT_FOUND</code>	The name specified is unknown.
<code>TRY_AGAIN</code>	Temporary error; information not currently accessible.
<code>NO_RECOVERY</code>	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. Refer to *z/OS Communications Server: IP Configuration Guide* for information on 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
<i>h_name</i>	The address of the official name of the host
<i>h_aliases</i>	A pointer to a zero-terminated list of addresses pointing to alternate names for the host
<i>h_addrtype</i>	The type of host address returned; currently set to <code>AF_INET</code>
<i>h_length</i>	The length of the host address, in bytes
<i>h_addr</i>	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 TCPIP stack learned at startup from the TCPIP.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
<i>name</i>	Character array to be filled with the host name
<i>namelen</i>	Length of <i>name</i> ; 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 <i>name</i> 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 *z/OS Communications Server: IP Configuration Reference*.

For detailed information about this method, see *z/OS Communications Server: New Function Summary*.

```
#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
<i>cmd</i>	The command to perform. For TCP/IP V3R2 for MVS, <code>IBMTCP_IMAGE</code> is supported.
<i>buf</i>	Points to the structure filled in by the call.

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
<code>X'8xxx'</code>	Active
<code>X'4xxx'</code>	Terminating
<code>X'2xxx'</code>	Down
<code>X'1xxx'</code>	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.

I The version field for z/OS V1R6 is X'0616'.

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 <i>name</i> 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
<i>s</i>	The socket descriptor
<i>level</i>	The level for which the option is set
<i>optname</i>	The name of a specified socket option
<i>optval</i>	Points to option data
<i>optlen</i>	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 <i>s</i> parameter is not a valid socket descriptor.
EFAULT	Using <i>optval</i> and <i>optlen</i> parameters would result in an attempt to access storage outside the caller address space.
EINVAL	This is returned when <i>optlen</i> 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. Refer to *z/OS Communications Server: IP Configuration Guide* for information on using local host tables.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <netdb.h>
struct netent *getnetbyaddr(unsigned long net, int type)
```

Parameter	Description
<i>net</i>	The network address
<i>type</i>	The address domain supported (<code>AF_INET</code>)

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 <code>AF_INET</code> .
<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. Refer to the *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 <code>AF_INET</code> .
<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. Refer to *z/OS Communications Server: IP Configuration Guide* for information on 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
<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 <code>AF_INET</code> .
<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.

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` are ignored, but are 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
<code>s</code>	The socket descriptor.
<code>name</code>	Points to a structure containing the internet address of the connected socket that is filled in by <code>getpeername()</code> before it returns. The exact format of <code>name</code> is determined by the domain in which communication occurs.
<code>namelen</code>	Points to a fullword containing the size of the address structure pointed to by <code>name</code> in bytes.

Return Values

The value 0 indicates success; the value -1 indicates an error. `Errno` identifies the specific error.

Errno	Description
<code>EBADF</code>	The <code>s</code> parameter is not a valid socket descriptor.
<code>EFAULT</code>	Using the <code>name</code> and <code>namelen</code> parameters as specified would result in an attempt to access storage outside of the caller address space.
<code>ENOTCONN</code>	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
<i>name</i>	Points to the specified protocol.

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()`, `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
<i>proto</i>	Protocol number

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()`, `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
<i>s_proto</i>	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 `tcprerror()` call cannot be validated.

Related Calls

`endservant()`, `getservbyport()`, `getservent()`, `setservent()`

getservbyport()

The getservbyport() call searches the *hlq.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
<i>port</i>	Port number
<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
<i>s_proto</i>	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
<code>s</code>	The socket descriptor.
<code>name</code>	The address of the buffer into which <code>getsockname()</code> copies the name of <code>s</code> .
<code>namelen</code>	Points to a fullword containing the size of the address structure pointed to by <code>name</code> in bytes.

Return Values

A value of 0 indicates success; the value -1 indicates an error. `Errno` identifies the specific error.

Errno Value	Description
<code>EBADF</code>	The <code>s</code> parameter is not a valid socket descriptor.
<code>EFAULT</code>	Using the <code>name</code> and <code>namelen</code> 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
<i>s</i>	The socket descriptor.
<i>level</i>	The level to which the option is set.
<i>optname</i>	The name of a specified socket option. Refer to Appendix D, “GETSOCKOPT/SETSOCKOPT command values” for the numeric values of <i>optname</i> .
<i>optval</i>	Points to option data.
<i>optlen</i>	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	<p>Returns the status of Nagle algorithm (RFC 896). This option is not supported for AF_IUCV sockets.</p> <p>When <i>optval</i> 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.</p> <p>When <i>optval</i> is nonzero, Nagle algorithm is disabled and TCP will send small packets of data even before the acknowledgment for previous data sent is received.</p>
--------------------	---

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	<p>Toggles the ability to broadcast messages. The default is <i>disabled</i>. When this option is enabled, it allows the application to send broadcast messages over <i>s</i> when the interface specified in the destination supports the broadcasting of packets. This option has no meaning for stream sockets.</p>
---------------------	--

SO_CLUSTERCONNTYPE	<p>Returns a bit mapped 32-bit value. One or more than one of the following will be returned:</p> <ul style="list-style-type: none"> • 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). <p>An internal indicator means that for this side of the connection, the link/interface type is one of the following:</p> <ul style="list-style-type: none"> – CTC – HiperSockets (iQDIO) – MPCPTP (including XCF and IUTSAMEH connections) – 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 due to 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 due to 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 an 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 keepalive mechanism for a stream socket. The default is *disabled*. When activated, the keepalive 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_OOBINLINE

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 <i>s</i> parameter is not a valid socket descriptor.
EFAULT	Using <i>optval</i> and <i>optlen</i> parameters would result in an attempt to access storage outside the caller address space.
EINVAL	The <i>optname</i> parameter is unrecognized, or the <i>level</i> parameter is not SOL_SOCKET.

Example

The following are examples of the `getsockopt()` call. See “`setsockopt()`” on page 197 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 slave 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
<i>d</i>	The descriptor of a socket to be given to another application.
<i>clientid</i>	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 filled in as follows:

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

<i>domain</i>	This call is supported only in the AF_INET domain.
<i>name</i>	The slave program address space name, left-justified and padded with blanks. The slave 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.
<i>subtaskname</i>	Specifies blanks.
<i>reserved</i>	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 slave program. One way to pass a socket is by passing the slave program startup parameter list.

The slave program calls takesocket(), specifying the master program client ID and socket descriptor.

Waiting for the slave 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 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 <i>d</i> 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 <i>clientid</i> parameter as specified would result in an attempt to access storage outside the caller address space.
EINVAL	The <i>clientid</i> 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
<i>a</i>	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
<i>a</i>	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
<i>cp</i>	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.

Negative 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
<i>in</i>	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
<i>net</i>	The network number
<i>lna</i>	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
<i>in</i>	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
<i>cp</i>	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
<i>in</i>	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.

This call can be used only in the `AF_INET` domain.

```
#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
<i>s</i>	The socket descriptor.
<i>cmd</i>	The command to perform.
<i>arg</i>	Points to the data associated with <i>cmd</i> .

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 are valid `ioctl()` keywords:

Keyword	Description
FIONBIO	Sets or clears nonblocking I/O for a socket. The variable <i>arg</i> 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 <i>arg</i> points to an integer.
SIOCADDRT	Adds a routing table entry. The variable <i>arg</i> points to a <i>rtentry</i> structure, as defined in <code>RTRROUTE.H</code> . 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 <i>arg</i> 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.
SIOCDELRT	Deletes a routing table entry. The variable <i>arg</i> points to a <i>rtentry</i> structure, as defined in <code>RTRROUTE.H</code> . 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 <i>arg</i> points to an <i>ifreq</i> structure, as defined in <code>IF.H</code> . The interface address is returned in the argument.
SIOCGIFBRDADDR	Gets the network interface broadcast address. The variable <i>arg</i> points to an <i>ifreq</i> structure, as defined in <code>IF.H</code> . 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.

SIOCSIFMETRIC

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 *hlq*.SEZANMAC.

Note: The ARP counter data provided differs depending on the type of device. Refer to the *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 <i>s</i> parameter is not a valid socket descriptor.
EINVAL	The request is not valid, or not supported.
EFAULT	The <i>arg</i> 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. Once 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
<i>s</i>	Socket descriptor
<i>backlog</i>	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
<code>EBADF</code>	The <i>s</i> parameter is not a valid socket descriptor.
<code>EOPNOTSUPP</code>	The <i>s</i> parameter is not a socket descriptor that supports the <code>listen()</code> 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
<i>totdesc</i>	Points to an integer containing a value one greater than the largest socket number desired. The maximum allowed value is 2000. Note: If a <i>totdesc</i> value greater than 2000 is specified, the internal value is set to 2000. In all cases, use <code>getdtablesize()</code> to verify the value set by <code>maxdesc()</code> .
<i>inetdesc</i>	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 <i>totdesc</i> or <i>inetdesc</i> 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 <code>maxdesc()</code> after creating a socket, or after a previous call to <code>maxdesc()</code> .
EINVAL	Indicates that <i>*totdesc</i> is less than <i>*inetdesc</i> ; <i>*totdesc</i> is less than or equal to 0; or <i>*inetdesc</i> 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
<i>a</i>	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
<i>a</i>	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 161, or “`fcntl()`” on page 122 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
<code>s</code>	Socket descriptor.
<code>buf</code>	Points to the buffer that receives the data.
<code>len</code>	Length in bytes of the buffer pointed to by <code>buf</code> .

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 <code>s</code> is not a valid socket descriptor.
EFAULT	Using the <code>buf</code> and <code>len</code> 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()`, `writv()`

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
<i>s</i>	The socket descriptor.
<i>iov</i>	Points to an iovec structure.
<i>iovcnt</i>	The number of buffers pointed to by the <i>iov</i> 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
<i>iov_base</i>	Points to the buffer.
<i>iov_len</i>	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 122 or “`ioctl()`” on page 161 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 <i>s</i> is not a valid socket descriptor.
EFAULT	Using <i>iov</i> and <i>iovcnt</i> would result in an attempt to access storage outside the caller address space.
EINVAL	<i>iovcnt</i> was not valid, or one of the fields in the <i>iov</i> array was not valid. Also returned for a NULL <i>iov</i> pointer.
EWouldBlock	Indicates that <i>s</i> is in nonblocking mode and data is not available to read.

Related Calls

`connect()`, `fcntl()`, `getsockopt()`, `ioctl()`, `read()`, `recv()`, `recvmsg()`, `recvfrom()`, `select()`, `selectx()`, `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 122 or “`ioctl()`” on page 161 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
<i>s</i>	Socket descriptor.
<i>buf</i>	Points to the buffer that receives the data.
<i>len</i>	Length in bytes of the buffer pointed to by <i>buf</i> .
<i>flags</i>	Set the <i>flags</i> parameter by specifying one or more of the following flags. If more than one flag is specified, the logical OR operator (<code> </code>) must be used to separate them. Setting this parameter is supported only for sockets in the <code>AF_INET</code> domain. Setting these flags is not supported for sockets in the <code>AF_IUCV</code> 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
<code>EBADF</code>	Indicates that <i>s</i> is not a valid socket descriptor.
<code>EFAULT</code>	Using the <i>buf</i> and <i>len</i> 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()`, `writv()`

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 122 or “`ioctl()`” on page 161 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
-----------	-------------

<i>s</i>	Socket descriptor.
----------	--------------------

<i>buf</i>	Pointer to the buffer to receive the data.
------------	--

<i>len</i>	Length in bytes of the buffer pointed to by <i>buf</i> .
------------	--

<i>flags</i>	A parameter that can be set to 0 or <code>MSG_PEEK</code> , or <code>MSG_OOB</code> . Setting this parameter is supported only for sockets in the <code>AF_INET</code> domain. Setting these flags is not supported for sockets in the <code>AF_IUCV</code> 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.

<i>name</i>	Points to a <i>socket address</i> structure from which data is received. If <i>name</i> is a nonzero value, the source address is returned (datagram sockets).
<i>namelen</i>	Points to the size of <i>name</i> 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 <i>s</i> is not a valid socket descriptor.
EFAULT	Using the <i>buf</i> and <i>len</i> parameters would result in an attempt to access storage outside the caller address space.
EWOULDBLOCK	Indicates that <i>s</i> 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 <code>getsockopt()</code> using <code>SO_SNDBUF</code> for the socket type, either TCP, UDP, or RAW.
EINVAL	Parameter <i>namelen</i> is not valid.

Related Calls

`fcntl()`, `getsockopt()`, `ioctl()`, `read()`, `readv()`, `recv()`, `recvmsg()`, `select()`, `selectex()`, `send()`, `sendmsg()`, `sendto()`, `setsockopt()`, `socket()`, `write()`, `writv()`

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
<i>s</i>	Socket descriptor.
<i>msg</i>	Points to an <code>msghdr</code> structure.
<i>flags</i>	Set the <i>flags</i> parameter by specifying one or more of the following flags. If more than one flag is specified, the logical OR operator (<code> </code>) must be used to separate them. Setting this parameter is supported only for sockets in the <code>AF_INET</code> domain. Setting these flags is not supported for sockets in the <code>AF_IUCV</code> 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
<i>msg_name</i>	An optional pointer to a buffer where the sender address is stored for datagram sockets.
<i>msg_namelen</i>	The size of the address buffer.
<i>msg_iov</i>	An array of <code>iovec</code> buffers into which the message is placed. An <code>iovec</code> buffer contains the following variables:
<i>iov_base</i>	Points to the buffer.
<i>iov_len</i>	The length of the buffer.
<i>msg_iovlen</i>	The number of elements in the <code>msg_iov</code> array.
<i>msg_accrights</i>	The access rights received. This field is ignored.
<i>msg_accrightslen</i>	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 122 or “`ioctl()`” on page 161 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 <i>s</i> is not a valid socket descriptor.
EFAULT	Using <i>msg</i> would result in an attempt to access storage outside the caller address space. Also returned when <i>msg_namelen</i> is not valid.
EWOULDBLOCK	Indicates that <i>s</i> 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 <code>getsockopt()</code> using <code>SO_SNDBUF</code> 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()`, `writew()`

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 nfd, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout)
```

Parameter	Description
<i>nfd</i>	<p>The number of socket descriptors to be checked. This value should be one greater than the greatest number of sockets to be checked.</p> <p>You can use the <code>select()</code> 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 <i>nfd</i> parameter to force <code>select()</code> to check only a subset of the allocated socket bit set.</p> <p>If your application allocates sockets 3, 4, 5, 6, and 7, and you want to check all of your allocations, <i>nfd</i> should be set to 8, the highest socket descriptor you specified, plus 1. If your application checks sockets 3 and 4, <i>nfd</i> should be set to 5.</p> <p>Socket numbers are assigned starting with number 3 because numbers 0, 1, and 2 are used by the C socket interface.</p>
<i>readfds</i>	Points to a bit set of descriptors to check for reading.
<i>writefds</i>	Points to a bit set of descriptors to check for writing.
<i>exceptfds</i>	Points to a bit set of descriptors to check for exception conditions pending.
<i>timeout</i>	Points to the time to wait for <code>select()</code> 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 *nfd* to be a NULL pointer.

To completely 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(&<i>fdset</i>)	Sets all bits in bit set <i>fdset</i> 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 <code>FD_SET()</code> to set a socket as a member.
FD_SET(sock, &<i>fdset</i>)	Sets the bit for the socket <i>sock</i> to 1, making <i>sock</i> a member of bit set <i>fdset</i> .
FD_CLR(sock, &<i>fdset</i>)	Clears the bit for the socket <i>sock</i> in bit set <i>fdset</i> . This operation sets the appropriate bit to 0.
FD_ISSET(sock, &<i>fdset</i>)	Returns greater than 0 if <i>sock</i> is a member of the bit set <i>fdset</i> . Returns 0 if <i>sock</i> is not a member of <i>fdset</i> . (This operation returns the bit representing <i>sock</i> .)

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 *writelfds* 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 *writelfds* 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. [<code>FD_ZERO()</code> 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 <code>timeval</code> structure is not valid.

Note: If the number of ready sockets is greater than 65 535, only 65 535 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 177 for more information about `selectex()`.

```
#include <manifest.h>
#include <socket.h>
#include <bsdtypes.h>
#include <bsdtime.h>
```

```
int selectex(int nfd, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout, int *ecbptr)
```

Parameter	Description
<i>nfd</i>	The number of socket descriptors to be checked.
<i>readfds</i>	Points to a bit set of descriptors to be checked for reading.
<i>writefds</i>	Points to a bit set of descriptors to be checked for writing.
<i>exceptfds</i>	Points to a bit set of descriptors to be checked for exception pending conditions.
<i>timeout</i>	Points to the time to wait for <code>selectex()</code> to complete.
<i>ecbptr</i>	Points to the event control block (ECB) or ECB list. For an ECB list, the high-order bit must be turned on in <i>ecbptr</i> . 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 only supported 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 <i>timeval</i> structure is not valid.

Note: If the number of ready sockets is greater than 65 535, only 65 535 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 122 or “`ioctl()`” on page 161 to set nonblocking mode. See “`select()`” on page 177 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 wishes 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
<code>s</code>	Socket descriptor.
<code>msg</code>	Points to the buffer containing the message to transmit.
<code>len</code>	Length of the message pointed to by <code>msg</code> .
<code>flags</code>	Set the <code>flags</code> parameter by specifying one or more of the following flags. If more than one flag is specified, the logical OR operator (<code> </code>) must be used to separate them. Setting this parameter is supported only for sockets in the <code>AF_INET</code> domain. Setting these flags is not supported in the <code>AF_IUCV</code> 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
<code>EBADF</code>	Indicates that <code>s</code> is not a valid socket descriptor.
<code>EFAULT</code>	Using the <code>msg</code> and <code>len</code> parameters would result in an attempt to access storage outside the caller address space.
<code>ENOBUFS</code>	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()`, `writv()`

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
<code>s</code>	Socket descriptor.
<code>msg</code>	Points to an <code>msghdr</code> structure.
<code>flags</code>	Set the <code>flags</code> parameter by specifying one or more of the following flags. If more than one flag is specified, the logical OR operator (<code> </code>) must be used to separate them. Setting this parameter is supported only for sockets in the <code>AF_INET</code> domain. Setting these flags is not supported in the <code>AF_IUCV</code> domain.
	MSG_OOB Sends out-of-band data on the socket.
	MSG_DONTROUTE The <code>SO_DONTROUTE</code> 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
<code>msg_name</code>	The pointer to the buffer containing the recipient address. This is required for datagram sockets where an explicit <code>connect()</code> has not been done.
<code>msg_namelen</code>	The size of the address buffer. This is required for datagram sockets where an explicit <code>connect()</code> has not been done.
<code>msg_iov</code>	An array of <code>iovec</code> buffers containing the message. The <code>iovec</code> buffer contains the following: iov_base Points to the buffer. iov_len The length of the buffer.
<code>msg_iovlen</code>	The number of elements in the <code>msg_iov</code> array.
<code>msg_accrights</code>	The access rights sent. This field is ignored.
<code>msg_accrightslen</code>	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 122 or “`ioctl()`” on page 161 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 wishes 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 <i>s</i> is not a valid socket descriptor.
EFAULT	Using <i>msg</i> would result in an attempt to access storage outside the caller address space.
EINVAL	Indicates that <i>msg_namelen</i> 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 <i>s</i> 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()`, `writev()`

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 122 or “`ioctl()`” on page 161 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 wishes 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
-----------	-------------

<i>s</i>	Socket descriptor.
----------	--------------------

<i>msg</i>	Points to the buffer containing the message to be transmitted.
------------	--

<i>len</i>	Length of the message in the buffer pointed to by <i>msg</i> .
------------	--

<i>flags</i>	A parameter that can be set to 0 or <code>MSG_DONTROUTE</code> . Setting this parameter is supported only for sockets in the <code>AF_INET</code> domain. Setting these flags is not supported in the <code>AF_IUCV</code> domain.
--------------	--

MSG_DONTROUTE	
----------------------	--

The <code>SO_DONTROUTE</code> option is turned on for the duration of the operation. This is usually used by diagnostic or routing programs only.	
---	--

<i>to</i>	Address of the target.
-----------	------------------------

<i>tolen</i>	Size of the structure pointed to by <i>to</i> .
--------------	---

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 <i>s</i> 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()`, `writv()`

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. Refer to *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
<i>stayopen</i>	A nonzero flag value prevents the cached local host table contents from being freed after an <code>endhostent()</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

`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
<i>cmd</i>	The command to perform. For TCP/IP V3R2 for MVS, IBMTCP_IMAGE is supported.
<i>buf</i>	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 wishes 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 fill in *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
<i>s</i>	Socket descriptor.
<i>level</i>	Level for which the option is being set. Only SOL_SOCKET is supported.
<i>optname</i>	The name of a specified socket option.
<i>optval</i>	Points to option data.
<i>optlen</i>	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 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 <i>s</i> parameter is not a valid socket descriptor.
EFAULT	Using <i>optval</i> and <i>optlen</i> parameters would result in an attempt to access storage outside the caller address space.
ENPROTOOPT	The <i>optname</i> parameter is unrecognized, or the <i>level</i> parameter is not SOL_SOCKET.

Related Calls

getibmssockopt(), getsockopt(), ibmsflush(), setsockopt()

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);
  }
}
```

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. Refer to *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
<i>stayopen</i>	A nonzero flag value prevents the cached local host table contents from being freed after an <code>endnetent()</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

`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 *z/OS Communications Server: IP Configuration Reference*.

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

int setprotoent(int stayopen)
```

Parameter	Description
<code>stayopen</code>	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 *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 <i>hlq.ETC.SERVICES</i> closing after each call to setservent().
-----------------	---

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
<i>s</i>	The socket descriptor
<i>level</i>	The level for which the option is being set
<i>optname</i>	The name of a specified socket option. Refer to Appendix D, “GETSOCKOPT/SETSOCKOPT command values” for the numeric values of <i>optname</i> .
<i>optval</i>	The pointer to option data
<i>optlen</i>	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 begin 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	<p>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.</p> <p>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.</p> <p>Note:</p> <p>When Nagle algorithm is enabled, TCP will wait to send small amounts of data until the acknowledgment for the previous data is received.</p> <p>When Nagle algorithm is disabled, TCP will send small amounts of data even before the acknowledgment for previous data sent is received.</p>
--------------------	--

The following keywords are recognized at the socket level:

Keyword	Description
---------	-------------

SO_RCVBUF	<p>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 occurs:</p>
------------------	---

- | |
|--|
| <ul style="list-style-type: none"> • If the protocol is TCP, a return value of -1 and errno of ENOBUFS is set. The receive buffer size is unchanged. <p>For maximum values for the TCP protocol, see the TCPCONFIG TCPRCVBUFRSIZE and TCPMAXRCVBUFSIZE parameters in the <i>z/OS Communications Server: IP Configuration Guide</i>.</p> <ul style="list-style-type: none"> • If the protocol is UDP or RAW, a return value of 0 is returned and the buffer size is set to 65535. |
|--|

SO_SNDBUF	<p>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 occurs:</p>
------------------	---

- | |
|---|
| <ul style="list-style-type: none"> • If the protocol is TCP, a return value of -1 and errno of ENOBUFS is set. The send buffer size is unchanged. <p>For maximum values for the TCP protocol, see the TCPCONFIG TCPSENDBUFRSIZE parameters in the <i>z/OS Communications Server: IP Configuration Guide</i>.</p> <ul style="list-style-type: none"> • 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 keepalive mechanism for a stream socket. The default is *disabled*. When activated, the keepalive 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_OOINLINE

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 enable.
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 <i>s</i> parameter is not a valid socket descriptor.
EFAULT	Using <i>optval</i> and <i>optlen</i> parameters would result in an attempt to access storage outside the caller address space.
ENOBUFS	No buffer space is available.
ENOPROTOPT	The <i>optname</i> parameter is unrecognized, or the <i>level</i> parameter is not SOL_SOCKET.

Example

See “getsockopt()” on page 145 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 37 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
<i>s</i>	The socket descriptor.
<i>how</i>	The <i>how</i> condition can have a value of 0, 1, or 2, where: <ul style="list-style-type: none">• 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
-------	-------------

Errno	Description
EBADF	Indicates that <i>s</i> is not a valid socket descriptor.
EINVAL	The <i>how</i> 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(int onoff)
```

Parameter	Description
<i>onoff</i>	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
<i>onoff</i>	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
<i>domain</i>	The address domain requested. It is either <code>AF_INET</code> or <code>AF_IUCV</code> .
<i>type</i>	The type of socket created, either <code>SOCK_STREAM</code> , <code>SOCK_DGRAM</code> , or <code>SOCK_RAW</code> .
<i>protocol</i>	The protocol requested. Possible values are 0, <code>IPPROTO_UDP</code> , or <code>IPPROTO_TCP</code> .

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 <code>AF_INET</code> and <code>AF_IUCV</code> 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 <code>AF_INET</code> domain only.
--	---

SOCK_RAW	
-----------------	--

	Provides the interface to internal protocols (such as IP and ICMP). This type is supported in the <code>AF_INET</code> 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. Once 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.

Once a connection has been established between stream sockets, any of the data transfer calls can be used: (`read()`, `write()`, `send()`, `recv()`, `readv()`, `writenv()`, `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()`, `writenv()`, `send()`, and `recv()` can be used also. See “`connect()`” on page 115 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
<i>clientid</i>	Points to the <i>clientid</i> of the application from which you are taking a socket.
<i>hisdesc</i>	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 <i>hisdesc</i> parameter does not specify a valid socket descriptor owned by the other application. The socket has already been taken.
EFAULT	Using the <i>clientid</i> parameter as specified would result in an attempt to access storage outside the caller address space.
EINVAL	The <i>clientid</i> 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 <i>clientid</i> 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 `perror()` function.

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

void tcperror(char *s)
```

Parameter	Description
<i>s</i>	A NULL or NULL-terminated character string

Example

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 122 or “`ioctl()`” on page 161 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 wishes 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
<code>s</code>	Socket descriptor.
<code>buf</code>	Points to the buffer holding the data to be written.
<code>len</code>	Length in bytes of <code>buf</code> .

Return Values

If successful, the number of bytes written is returned. The value -1 indicates an error. `Errno` identifies the specific error.

Errno	Description
<code>EBADF</code>	Indicates that <code>s</code> is not a valid socket descriptor.
<code>EFAULT</code>	Using the <code>buf</code> and <code>len</code> parameters would result in an attempt to access storage outside the caller address space.
<code>ENOBUFS</code>	Buffer space is not available to send the message.
<code>EWOULDBLOCK</code>	Indicates that <code>s</code> 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 122 or “ioctl()” on page 161.

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 wishes 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 <i>iov</i> and <i>iovcnt</i> 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 <i>s</i> 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 section contains sample C socket programs. The C source code can be found in the *hlq.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 765.

Program	Description
MTCCCLNT	C socket MTC client
MTCSRVR	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 (TCPC)

Following is an example of a C socket TCP client program. The source code can be found in the TCPC member of the *hlq.SEZAINST* data set.

```

/**** IBMCOPYR *****/
/*
/* Part Name: TCPC
/*
/*
/* 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
/*
/**** IBMCOPYR *****/
static char ibmcopyr[] =
    "TCPC      - 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 */
    /*

```

Figure 47. C socket TCP client sample (Part 1 of 3)

```

    * 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)

```

Figure 47. C socket TCP client sample (Part 2 of 3)


```

{
    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 47. C socket TCP client sample (Part 3 of 3)

C socket TCP server (TCPS)

The following is an example of a C socket TCP server program. The source code can be found in the TCPS member of the *hlq.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 */
    /*

```

Figure 48. C socket TCP server sample (Part 1 of 3)

```

    * 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);
}

```

Figure 48. C socket TCP server sample (Part 2 of 3)

```

/*
 * 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 48. C socket TCP server sample (Part 3 of 3)

C socket UDP server (UDPS)

The following is an example of a C socket UDP server program. The source code can be found in the UDPS member of the *hlq.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);
    }
    /*

```

Figure 49. C socket UDP server sample (Part 1 of 3)

```

* 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 paramters
* 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\

```

Figure 49. C socket UDP server sample (Part 2 of 3)

```

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 49. C socket UDP server sample (Part 3 of 3)

C socket UDP client (UDPC)

The following is an example of a C socket UDP client program. The source code can be found in the UDPC member of the *hlq.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)
    {

```

Figure 50. C socket UDP client sample (Part 1 of 2)


```

    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 50. C socket UDP client sample (Part 2 of 2)

Chapter 11. Using the X/Open Transport Interface (XTI)

This chapter describes the 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 chapter apply only to unconnected sessions.

For more information on the XTI protocol, see *CAE Specification: X/Open Transport Interface (XTI)*

Software requirements

Application programs using the X/Open Transport Interface (XTI) require the following:

- *hlq*.SEZACMAC (macro library routines)
- *hlq*.SEZACMTX (executable modules)
- *hlq*.SEZALOAD (executable modules)
- *hlq*.SEZAINST (sample programs)
- Current z/OS language environment run-time library

What is provided

The XTI support provided with TCP/IP includes the following:

- The XTI library containing the XTI calls for C language programmers
- The XTI management services that allow you to include additional protocol mappers
- The RFC1006 protocol mapping component that creates the protocol expected by the XTI interface

For more information about RFC1006, see Appendix F, “Related protocol specifications (RFCs),” on page 811.

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 51 is a high-level diagram to show how the XTI interface works in an z/OS environment.

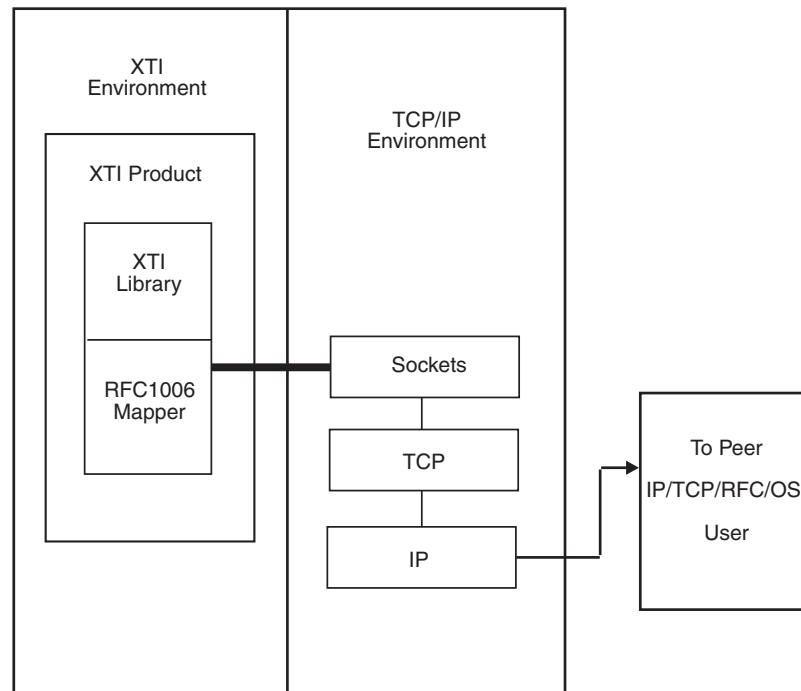


Figure 51. Using XTI with TCP/IP

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

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 230, and “XTI socket server sample program” on page 239 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.

Initializing a transport endpoint

Table 8 on page 227 lists the routines needed to initialize a transport endpoint. For more information see *CAE Specification: X/Open Transport Interface (XTI)*.

Table 8. 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.

Establishing a connection

Table 9 lists the routines needed to establish a connection. For more information see *CAE Specification: X/Open Transport Interface (XTI)*.

Table 9. 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.

Transferring data

Table 10 lists the routines needed to transfer data. For more information see *CAE Specification: X/Open Transport Interface (XTI)*.

Table 10. 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.

Releasing a connection

Table 11 lists the routines needed to release a connection. For more information see *CAE Specification: X/Open Transport Interface (XTI)*.

Table 11. 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.

Disabling a connection

Table 12 lists the routines needed to disable a connection. For more information see *CAE Specification: X/Open Transport Interface (XTI)*.

Table 12. 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.

Managing events

Table 13 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 13. 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.

Using utility calls

Table 14 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 14. 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> .

Using system calls

Table 15 lists system routines that you can use to manage your program. For more information see *CAE Specification: X/Open Transport Interface (XTI)*.

Table 15. System function calls

Call	Description
fcntl()	Controls the operating characteristics of sockets. For more information, see “selectex()” on page 181.
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 177.
selectex()	Extends the select() calls by allowing you to add an ECB to define extra events. For more information, see “selectex()” on page 181.

Compiling and linking XTI applications using cataloged procedures

Several methods are available to compile, link-edit, and run your XTI program. This section 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.

```
//XTICL JOB XTICLJOB
:
:
//CCOMP PROC REG='3072K',
//          CPARM='DEF(MVS),SOURCE,LIST,NOMARG,SEQ(73,80)',
//          INSTLIB=,
//          SEZALNK=,
//          SEZAMAC=,
:
:
//*
//*****
//*  COMPILE STEP:
//*****
//*
//COMPILE EXEC PGM=EDCCOMP,
//          PARM=('&CPARM'),
//          REGION=&REG
//STEPLIB DD DSN=&SEZALNK,DISP=SHR
//          DD DSN=&SEDCLNK,DISP=SHR
//          DD DSN=&IBMLINK,DISP=SHR
//          DD DSN=&SEDCOMP,DISP=SHR
//SYSLIB DD DSN=&SEZAMAC,DISP=SHR
//          DD DSN=&CHEADRS,DISP=SHR,DCB=(BLKSIZE=3120)
//SYSIN DD DSN=&INSTLIB(&INSTMEM),DISP=SHR
//SYSLIN DD DSN=&OBJLIB(&INSTMEM),DISP=SHR
//SYMSGS DD DSN=&CMMSG,DISP=SHR
//SYSPRINT DD SYSOUT=&SOUT
//SYSPRT DD SYSOUT=&SOUT
//SYSTEM DD DUMMY
:
:
//*
//*****
//*  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=&SEDCBSE,DISP=SHR
//          DD DSN=&IBMBASE,DISP=SHR
//SYSLMOD DD DSN=&XTILOAD,DISP=SHR
//SYSPRINT DD SYSOUT=&SOUT
//SYSUT1 DD DSN=&SYSUT1,
//          UNIT=VIO,
//          DISP=(NEW,DELETE),
//          SPACE=(32000,(30,30))
// 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)
//*
```

Note: For more information about compiling and linking, refer to the *IBM C/370 Programming Guide*.

Understanding XTI sample programs

This section contains sample XTI socket programs. The XTI source code can be found in the *hlq.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	EZAEC0YL	XTI socket client sample program
XTISC	EZAEC0YM	XTI socket server sample program

XTI socket client sample program

The following is an example of an XTI socket client program:


```

/**** IBMCOPYR *****/
/*
/* Component Name: XTICC.C (alias EZAE0YL)
/*
/* 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: EZAE0YL
/*
/*
/**** 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 "xti006.h"
#include "stdio.h"

```

Figure 52. Sample client code for XTI (Part 1 of 9)

```

/*
 * 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 pnum = 102;
char *port = "102";
char *ctsel = "client";
char *stsel = "server";
unsigned int rqlen = 0;

```

Figure 52. Sample client code for XTI (Part 2 of 9)

```

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];
        pnum = (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

```

Figure 52. Sample client code for XTI (Part 3 of 9)

```

*/

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;

```

Figure 52. Sample client code for XTI (Part 4 of 9)

```

/*
 * issue connect request
 */

looking = t_connect(fd,&scall,&rcall);
if (looking < 0 & t_errno != TNO DATA)
{
    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 != TNO DATA)
        {
            t_error("ERROR ON RCVCONNECT");
            cleanup(fd);
            exit(t_errno);
        }

/*
 * place message in buffer
 */

memset(buf,'B',25);

/*
 * send message to server
 */

```

Figure 52. Sample client code for XTI (Part 5 of 9)

```

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
    {
        t_error("ERROR DISCONNECTING FROM SERVER");
        cleanup(fd);
        exit(t_errno);
    }

/*
 * if fd is an endpoint, try to close it
 */

```

Figure 52. Sample client code for XTI (Part 6 of 9)

```

    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,tse1str1006,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;

/*
 * tse1 represented as a string
 */

char          *tse1str1006;

/*
 * one possible endpoint to close if
 * an error occurs in forming address
 */

int    fd1;

/*
 * other possible endpoint to close
 */

int    fd2;

```

Figure 52. Sample client code for XTI (Part 7 of 9)

```

{

/*
 * 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 */

```

Figure 52. Sample client code for XTI (Part 8 of 9)


```

    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 52. Sample client code for XTI (Part 9 of 9)

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 (API),” on page 95.

The following is an example of 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 "xti006.h"
#include "stdio.h"

/*

```

Figure 53. Sample server code for XTI (Part 1 of 9)

```

* 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";

```

Figure 53. Sample server code for XTI (Part 2 of 9)

```

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],0_NONBLOCK,NULL)) < 0)
    {
        t_error("Error on t_open");
        exit(t_errno);
    }

    /*
     * establish endpoint to t_accept() on
     */

    if ((resfd = t_open(tprov[0],0_NONBLOCK,NULL)) < 0)
    {
        t_error("Error on t_open");
        cleanup(fd);
        exit(t_errno);
    }

```

Figure 53. Sample server code for XTI (Part 3 of 9)

```

}

/*
 * 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, \
               pnum, \
               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);
}

```

Figure 53. Sample server code for XTI (Part 4 of 9)

```

    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);
        }

```

Figure 53. Sample server code for XTI (Part 5 of 9)

```

/*
 * 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)

```

Figure 53. Sample server code for XTI (Part 6 of 9)

```

    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,tse1str1006,fd1,fd2)

/*
 * formats the provided address information
 * into the buffer for RFC1006
 */

/*
 * address buffer to be filled in
 */

struct xti1006tsap *addrbuf1006;

```

Figure 53. Sample server code for XTI (Part 7 of 9)


```

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
     */

```

Figure 53. Sample server code for XTI (Part 8 of 9)

```

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 53. Sample server code for XTI (Part 9 of 9)

Chapter 12. Using the macro application programming interface (API)

This chapter 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
- Defining storage for the API macro
- 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

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 may only be invoked in TCB mode (task mode).
Cross-memory mode	These APIs may 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.
INITAPI/TERMAPI macros	The INITAPI/TERMAPI macros 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 interface call that is issued after an EZASMI CALL interface 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 CALL interface might 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 UNIX System Services	Address spaces using the EZASMI CALL interface should not use any UNIX System Services facilities such as EZASMI CALL interface with UNIX System Services BPX1 Assembler Callable Services or EZASMI CALL interface with Language Environment for z/OS functions in the C/C++ Runtime Library. Doing so can yield unpredictable results.

Linkage conventions for the macro API

Input register information

Before invoking the sockets API, the general purpose registers (GPRs) need to contain the following:

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.

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.

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.

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 API macro

The macro API requires both global and task storage areas.

The global storage area must be known and addressable to all socket users within an address space. It should be defined by the primary task within the address space, preferably by the JobStep task. This task can define storage in two ways:

- Enter the instruction EZASMI TYPE=GLOBAL with STORAGE=CSECT as part of your program code. This makes the program nonreentrant, but simplifies the code.
- Enter the instruction EZASMI TYPE=GLOBAL with STORAGE=DSECT as part of your program code. This instruction generates the equate field GWALENTH, which is equal to the length of the storage area. GWALENTH is used to issue an MVS GETMAIN to allocate the required storage.

The defining task must make the address of this storage available to all other tasks within the address space using the interface. Programs running in these tasks must define the storage mapping using the instructions EZASMI TYPE=GLOBAL and STORAGE=DSECT.

The second storage area is a task storage area that 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 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.
- Code the instruction EZASMI TYPE=TASK with STORAGE=DSECT as part of the program code. The expansion of this instruction generates the equate field, TIELENT, which is equal to the length of the storage area. This can be used to issue an MVS GETMAIN to allocate the required storage.

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.

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 255. 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(TIELENT)	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

This section describes the parameters and concepts common to the macros described in this section.

Parameter	Description
<i>address</i>	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. <pre>MVC SOCKNO,=H'2' EZASMI TYPE=BIND,S=SOCKNO</pre>
<i>*indaddr</i>	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. <pre>MVC SOCKNO,=H'2' LA 0,SOCKNO ST 0,SOCKADD EZASMI TYPE=BIND,S=*SOCKADD</pre>
<i>(reg)</i>	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. <pre>MVC SOCKNO,=H'2' LA 3,SOCKNO EZASMI TYPE=BIND,SOCKNO=(3)</pre>
<i>'value'</i>	A literal value for the parameter; for example, AF='INET'

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, “Return codes,” on page 781.

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. Once 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.

Once 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 440.

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.

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 may 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.

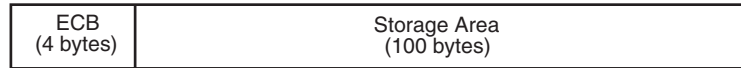


Figure 54. ECB input parameter

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.

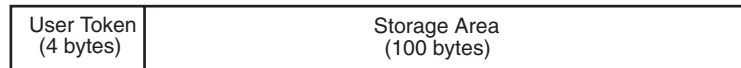


Figure 55. User token setting

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 does one of the following:
 - 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 will be posted when the function has completed. Note that the ECB may 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 will be 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:

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
WAIT     ECB=MYECB    TELL MVS TO WAIT UNTIL READ IS DONE
```

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 will be 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 will be deferred

- EZASMI macro calls within the asynchronous exits.

- Linkage stack.

- 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.

The unsolicited event exit routine allows the application to specify an unsolicited event exit routine to be invoked when an unsolicited event takes place. This exit routine can be a part of the program that specifies it, or it can be a separate module. In either case, it must be resident at the time the INITAPI is issued and must stay resident until the TERMAPI is issued.

The diagram illustrates the UEXIT instruction format. It consists of a 32-bit instruction word. The first bit is the UEXIT flag. The next 26 bits are the address field. The following 5 bits are the *indaddr field. The final 6 bits are the (reg) field.

258 z/OS V1R6.0 CS: IP Application Programming Interface Guide

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 may be the cause of an error. For more information on the SOCKAPI trace option, refer to *z/OS Communications Server: IP Diagnosis Guide*.

Macros for assembler programs

This section contains the description, syntax, parameters, and other related information for every macro included in this API.

ACCEPT

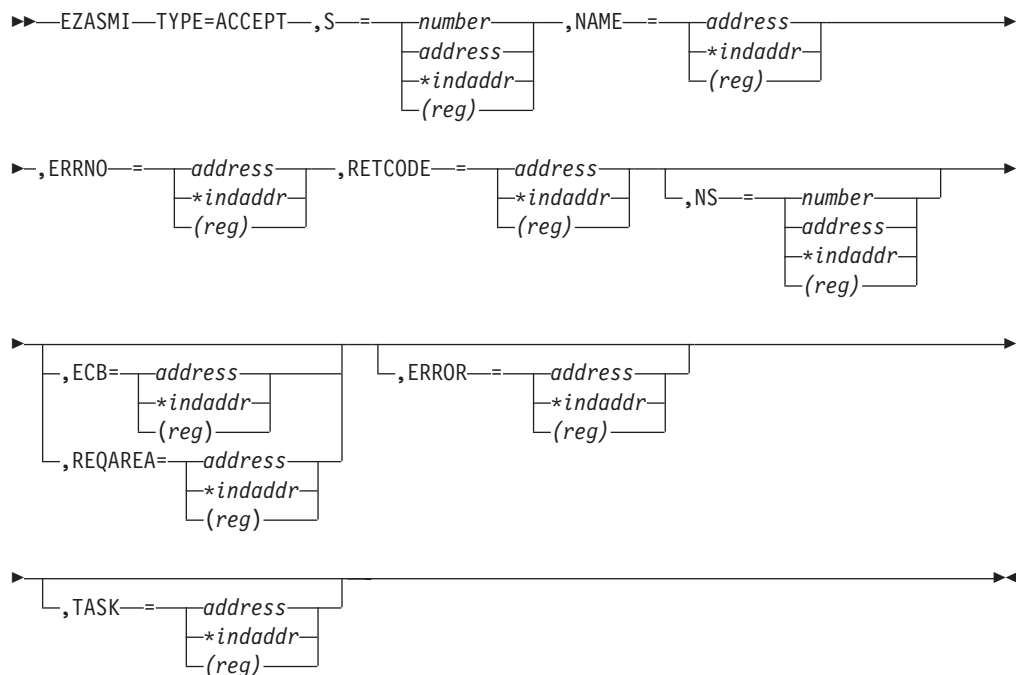
The ACCEPT macro is issued when the server receives a connection request from a client. ACCEPT points to a socket that was created with a SOCKET macro and marked by a LISTEN macro. If a process waits for the completion of connection requests from several peer processes, a later ACCEPT macro can block 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 does the following:

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 “Environmental restrictions and programming requirements” on page 429.
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, "Return codes," on page 781 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

A 100-byte 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 only interested in receiving new TCP connections or UDP datagrams that are targeted towards 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.

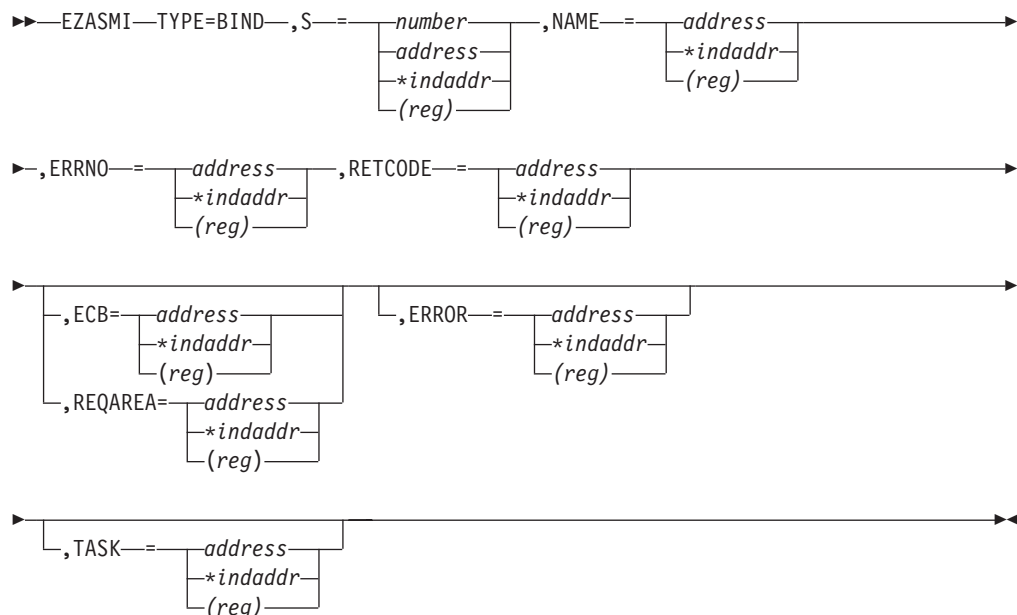
Note: Even if an application specifies a value of 0 for the IP address on the BIND, the system administrator can override that value by specifying the BIND parameter on the PORT reservation statement in the TCP/IP profile. This has a similar effect to the application specifying an explicit IP address on the BIND macro. For more information, refer to the *z/OS Communications Server: IP Configuration Reference*.

Note that even if an application specifies a value of 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. This has a similar effect to the application specifying an explicit IP address on the BIND macro. For more information, refer to 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 “Environmental restrictions and programming requirements” on page 429.
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 27 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 will bind 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 will bind 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 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.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, **ERRNO** contains a valid error number. Otherwise, ignore **ERRNO**.

See Appendix B, "Return codes," on page 781 for information about **ERRNO** return codes.

RETCODE Output parameter. A fullword binary field that returns one of the following:

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

A 100-byte 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.

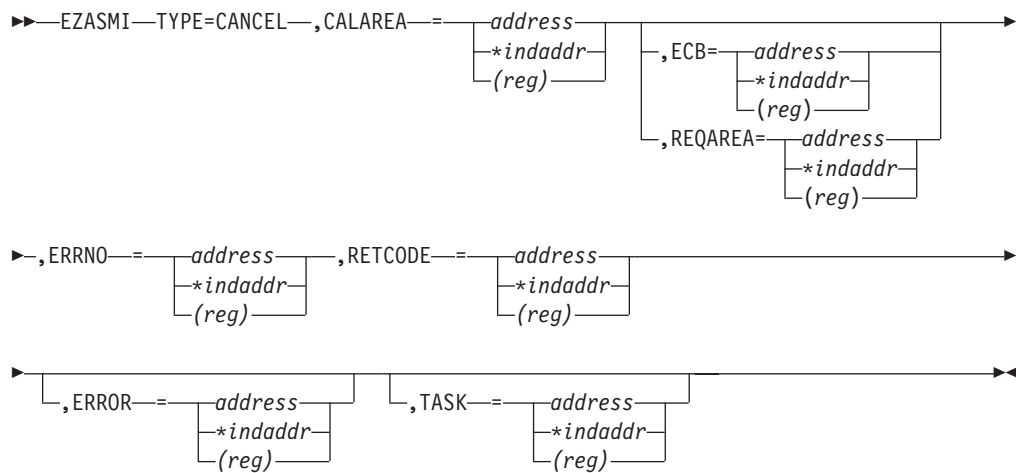
TASK Input parameter. The location of the task storage area in your program.

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 "Environmental restrictions and programming requirements" on page 429.
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

A 100-byte 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

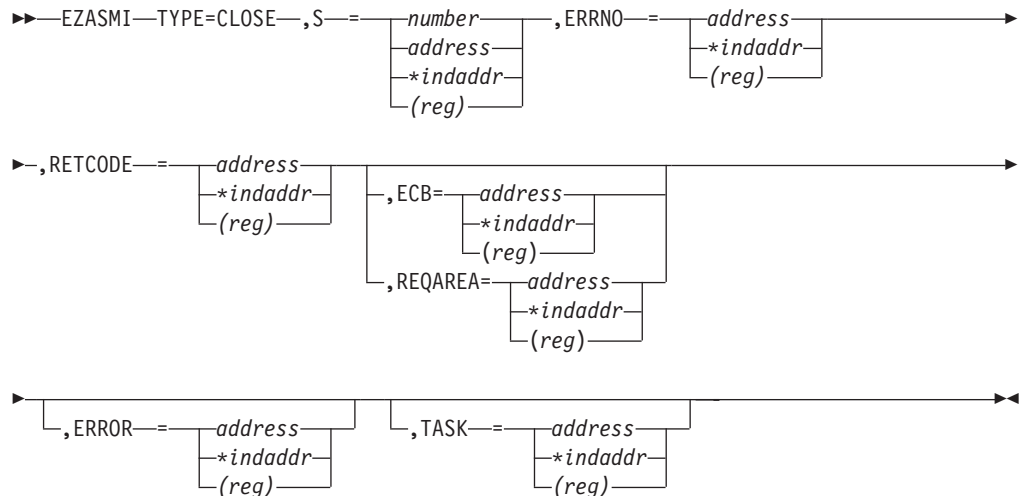
The CLOSE macro shuts down the socket and frees the resources that are allocated to the socket. Issue the SHUTDOWN macro before you issue the CLOSE macro.

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 530.

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 “Environmental restrictions and programming requirements” on page 429.
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, "Return codes," on page 781 for information about ERRNO return codes.						
RETCODE	Output parameter. A fullword binary field that returns one of the following: <table> <tr> <th>Value</th><th>Description</th></tr> <tr> <td>0</td><td>Successful call.</td></tr> <tr> <td>-1</td><td>Check ERRNO for an error code.</td></tr> </table>	Value	Description	0	Successful call.	-1	Check ERRNO for an error code.
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: <div> <p>For ECB A 4-byte ECB posted by TCP/IP when the macro completes.</p> <p>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.</p> <p>For ECB/REQAREA A 100-byte storage field used by the interface to save the state information.</p> </div> <p>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.</p>						
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.

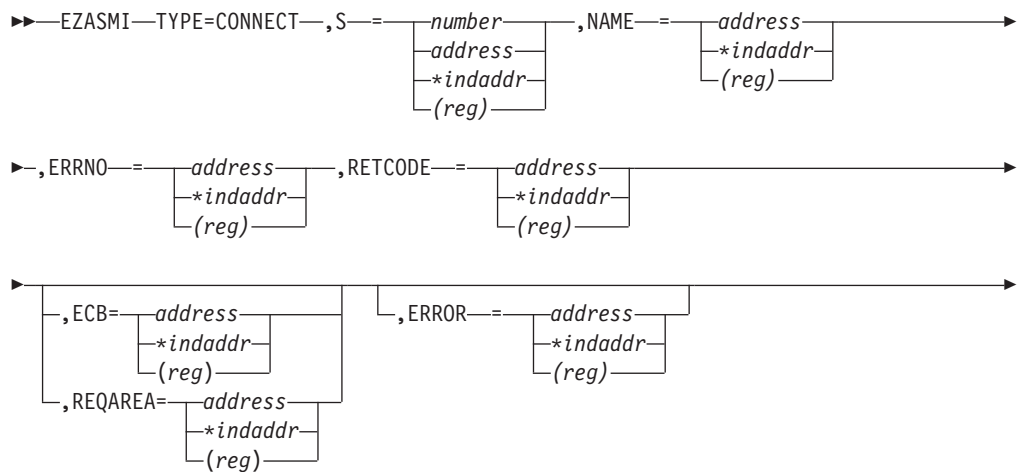
If the socket is in nonblocking mode, the return code indicates the success of the connection request.

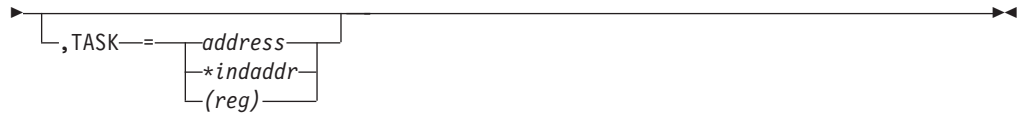
- A 0 RETCODE indicates that the connection was completed.
- A nonzero RETCODE with an ERRNO of 36 (EINPROGRESS) indicates that the connection could not be completed, but since the socket is nonblocking, the CONNECT macro completes its processing.

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 "Environmental restrictions and programming requirements" on page 429.
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 address of the server host machine in network byte order. For example, if the Internet address is 129.4.5.12 in dotted decimal notation, it is set to X'8104050C'.

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 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 address, in network byte order, of the client host machine. For example, if the IPv6 Internet 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.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, **ERRNO** contains a valid error number. Otherwise, ignore **ERRNO**.

See Appendix B, "Return codes," on page 781 for information about **ERRNO** return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following:

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

A 100-byte 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.

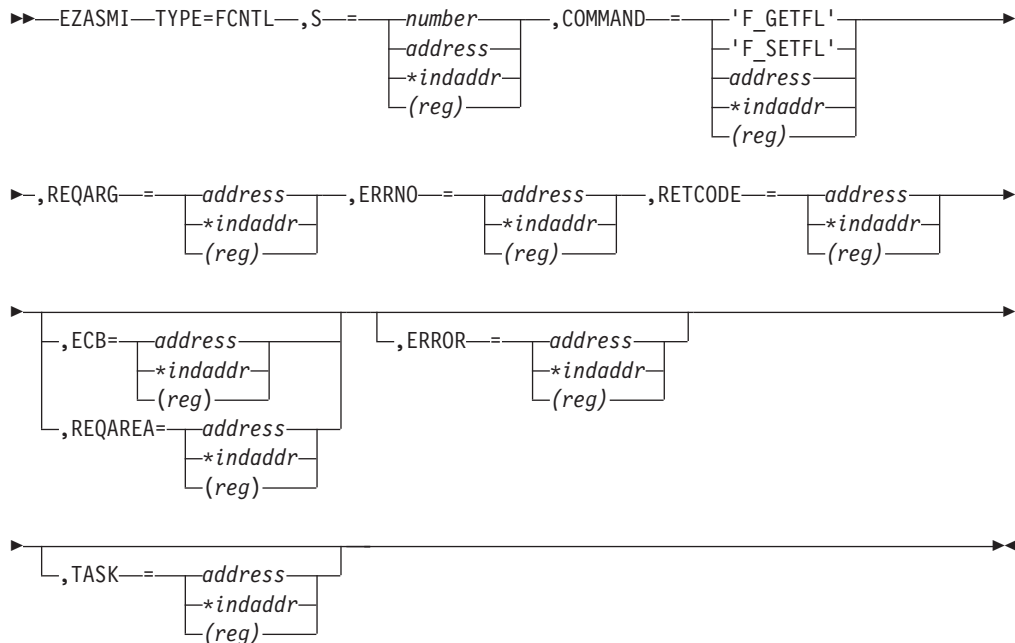
FCNTL

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 487 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 “Environmental restrictions and programming requirements” on page 429.
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. Refer to the *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, "Return codes," on page 781 for information about **ERRNO** return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following:

- 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

A 100-byte 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.

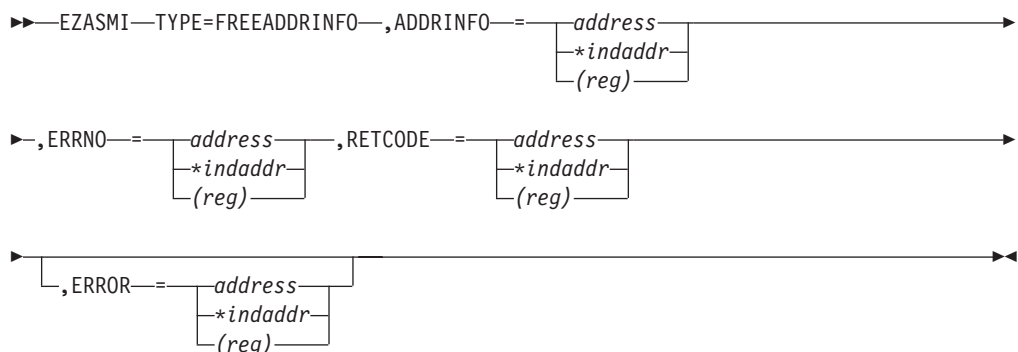
TASK Input parameter. The location of the task storage area in your program.

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 “Environmental restrictions and programming requirements” on page 429.
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.
-----------------	---

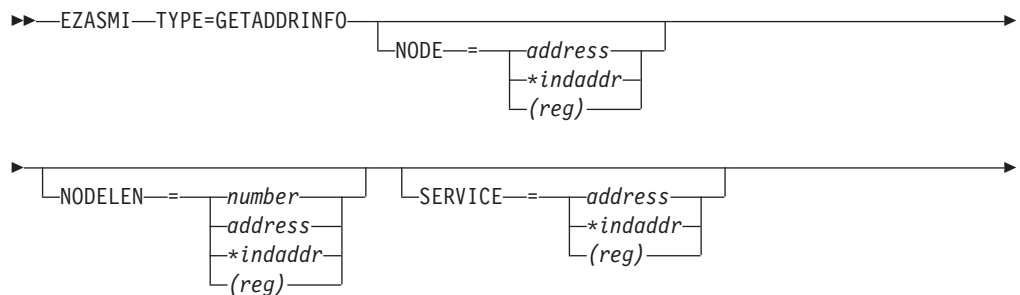
ERRNO	Output parameter. A fullword binary field. If RETCODE is negative, ERRNO contains a valid error number. Otherwise, ignore the ERRNO field. See Appendix B, “Return codes,” on page 781 for information about ERRNO return codes.						
RETCODE	Output parameter. A fullword binary field that returns one of the following: <table> <tr> <th>Value</th><th>Description</th></tr> <tr> <td>0</td><td>Successful call.</td></tr> <tr> <td>-1</td><td>Check ERRNO for an error code.</td></tr> </table>	Value	Description	0	Successful call.	-1	Check ERRNO for an error code.
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.						

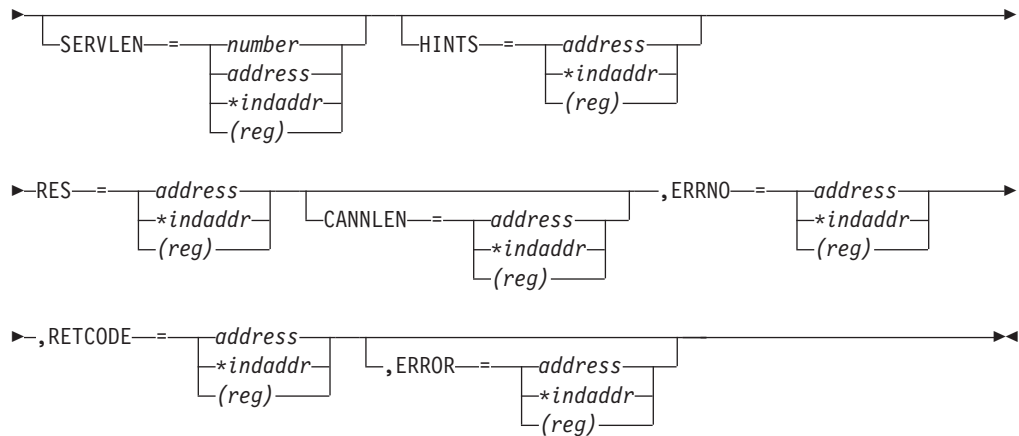
GETADDRINFO

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 “Environmental restrictions and programming requirements” on page 429.
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 will consist of up to NODELEN or up to the first binary zero.
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 will consist 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 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 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:

AI_PASSIVE (X'00000001')

- Specifies how to fill in the **NAME** pointed to in 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 (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** will be set to **INADDR_ANY** for an IPv4 address or **IN6ADDR_ANY** for an IPv6 address.
- If this flag is not set, the returned address information will be 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** will be set to the default loopback address for an IPv4 address (127.0.0.0) 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 will accept IPv4-mapped IPv6 addresses. When the **AI_ALL** flag is not also specified and no IPv6 addresses are found, then a query is made for IPv4 addresses. If any

IPv4 addresses are found, they are returned as IPv4-mapped IPv6 addresses.

- If the **AF** field does not have the value of **AF_INET6** or the **AF** field contains **AF_UNSPEC** but IPv6 is not supported on the system, this flag is ignored.

AI_ALL (X'00000020')

- 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). When the **AF** field has a value of **AF_UNSPEC** when the system supports IPv6 and **AI_ALL** is set, the caller accepts IPv6 addresses and either IPv4 (if **AI_V4MAPPED** is not set) or IPv4-mapped IPv6 (if **AI_V4MAPPED** is set) 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 and any found are returned as IPv4 addresses (if **AI_V4MAPPED** was not set) or as IPv4-mapped IPv6 addresses (if **AI_V4MAPPED** was set).
- If the **AF** field does not have the value of **AF_INET6**, or the value of **AF_UNSPEC** when the system supports IPv6, 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 is true:

- If the system is IPv6 enabled and has at least one IPv6 interface, 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, the Resolver will make a query for IPv4 (A DNS) records.

The loopback address is not considered in this case as a valid interface.

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 are 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** will be accepted, it will only be valid 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** will be failed with return code of **EAI_BADFLAGS**.

If **SOCTYPE** and **PROTO** are both specified as 0 then **TYPE=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**), then **TYPE=GETADDRINFO** will search the appropriate services file (such as, *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 provision exists 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* means one of the following:

- **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 will accept any protocol.

The following are the acceptable protocols:

Protocol name	Decimal value	Description
IPPROTO_TCP	6	TCP
IPPROTO_UDP	17	user datagram

If **SOCKTYPE** 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** will be failed with return code of **EAI_BADFLAGS**.

If **PROTO** and **SOCKTYPE** are both specified as 0, then **TYPE=GETADDRINFO** will proceed as follows:

- If **SERVICE** is null, or if **SERVICE** is numeric, then any returned addrinfos will default to a specification of **SOCKTYPE** as **SOCK_STREAM**.
- If **SERVICE** is specified as a service name (for example, **SERVICE=FTP**), then **TYPE=GETADDRINFO** will search the appropriate services file (such as, *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 provision exists 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 means one of the following:

- **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** will be failed with return code of **EAI_SERVICE**.

NAMELEN

A fullword binary field. This field must be 0.

CANONNAME

A fullword binary field. This field must be 0.

NAME

A fullword binary field. This field must be 0.

NEXT A fullword binary field. This field must be 0.

RES

Initially a fullword binary field. On a successful return, this field will contain a pointer to a chain of one or more addrinfo structures. The addrinfo storage will be allocated in the caller's key. This structure will occupy storage in the key that this command is issued in the application address space. Use the **EZBREHST** macro to establish addrinfo mapping. This pointer will also be used as input to the **TYPE=FREEADDRINFO** macro which must be used to free storage obtained by this macro.

The address information structure contains the following fields. All fields in this structure that are not filled in with an explicit value will be 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 may 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 may 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 may 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. The length of the **NAME** socket address structure. The value returned in this field may 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. 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 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 will contain the length of the returned canonical name.

NAME

A fullword binary field. The address of the returned socket address structure. The value returned in this field may 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.

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, “Return codes,” on page 781 for information about **ERRNO** return codes.

RETCODE Output parameter. A fullword binary field that returns one of the following:

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.
--------------	---

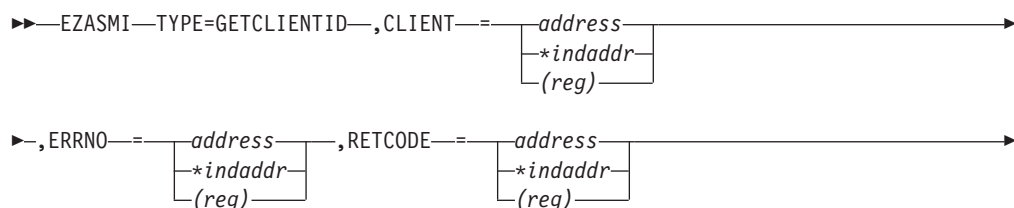
GETCLIENTID

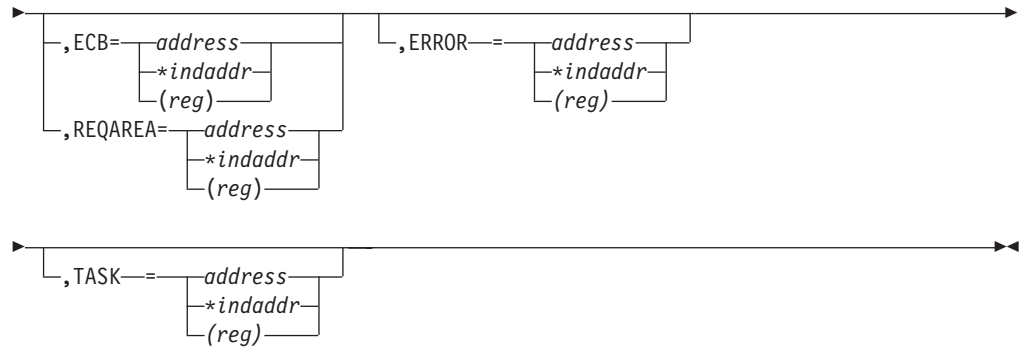
The GETCLIENTID macro returns the identifier by which the calling application is known to the TCP/IP address space. The client ID structure returned is used by the GIVESOCKET and TAKESOCKET macros.

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 “Environmental restrictions and programming requirements” on page 429.
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	This is 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	Initially, the application provides a pointer to an 8-byte character field, which 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 0's. 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, "Return codes," on page 781 for information about ERRNO return codes.						
RETCODE	Output parameter. A fullword binary field that returns one of the following: <table> <tr> <th>Value</th><th>Description</th></tr> <tr> <td>0</td><td>Successful call.</td></tr> <tr> <td>-1</td><td>Check ERRNO for an error code.</td></tr> </table>	Value	Description	0	Successful call.	-1	Check ERRNO for an error code.
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 REOAREA

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/REOAREA

A 100-byte 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.
--------------	---

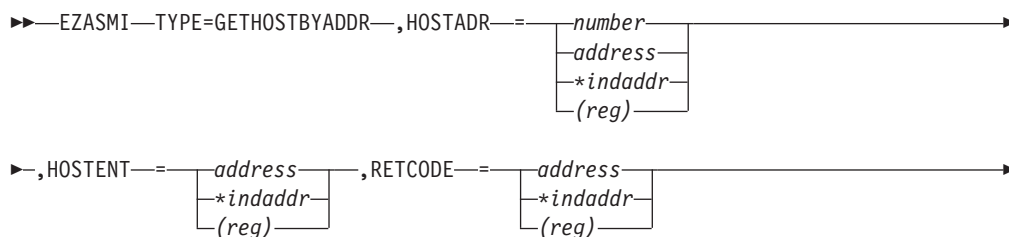
TASK	Input parameter. The location of the task storage area in your program.
-------------	---

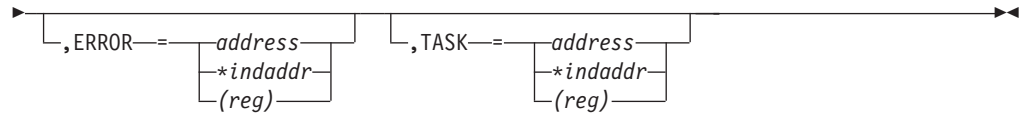
GETHOSTBYADDR

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 “Environmental restrictions and programming requirements” on page 429.
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 56 on page 286.						
RETCODE	Output parameter. A fullword binary field that returns one of the following: <table> <tr> <th>Value</th><th>Description</th></tr> <tr> <td>>0</td><td>Successful call.</td></tr> <tr> <td>-1</td><td>An error occurred.</td></tr> </table>	Value	Description	>0	Successful call.	-1	An error occurred.
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.						

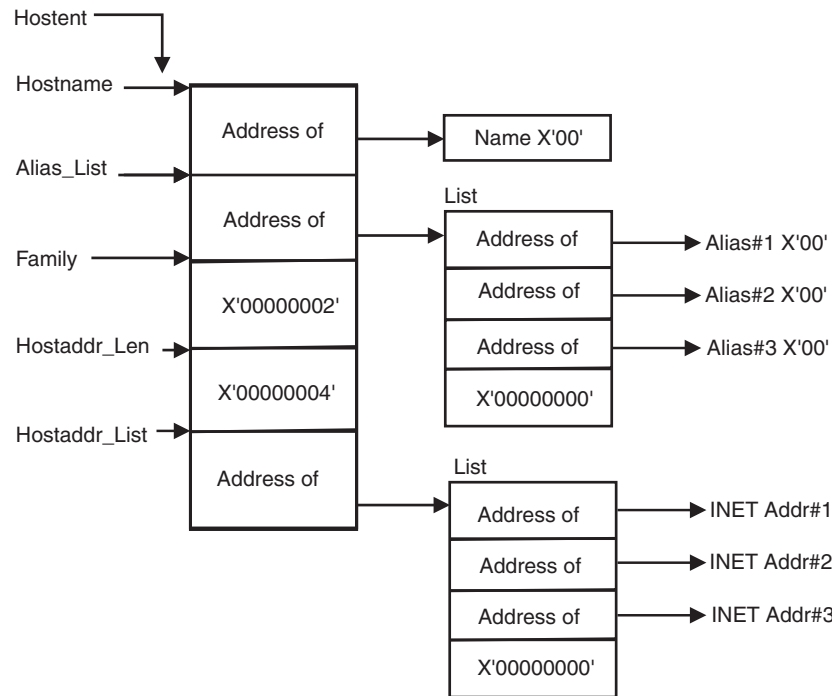


Figure 56. HOSTENT structure returned by the GETHOSTBYADDR macro

GETHOSTBYADDR returns the HOSTENT structure shown in Figure 56. 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.

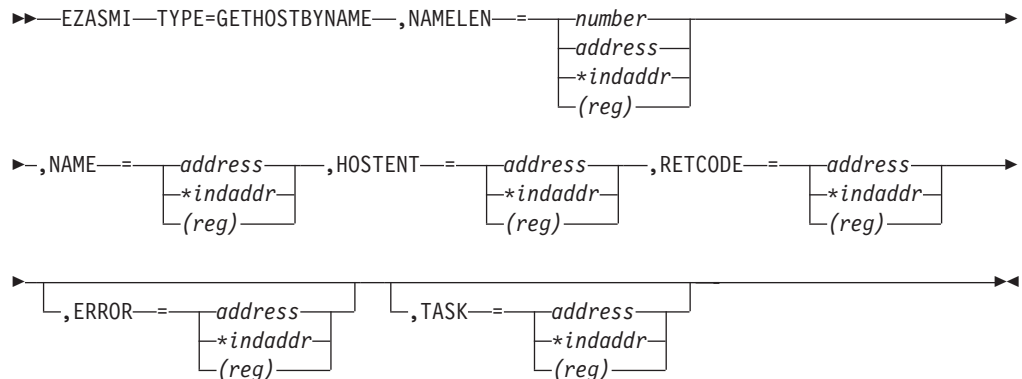
The name resolution attempted depends on how the resolver is configured and if any local host tables exist. Refer to *z/OS Communications Server: IP Configuration Guide* for information about configuring the resolver and using local host tables.

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 “Environmental restrictions and programming requirements” on page 429.
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 will be 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 57 on page 288.
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.

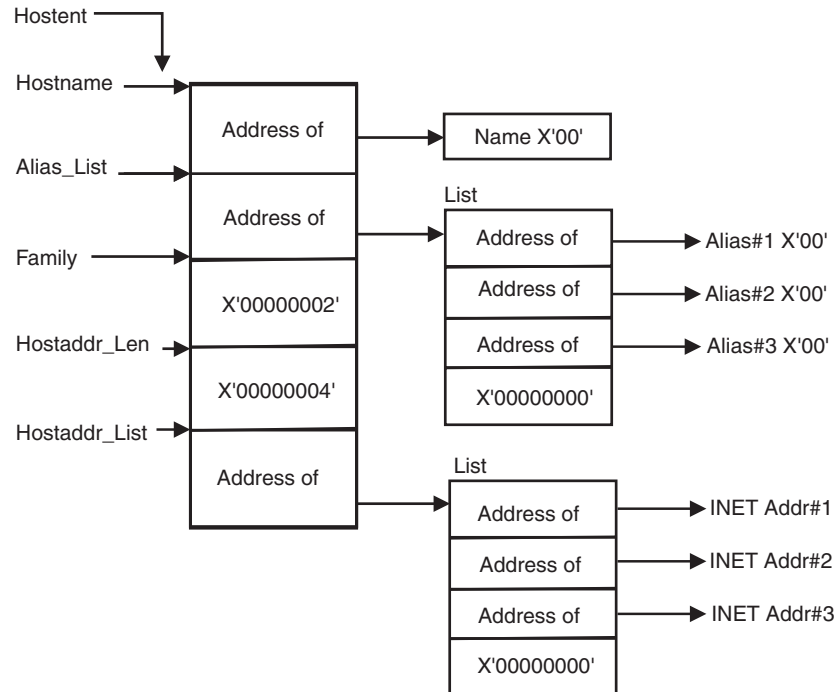


Figure 57. HOSTENT structure returned by the GETHOSTBYNAME macro

GETHOSTBYNAME returns the HOSTENT structure shown in Figure 57. 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'.

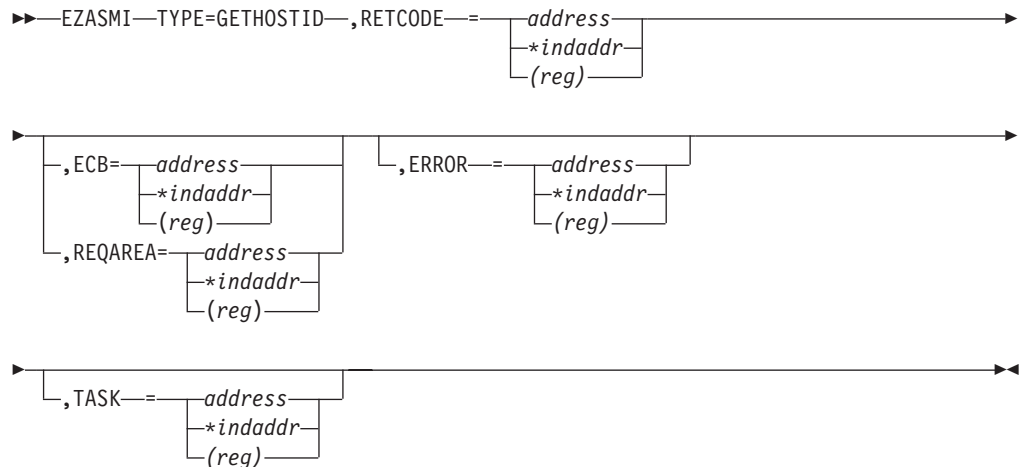
The HOSTENT structure uses indirect addressing to return a variable number of alias names and Internet addresses.

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 “Environmental restrictions and programming requirements” on page 429.
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 **API**TYPE=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

A 100-byte 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.

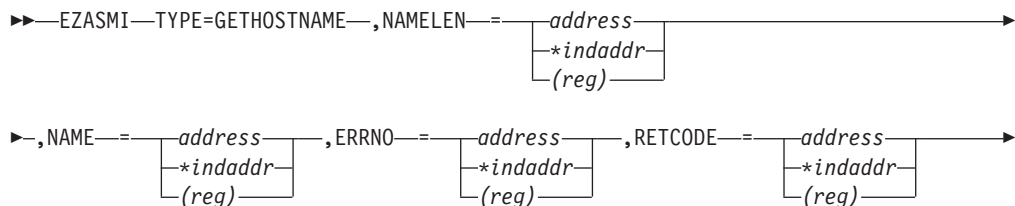
GETHOSTNAME

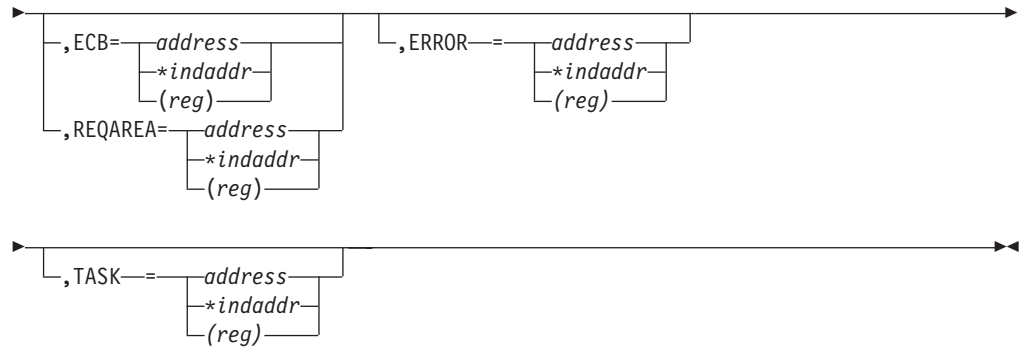
The GETHOSTNAME macro returns the name of the host processor on which the program is running. As many as NAMELEN characters are copied into the NAME field.

Note: The host name 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 "Environmental restrictions and programming requirements" on page 429.
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 maximum length that can be specified in the field is 255 characters.						
NAME	Initially, the application provides a pointer to a receiving field for the host name. TCP/IP Services allows a maximum length of 24 characters. This field is filled with a host name with the length returned in NAMELEN when the call completes. The name returned is the value from the TCP/IP stack's TCPIP.DATA HOSTNAME statement.						
ERRNO	Output parameter. A fullword binary field. If RETCODE is negative, ERRNO contains a valid error number. Otherwise, ignore ERRNO . See Appendix B, "Return codes," on page 781 for information about ERRNO return codes.						
RETCODE	Output parameter. A fullword binary field that returns one of the following: <table> <tr> <th>Value</th><th>Description</th></tr> <tr> <td>0</td><td>Successful call.</td></tr> <tr> <td>-1</td><td>An error occurred.</td></tr> </table>	Value	Description	0	Successful call.	-1	An error occurred.
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 A 100-byte 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.						

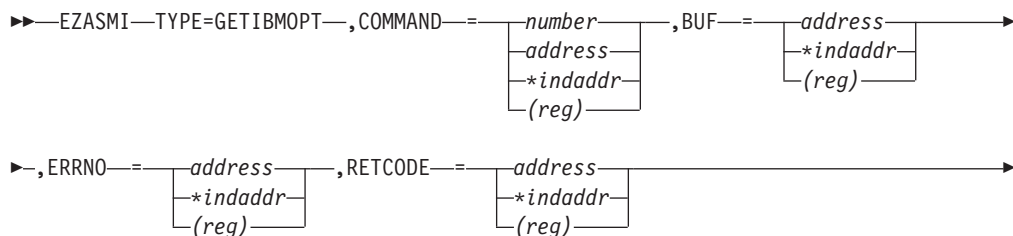
TASK	Input parameter. The location of the task storage area in your program.
-------------	---

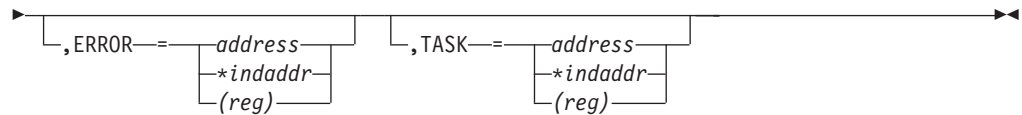
The GETIBMOPT macro returns the number of TCP/IP images installed on a given MVS system and the status, version, and name of each.

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 *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 “Environmental restrictions and programming requirements” on page 429.
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)

Figure 58. 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 version field is:

Version	Field
TCP/IP OS/390 CS V2R10	X'0510'
TCP/IP z/OS CS V1R2	X'0612'
TCP/IP z/OS CS V1R4	X'0614'
TCP/IP z/OS CS V1R5	X'0615'
TCP/IP z/OS CS V1R6	X'0616'

The name field is the PROC name, left-justified, 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, "Return codes," on page 781 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.

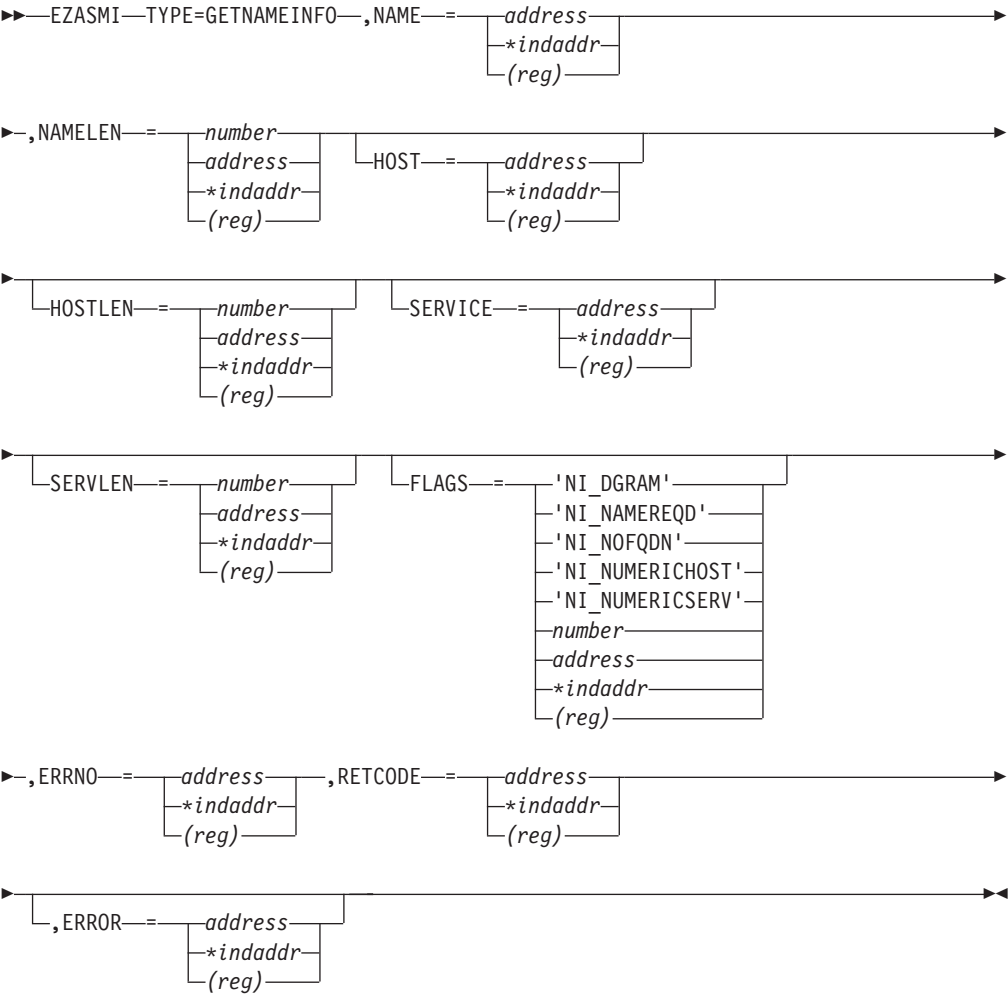
GETNAMEINFO

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.
<p>Note: See "Addressability mode (Amode) considerations" under "Environmental restrictions and programming requirements" on page 429.</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		
NAME	<p>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 <i>SOCKADDR</i> 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.</p> <p>The IPv4 socket address structure must specify the following fields:</p> <table> <tr> <th>Field</th><th>Description</th></tr> </table>	Field	Description
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 specifying 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 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.

FLOW-INFO

This field is ignored by the **TYPE=GETNAMEINFO** macro.

IPv6-ADDRESS

A 16-byte binary field specifying the 128-bit IPv6 Internet address, in network byte order.

SCOPE-ID

This field is ignored by the **TYPE=GETNAMEINFO** macro.

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 an error is returned. Either **HOST/HOSTLEN** or **SERVICE/SERVLN** parameters, or both parameters, are required. An error occurs if both are omitted. The **HOST** name being queried will consist of up to **HOSTLEN** or up to the first binary 0.

HOSTLEN

Initially an input parameter. A fullword binary field that contains the length of the **HOST** storage used to contain the returned resolved host name. If **HOSTLEN** is 0 on input, then the resolved host name will not be returned. **HOSTLEN** must be equal to or greater than the length of the longest host name to be returned.

The TYPE=GETNAMEINFO will return the host name up to the length specified by HOSTLEN. On output, HOSTLEN will contain the length of the returned resolved host name. This is an optional field but if specified you must also code HOST. Either HOST/HOSTLEN or SERVICE/SERVLEN parameters, or both parameters, are required. An error occurs if both are omitted.

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 presentation form of the service address is returned instead of its name. This is an optional field but if specified you must also code SERVLEN. The SERVICE name being queried will consist of up to SERVLEN or up to the first binary zero. Either HOST/HOSTLEN or SERVICE/SERVLEN parameters, or both parameters, are required. An error occurs if both are omitted.

SERVLEN 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 SERVLEN is 0 on input, then the service name information will not be returned. SERVLEN must be equal to or greater than the length of the longest service name to be returned. The TYPE=GETNAMEINFO will return the service name up to the length specified by SERVLEN. On output, SERVLEN will contain the length of the returned resolved service name. This is an optional field but if specified you must also code SERVICE. Either HOST/HOSTLEN or SERVICE/SERVLEN parameters, or both parameters, are required. An error occurs if both are omitted.

FLAGS An input parameter. A fullword binary field. This is an optional field. 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.
'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.

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, "Return codes," on page 781 for information about ERRNO return codes.

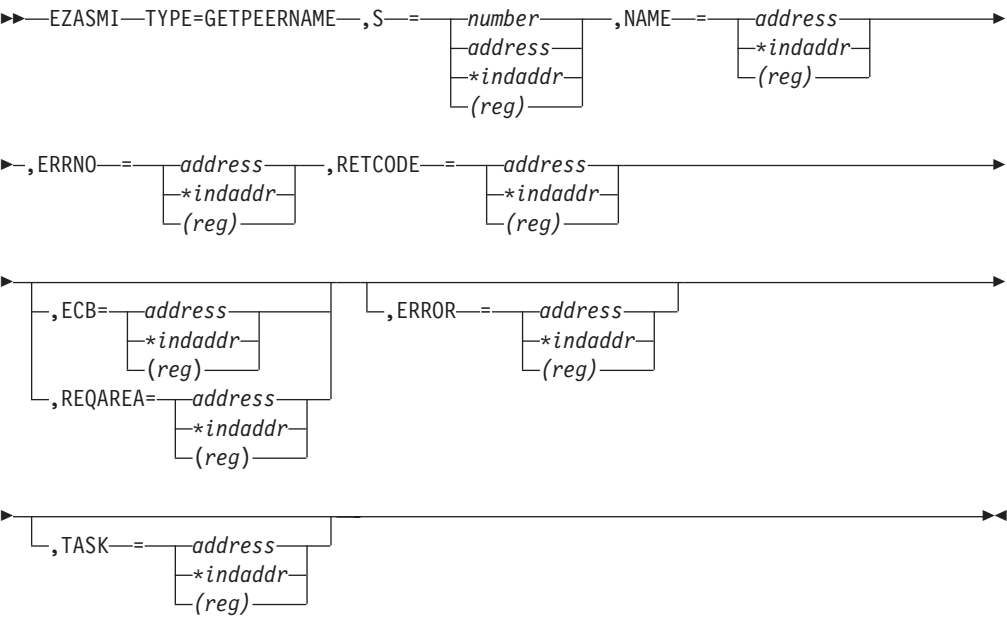
RETCODE	Output parameter. A fullword binary field that returns one of the following:
Value	Description
0	Successful call.
-1	Check ERRNO for an error code.
ERROR	Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot 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 “Environmental restrictions and programming requirements” on page 429.
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, "Return codes," on page 781 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

A 100-byte 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

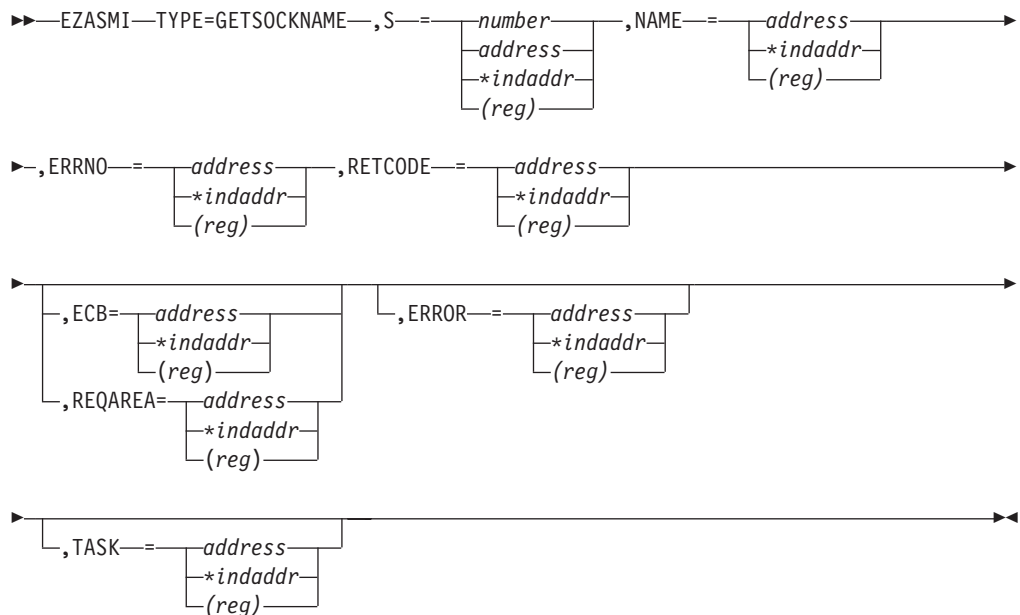
The **GETSOCKNAME** macro stores the name of the socket into the structure pointed to by **NAME** and returns the address to the socket that has been bound. If the socket is not bound to an address, the macro returns with the **FAMILY** field completed and the rest of the structure set to zeros.

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 “Environmental restrictions and programming requirements” on page 429.
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 filled in 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 0s. 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, "Return codes," on page 781 for information about **ERRNO** return codes.

RETCODE Output parameter. A fullword binary field that returns one of the following:

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

A 100-byte 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.

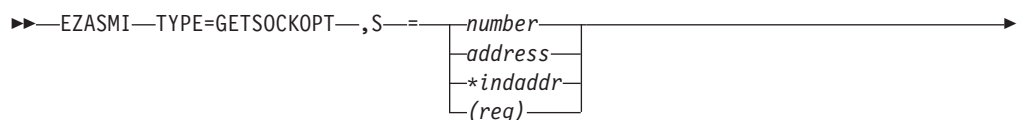
GETSOCKOPT

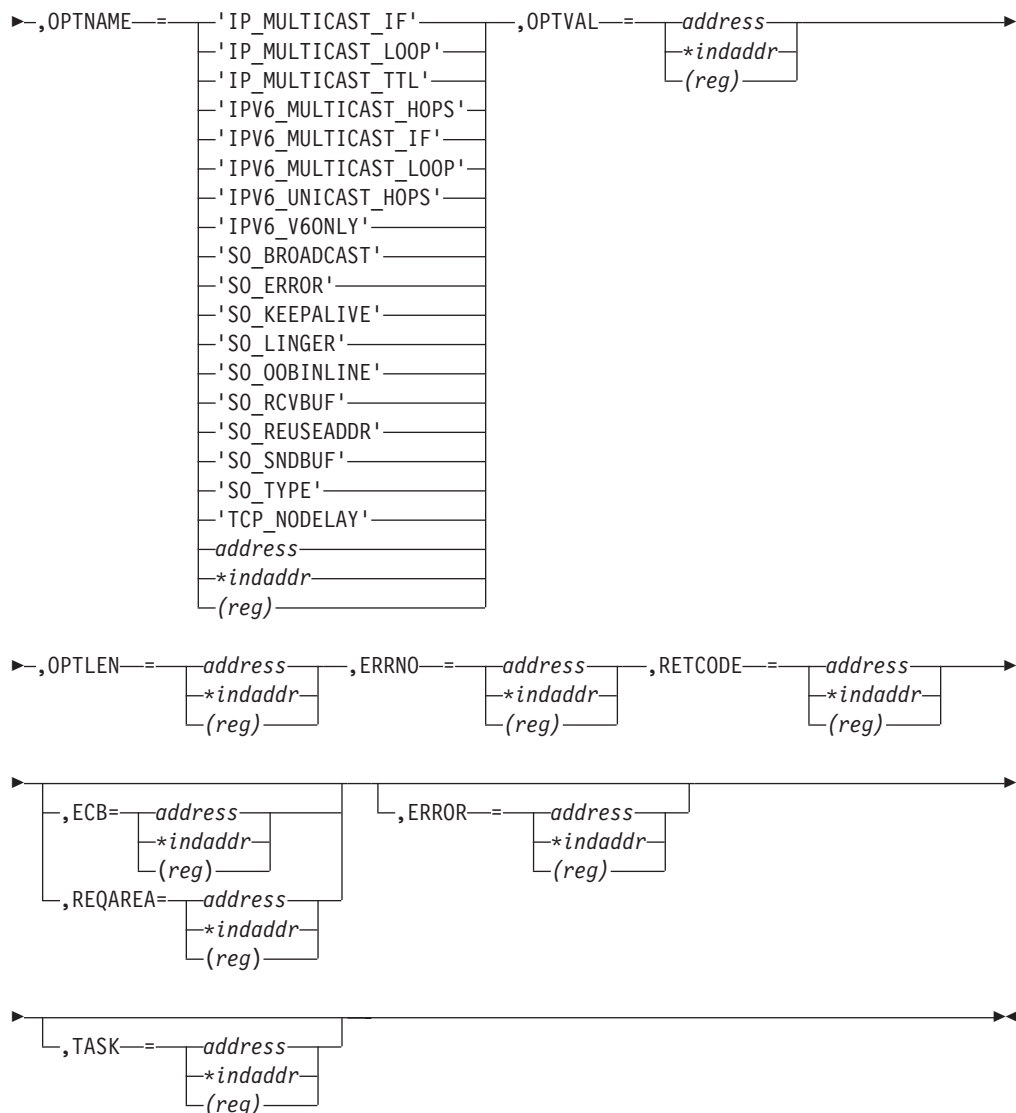
The GETSOCKOPT macro gets the options associated with a socket that were set using the SETSOCKOPT macro.

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 "Environmental restrictions and programming requirements" on page 429.
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 following table for a list of the options and their unique requirements. See Appendix D, "GETSOCKOPT/SETSOCKOPT command values," on page 803 for the numeric values of OPTNAME .
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 .
OPTVAL	Output parameter. See the following table 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, "Return codes," on page 781 for information about ERRNO return codes.

RETCODE Output parameter. A fullword binary field that returns one of the following:

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

A 100-byte 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 16. *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 <i>hlq.SEZAINST(CBLOCK)</i> for the PL/I example of IP_MREQ.</p> <p>The IP_MREQ definition for COBOL:</p> <pre> 01 IP-MREQ. 05 IMR-MULTIADDR PIC 9(8) BINARY. 05 IMR-INTERFACE PIC 9(8) BINARY.</pre>	N/A

Table 16. *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.</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 <i>hlq</i>.SEZAINST(CBLOCK) for the PL/I example of IP_MREQ.</p> <p>The IP_MREQ definition for COBOL:</p> <pre>01 IP-MREQ. 05 IMR-MULTIADDR PIC 9(8) BINARY. 05 IMR-INTERFACE PIC 9(8) BINARY.</pre>	N/A
<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>	A 4-byte binary field containing an IPv4 interface address.	A 4-byte binary field containing an IPv4 interface address.
<p>IP_MULTICAST_LOOP</p> <p>Use this option to control or determine whether a copy of multicast datagrams are 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>	A 1-byte binary field containing the value of '00'x to 'FF'x.	A 1-byte binary field containing the value of '00'x to 'FF'x.

Table 16. *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 <i>hlq</i>.SEZAINST(CBLOCK) for the PL/I example of IPV6_MREQ.</p> <p>The IPV6_MREQ definition for COBOL:</p> <pre> 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.</pre>	<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 <i>hlq</i>.SEZAINST(CBLOCK) for the PL/I example of IPV6_MREQ.</p> <p>The IPV6_MREQ definition for COBOL:</p> <pre> 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.</pre>	<p>N/A</p>

Table 16. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
IPv6_MULTICAST_HOPS Use to set or obtain the hop limit used for outgoing multicast packets. This is an IPv6-only socket option.	Contains a 4-byte binary value specifying the multicast hops. If not specified, then the default is 1 hop. -1 indicates use stack default. 0 - 255 is the valid hop limit range. 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.	Contains a 4-byte binary value in the range from 0 to 255 indicating the number of multicast hops.
IPv6_MULTICAST_IF Use this option to set or obtain the index of the IPv6 interface used for sending outbound multicast datagrams from the socket application. This is an IPv6-only socket option.	Contains a 4-byte binary field containing an IPv6 interface index number.	Contains a 4-byte binary field containing an IPv6 interface index number.
IPv6_MULTICAST_LOOP 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. This is an IPv6-only socket option.	A 4-byte binary field. To enable, set to 1. To disable, set to 0.	A 4-byte binary field. If enabled, contains a 1. If disabled, contains a 0.
IPv6_UNICAST_HOPS Use this option to set or obtain the hop limit used for outgoing unicast IPv6 packets. This is an IPv6-only socket option.	Contains a 4-byte binary value specifying the unicast hops. If not specified, then the default is 1 hop. -1 indicates use stack default. 0 - 255 is the valid hop limit range. Note: APF authorized applications are permitted to set a hop limit that exceeds the system configured default. CICS applications cannot execute as APF authorized.	Contains a 4-byte binary value in the range from 0 to 255 indicating the number of unicast hops.
IPv6_V6ONLY 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. This is an IPv6-only socket option.	A 4-byte binary field. To enable, set to 1. To disable, set to 0.	A 4-byte binary field. If enabled, contains a 1. If disabled, contains a 0.

Table 16. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
SO_ASCII Use this option to set or determine the translation to ASCII data option. When <code>SO_ASCII</code> is set, data is translated to ASCII. When <code>SO_ASCII</code> is not set, data is not translated to or from ASCII. Note: This is a REXX-only socket option.	To enable, set to ON. To disable, set to OFF. 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.	If enabled, contains ON. If disabled, contains OFF. 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.
SO_BROADCAST 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. Note: This option has no meaning for stream sockets.	A 4-byte binary field. To enable, set to 1 or a positive value. To disable, set to 0.	A 4-byte field. If enabled, contains a 1. If disabled, contains a 0.
SO_DEBUG Use <code>SO_DEBUG</code> 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. Notes: <ol style="list-style-type: none"> 1. This is a REXX-only socket option. 2. This option has meaning only for stream sockets. 	To enable, set to ON. To disable, set to OFF.	If enabled, contains ON. If disabled, contains OFF.
SO_EBCDIC Use this option to set or determine the translation to EBCDIC data option. When <code>SO_EBCDIC</code> is set, data is translated to EBCDIC. When <code>SO_EBCDIC</code> is not set, data is not translated to or from EBCDIC. This option is ignored by EBCDIC hosts. Note: This is a REXX-only socket option.	To enable, set to ON. To disable, set to OFF. 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.	If enabled, contains ON. If disabled, contains OFF. 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.
SO_ERROR 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.	N/A	A 4-byte binary field containing the most recent <code>ERRNO</code> for the socket.

Table 16. *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 keepalive 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 keepalive 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 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 16. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
SO_OOBINLINE Use this option to control or determine whether out-of-band data is received. Note: This option has meaning only for stream sockets. 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. 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.	A 4-byte binary field. To enable, set to 1 or a positive value. To disable, set to 0.	A 4-byte field. If enabled, contains a 1. If disabled, contains a 0.
SO_RCVBUF Use this option to control or determine the size of the data portion of the TCP/IP receive buffer. The size of the data portion of the receive buffer is protocol-specific, based on the following values prior to any SETSOCKOPT call: <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 65 535 for a raw socket 	A 4-byte binary field. To enable, set to a positive value specifying the size of the data portion of the TCP/IP receive buffer. To disable, set to a 0.	A 4-byte binary field. If enabled, contains a positive value indicating the size of the data portion of the TCP/IP receive buffer. If disabled, contains a 0.

Table 16. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
SO_REUSEADDR 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. When this option is enabled, the following situations are supported: <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, refer to the SHAREPORT option on the PORT statement in TCPIP.PROFILE. 	A 4-byte binary field. To enable, set to 1 or a positive value. To disable, set to 0.	A 4-byte field. If enabled, contains a 1. If disabled, contains a 0.
SO_SNDBUF 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: <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 65 535 for a raw socket 	A 4-byte binary field. To enable, set to a positive value specifying the size of the data portion of the TCP/IP send buffer. To disable, set to a 0.	A 4-byte binary field. If enabled, contains a positive value indicating the size of the data portion of the TCP/IP send buffer. If disabled, contains a 0.
SO_TYPE Use this option to return the socket type.	N/A	A 4-byte binary field indicating the socket type: X'1' indicates SOCK_STREAM. X'2' indicates SOCK_DGRAM. X'3' indicates SOCK_RAW.

Table 16. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
TCP_NODELAY Use this option to set or determine whether data sent over the socket is subject to the Nagle algorithm (RFC 896). 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. Note: Use the following to set TCP_NODELAY OPTNAME value for COBOL programs: <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>	A 4-byte binary field. To enable, set to a 0. To disable, set to a 1 or nonzero.	A 4-byte binary field. If enabled, contains a 0. If disabled, contains a 1.

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.

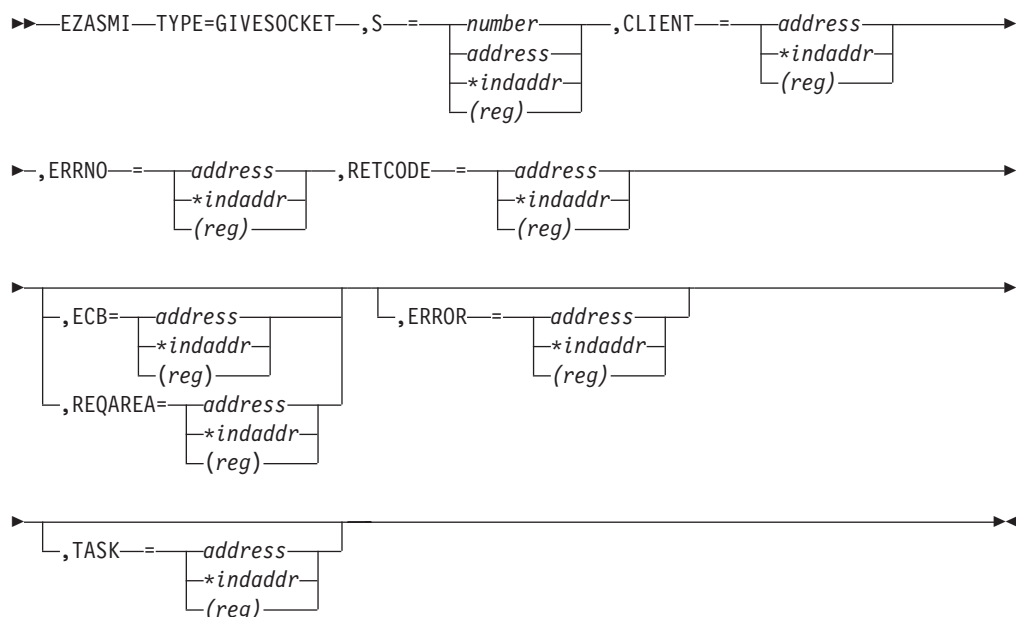
After a program has issued a GIVESOCKET macro for a socket, it can only issue a CLOSE macro for the same socket. Sockets which are given but not taken for a period of four days will be closed and will no longer be available for taking. If a select for the socket is outstanding, it is posted.

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. Note: See “Addressability mode (Amode) considerations” under “Environmental restrictions and programming requirements” on page 429.
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
	Initially, the application provides a pointer to an eight-character field, left-justified, 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, "Return codes," on page 781 for information about **ERRNO** return codes.

RETCODE Output parameter. A fullword binary field that returns one of the following:

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

A 100-byte 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 “Environmental restrictions and programming requirements” on page 429.
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.

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, it is recommended that you always code the INITAPI command.

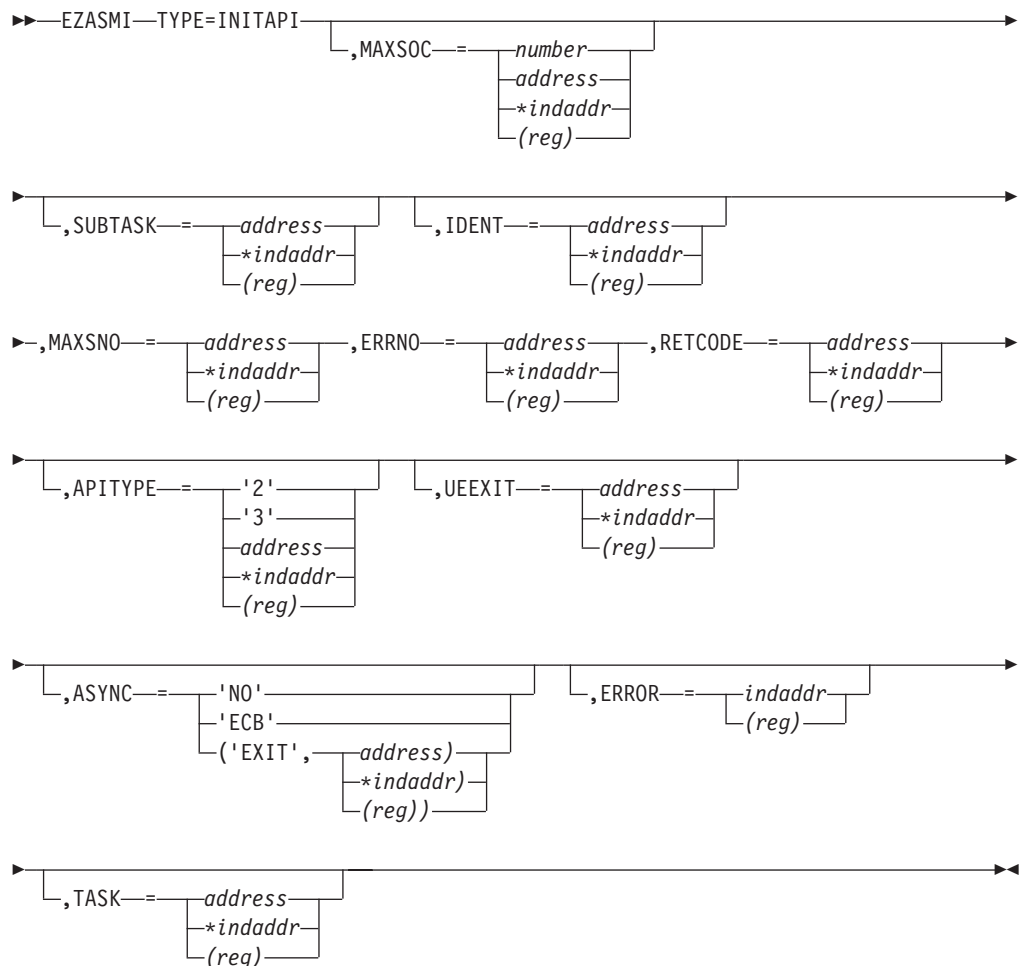
The exceptions to this rule are the following calls, which, when issued first, will generate a default INITAPI call:

- 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 “Environmental restrictions and programming requirements” on page 429.
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 *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, "Return codes," on page 781 for information about **ERRNO** return codes.

RETCODE Output parameter. A fullword binary field that returns one of the following:

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 255.

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: <ul style="list-style-type: none"> • 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.
ASYN	Optional input parameter. One of the following: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> – 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

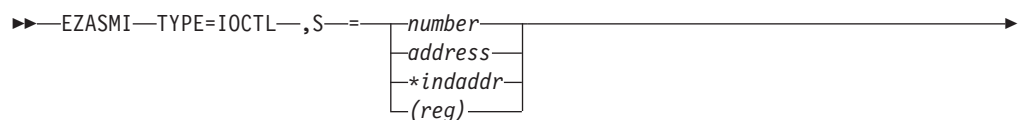
The IOCTL macro is used to control certain operating characteristics for a socket.

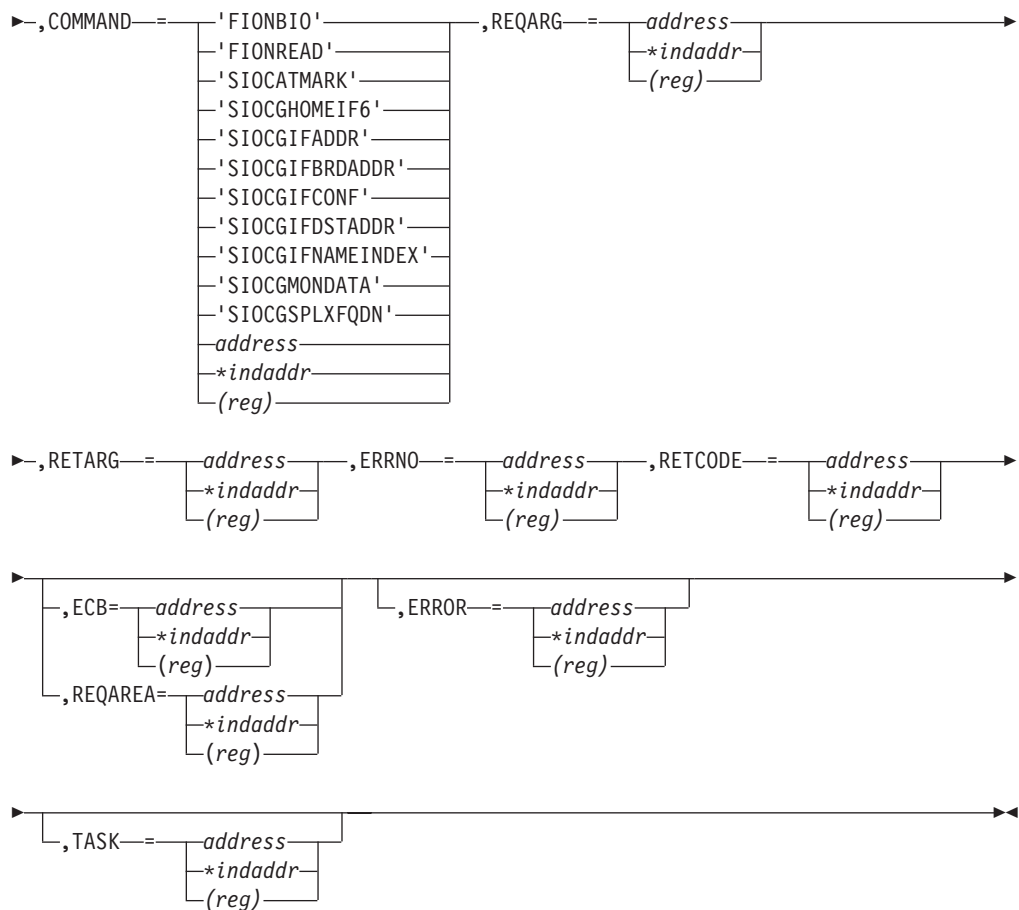
Before you issue an IOCTL macro, you must load a value representing the characteristic that you want to control in **COMMAND**.

Note: IOCTL can only be used 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 "Environmental restrictions and programming requirements" on page 429.
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 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. <ul style="list-style-type: none"> When the SIOCGHOMEIF6 IOCTL is issued, the REGARQ must contain a Network Configuration Header. The NETCONFHDR is defined in the SYS1.MACLIB(BPXIO6). 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 Figure 59.

NAME	DS	CL16	SOCKET NAME STRUCTURE
FAMILY	DC	AL2(2)	
PORT	DS	H	
ADDRESS	DS	F	
RESERVED	DS	XL8'00'	

Figure 59. Interface request structure (IFREQ) for IOCTL macro

'SIOCGIFBRDADDR'
Requests the IPv4 network interface broadcast address for an interface name. For the address format, see Figure 59.

'SIOCGIFCONF'
Requests the IPv4 network interface configuration. The configuration consists of a variable number of 32-byte arrays, formatted as shown in Figure 59.

- 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. Figure 60 contains an example of a routine that can be used to work with such a structure.

RETARG	DS	F	POINTER
COUNT	DS	F	VALUE(MAX NUMBER OF INTERFACES)
GTABLE	DS	F	GRP-IOCTL-TABLE (POINTER TO ARRAY OF TABENTRY)
TABENTRY	DS	0CL32	MAP OF TABENTRY
NAME	DS	CL16	SOCKET NAME STRUCTURE
FAMILY	DS	AL2(2)	ADDRESS FAMILY (AF_INET)
PORT	DS	H	PORT NUMBER
ADDR	DS	F	INET ADDRESS
RESERV	DS	XL8'00'	RESERVED

Figure 60. Assembler language example for SIOCGIFCONF

To get length:

1. Multiply COUNT by 32 to get ARRAY-LENGTH.
2. Set REQARG equal to ARRAY-LENGTH.
3. Issue the macro EZASMI TYPE=, S=, COMMAND=, REQARG=, RETARG=, ERRNO=, RETCODE=, ECB=, ERROR=
4. Set GTABLE to RETARG.

'SIOCGIFDSTADDR'

Requests the IPv4 network interface destination address.

'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 only returned to an interface if it has at least one available IP address. Refer to *z/OS Communications Server: IPv6 Network and Application Design Guide* for more details.

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 get the size of the array element.
3. To the size of the array add the size of IF_NITOTALIF and IF_NIENTRIES to get 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.

'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 *hlq.SEZANMAC*.

Note: The ARP counter data provided differs depending on the type of device. Refer to the *z/OS Communications Server: IP Configuration Guide* for information about devices that support ARP Offload and what is supported for each device.

'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.

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 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 17.

Table 17. 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.
SIOCGIFADDR X'C020A70D'	32	First 16-bytes - interface name. Last 16-bytes - not used.	32	Network interface address, see Figure 59 on page 321 for format.
SIOCGIFBRDADDR X'C020A712'	32	First 16-bytes - interface name. Last 16-bytes - not used.	32	Network interface address, see Figure 59 on page 321 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 Figure 59 on page 321 for format.
SIOCGIFNAMEINDEX X'4000F603'	4	First 4 bytes size of return buffer.		See IF_NAMEINDEX in macro BPXYIOCC.
SIOCGMONDATA X'C018D902'	—	See MONDATAIN structure in macro EZBZMONP.	—	See MONDATAOUT structure in macro EZBZMONP.

Table 17. IOCTL macro arguments (continued)

COMMAND/CODE	SIZE	REQARG	SIZE	RETARG
SIOCGSPLXFQDN X'C018D905'	408 ²	See sysplexFqDn and sysplexFqDnData in macro EZBZSDNP.	408 ²	See sysplexFqDn and sysplexFqDnData in macro EZBZSDNP.

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 Figure 60 on page 322 for format). 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. REQARG and RETARG must contain both sysplexFqDn and sysplexFqDnData.

ERRNO Output parameter. A fullword binary field. If **RETCODE** is negative, **ERRNO** contains a valid error number. Otherwise, ignore **ERRNO**.

See Appendix B, "Return codes," on page 781 for information about **ERRNO** return codes.

RETCODE Output parameter. A fullword binary field that returns one of the following:

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

A 100-byte 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:

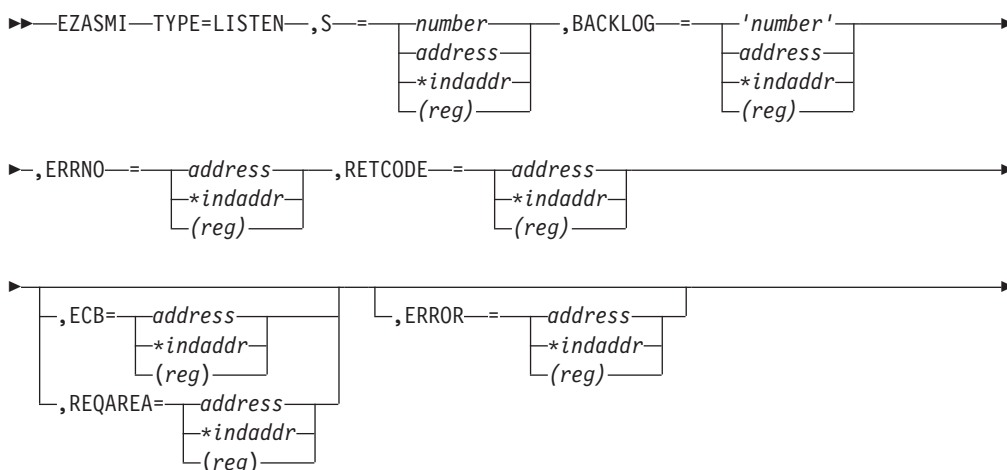
- Completes the bind, if BIND has not already been called for the socket. If the BIND has already been called for in the socket, the LISTEN macro uses what was specified in the BIND call.
- Creates a connection-request queue of a specified number of entries for incoming connection requests.

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 "Environmental restrictions and programming requirements" on page 429.
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





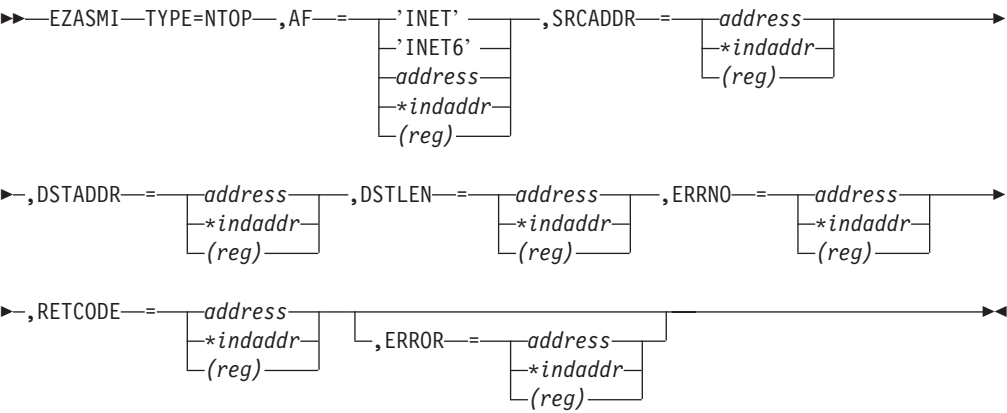
Keyword	Description						
S	Input parameter. A value or the address of a halfword binary number specifying the socket descriptor.						
BACKLOG	Input parameter. A value (enclosed in single quotation marks) or the address of a fullword binary number specifying the number of messages that can be backlogged.						
ERRNO	Output parameter. A fullword binary field. If RETCODE is negative, ERRNO contains a valid error number. Otherwise, ignore ERRNO . See Appendix B, "Return codes," on page 781 for information about ERRNO return codes.						
RETCODE	Output parameter. A fullword binary field that returns one of the following: <table> <tr> <th>Value</th><th>Description</th></tr> <tr> <td>0</td><td>Successful call.</td></tr> <tr> <td>-1</td><td>Check ERRNO for an error code.</td></tr> </table>	Value	Description	0	Successful call.	-1	Check ERRNO for an error code.
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: <p>For ECB A 4-byte ECB posted by TCP/IP when the macro completes.</p> <p>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.</p> <p>For ECB/REQAREA A 100-byte storage field used by the interface to save the state information.</p> <p>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.</p>						
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 “Environmental restrictions and programming requirements” on page 429.
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**.

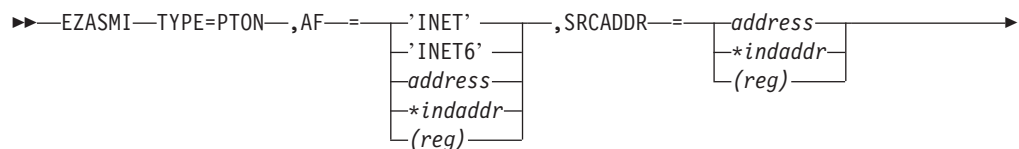
DSTLEN	Initially, an input parameter. The address of a binary halfword field that is used to specify the length of the DSTADDR field on input and upon a successful return will contain the length of the converted IP address.						
ERRNO	Output parameter. A fullword binary field. If RETCODE is negative, ERRNO contains a valid error number. Otherwise, ignore the ERRNO field. See Appendix B, “Return codes,” on page 781 for information about ERRNO return codes.						
RETCODE	A fullword binary field that returns one of the following: <table> <tr> <th>Value</th><th>Description</th></tr> <tr> <td>0</td><td>Successful call.</td></tr> <tr> <td>-1</td><td>Check ERRNO for an error code.</td></tr> </table>	Value	Description	0	Successful call.	-1	Check ERRNO for an error code.
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.						

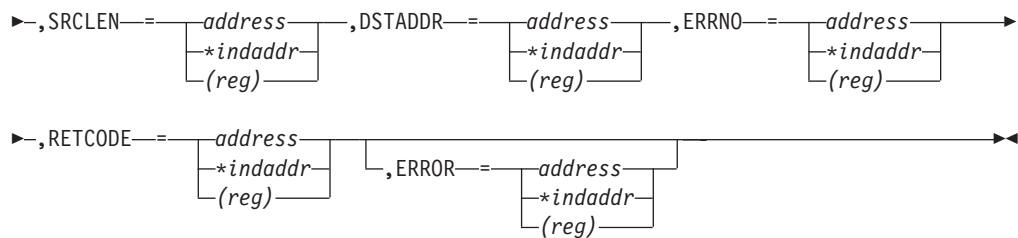
PTON

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 “Environmental restrictions and programming requirements” on page 429.
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, "Return codes," on page 781 for information about ERRNO return codes.
RETCODE	A fullword binary field that returns one of the following: 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.

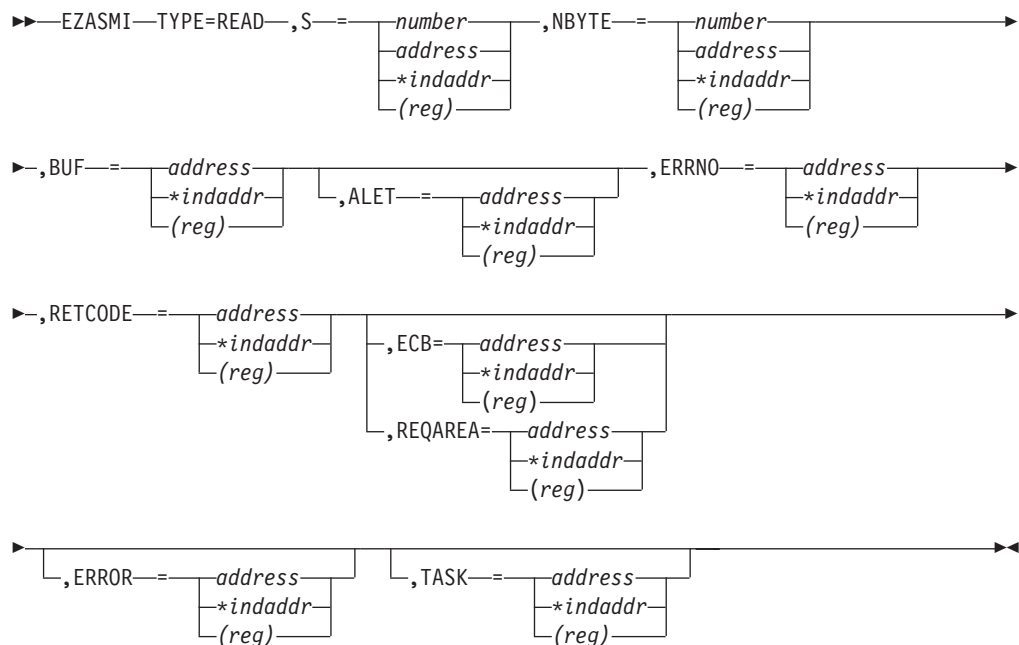
READ

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 “Environmental restrictions and programming requirements” on page 429.
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 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, **ERRNO** contains a valid error number. Otherwise, ignore the **ERRNO** field.

See Appendix B, "Return codes," on page 781 for information about **ERRNO** return codes.

RETCODE A fullword binary field that returns one of the following:

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:

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

A 100-byte 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.

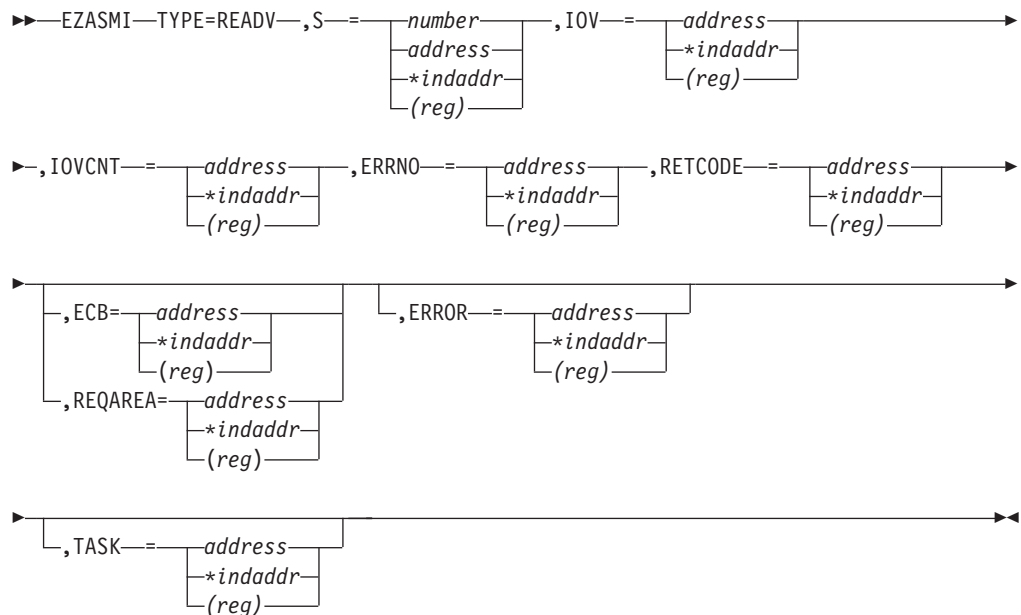
If data is not available for the socket and the socket is in blocking mode, the READ macro blocks the caller until data arrives. If data is not available and the socket is in nonblocking mode, READ returns a -1 and sets **ERRNO** to 35 (EWOULDBLOCK). See "IOCTL" on page 319 or "FCNTL" on page 272 for 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 “Environmental restrictions and programming requirements” on page 429.
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: <div style="margin-left: 20px;"> Fullword 1 Input parameter. A buffer to be filled by the completion of the call. </div>

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 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:

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

A 100-byte 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.

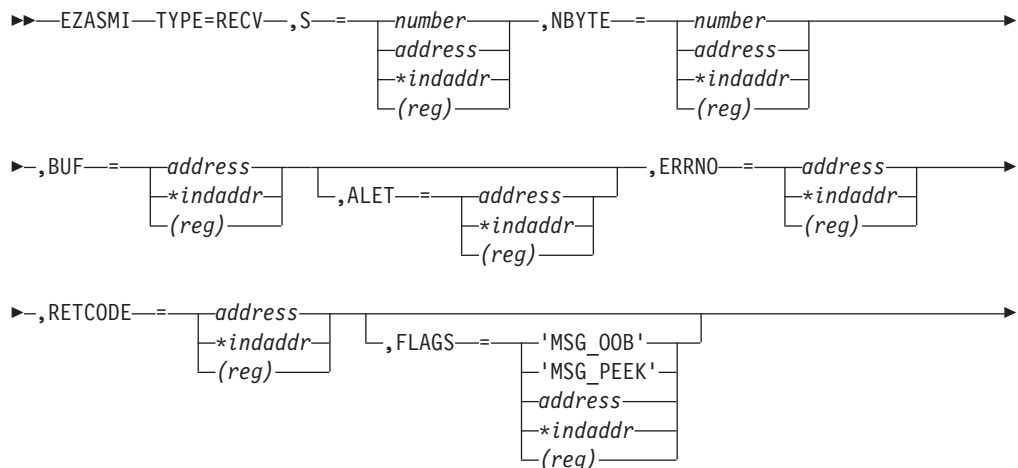
Note: Out-of-band data (called urgent data in TCP) appears to the application like a separate stream of data from the main data stream.

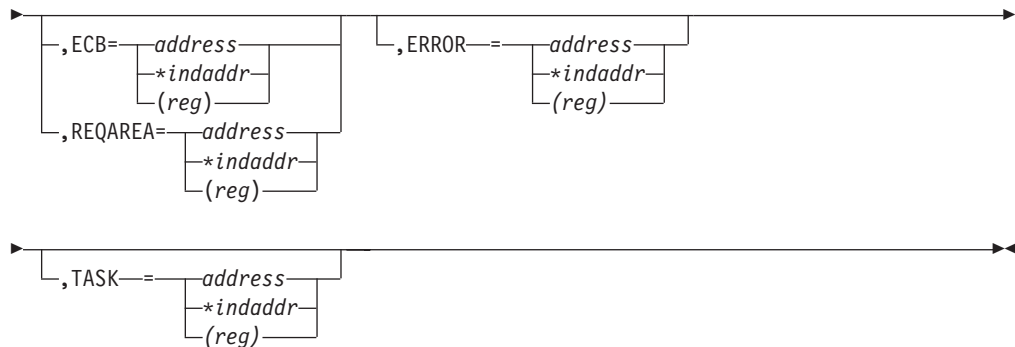
RECV returns the length of the incoming message or data. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard extra bytes.

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 “Environmental restrictions and programming requirements” on page 429.
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 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, "Return codes," on page 781 for information about ERRNO return codes.								
RETCODE	A fullword binary field that returns one of the following: <table> <tr> <th>Value</th><th>Description</th></tr> <tr> <td>0</td><td>A 0 return code indicates that the connection is closed and no data is available.</td></tr> <tr> <td>>0</td><td>A positive value indicates the number of bytes copied into the buffer.</td></tr> <tr> <td>-1</td><td>Check ERRNO for an error code.</td></tr> </table>	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.
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'	1	Receive out-of-band data (stream sockets only).
'MSG_PEEK'	2	Peek at the data, but do not destroy the data.

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

A 100-byte 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 272 or “**IOCTL**” on page 319 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.

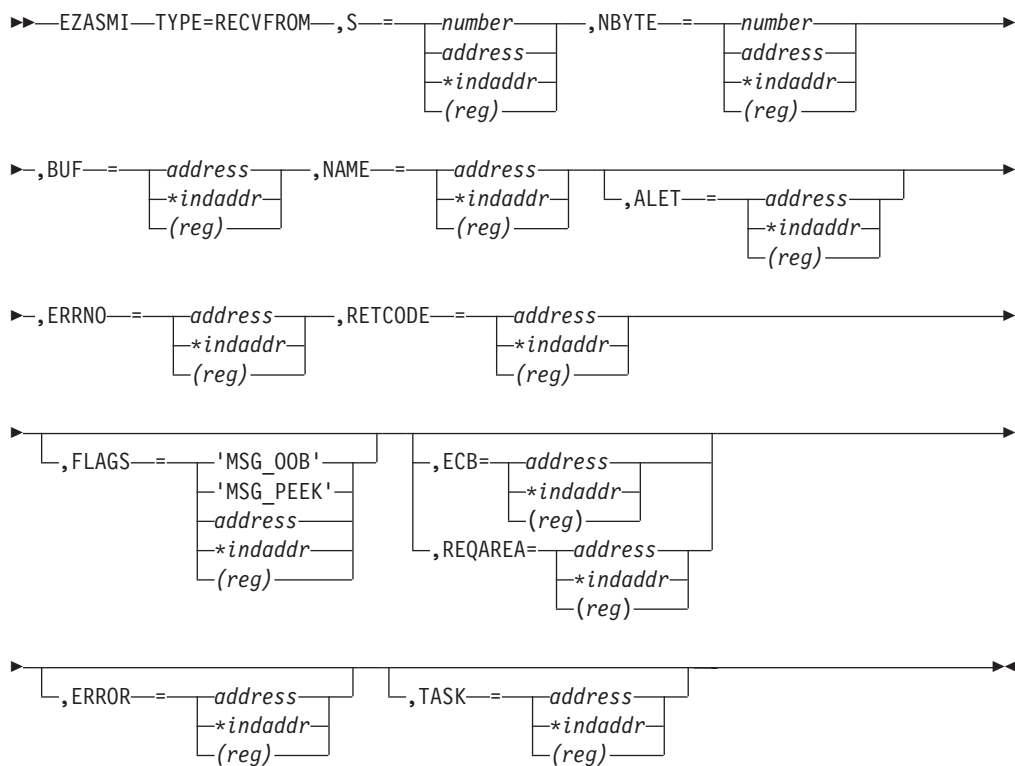
If data is not available and socket **S** is in nonblocking mode, **RECVFROM** returns a -1 and sets **ERRNO** to 35 (**EWOULDBLOCK**). Because **RECVFROM** returns the socket address in the **NAME** structure, it applies to any datagram socket, whether connected or unconnected. See “**FCNTL**” on page 272 or “**IOCTL**” on page 319 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 “Environmental restrictions and programming requirements” on page 429.
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 filled in 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 **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.
- ERRNO** Output parameter. A fullword binary field. If **RETCODE** is negative, this field contains an error number. See Appendix B, "Return codes," on page 781 for information about **ERRNO** return codes.
- RETCODE** A fullword binary field that returns one of the following:
- | 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'	1	Receive out-of-band data. (Stream sockets only.)
'MSG_PEEK'	2	Peek at the data, but do not destroy the data.

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

A 100-byte 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.

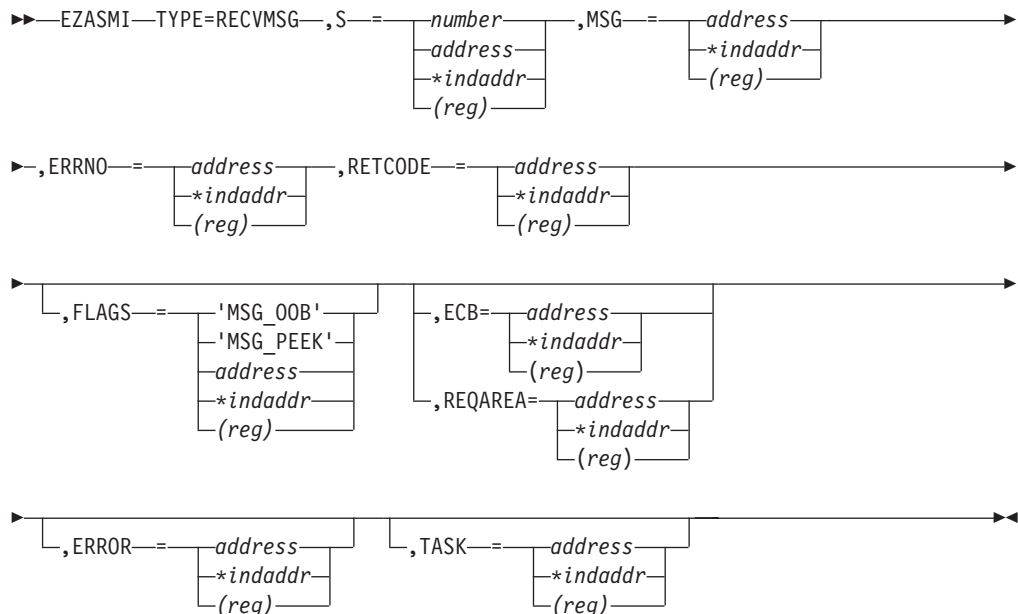
TASK Input parameter. The location of the task storage area in your program.

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 “Environmental restrictions and programming requirements” on page 429.
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. 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'	1	Receive out-of-band data. (Stream sockets only.)
'MSG_PEEK'	2	Peek at the data, but do not destroy the data.

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

A 100-byte 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.

To use the SELECT macro as a timer in your program, do either of the following:

- Set the read, write, and except 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 “Environmental restrictions and programming requirements” on page 429.
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 ()` call 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.

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 `RSNDMSK` 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 `WSNDMSK` 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 ESNDMSK character string and set the appropriate bits to '1'.

Returning the results

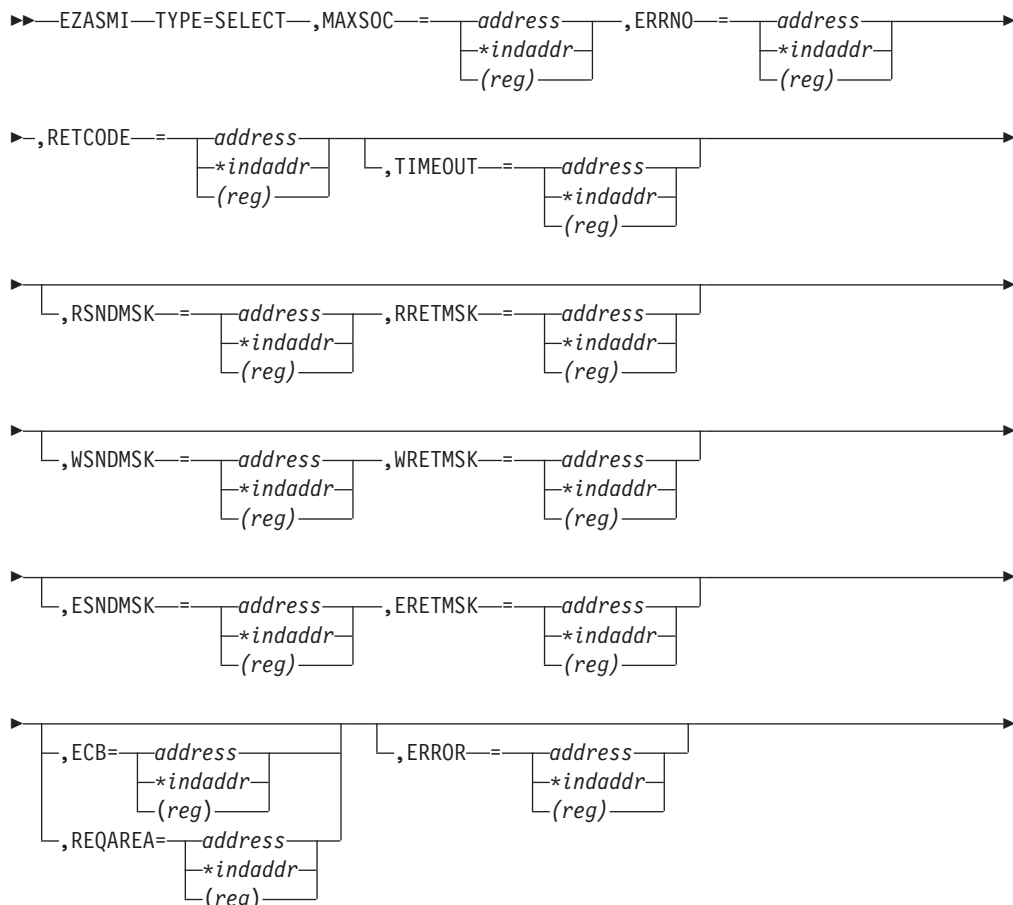
For each event tested by a *x*SNDMSK, a bit string records the results of the check. The bit strings are RRETMSK, WRETMSK, and ERETMSK for read, write, and exceptional events. On return from the SELECT macro, each bit set to '1' in the *x*RETMSK is a read, write, or exceptional event for the associated socket.

MAXSOC parameter

The SELECT call must test each bit in each string before returning any results. For efficiency, the MAXSOC parameter can be set to the largest socket number for any event type. The SELECT call tests only bits in the range 0 through the MAXSOC value.

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 specifying the largest socket descriptor number to be checked plus one. Socket numbering is in the range 0–1999.								
ERRNO	Output parameter. A fullword binary field. If RETCODE is negative, ERRNO contains a valid error number. Otherwise, ignore ERRNO . See Appendix B, “Return codes,” on page 781 for information about ERRNO return codes.								
RETCODE	Output parameter. A fullword binary field that returns one of the following: <table> <tr> <th>Value</th><th>Description</th></tr> <tr> <td>>0</td><td>Indicates the number of ready sockets in the three return masks. Note: If the number of ready sockets is greater than 65 535, only 65 535 is reported.</td></tr> <tr> <td>=0</td><td>Indicates that the SELECT time limit has expired.</td></tr> <tr> <td>–1</td><td>Check ERRNO for an error code.</td></tr> </table>	Value	Description	>0	Indicates the number of ready sockets in the three return masks. Note: If the number of ready sockets is greater than 65 535, only 65 535 is reported.	=0	Indicates that the SELECT time limit has expired.	–1	Check ERRNO for an error code.
Value	Description								
>0	Indicates the number of ready sockets in the three return masks. Note: If the number of ready sockets is greater than 65 535, only 65 535 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 TIMEOUT as follows: <ul style="list-style-type: none"> 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. A bit string sent to request read event status. <ul style="list-style-type: none"> 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 0, the SELECT will not check for read events. 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” on page 45 for more information.

- RRETMSK** Output parameter. A bit string that returns the status of read events.
- For each socket that is ready to read, the corresponding bit in the string will be set to 1.
 - For sockets to be ignored, the corresponding bit in the string will be set to 0.
- WSNDMSK** Input parameter. 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.
- WRETMSK** Output parameter. A bit string that returns the status of write events.
- For each socket that is ready to write, the corresponding bit in the string will be set to 1.
 - For sockets that are not ready to be written, the corresponding bit in the string will be set to 0.
- ESNDMSK** Input parameter. A bit string sent to request exception event status. The length of the string should be equal to the maximum number of sockets to be checked.
- 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.
- ERETMSK** Output parameter. A bit string that returns 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 for which exception status has been set, the corresponding bit will be set to 1.
 - For sockets that do not have exception status, the corresponding bit will be set to 0.

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

A 100-byte 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

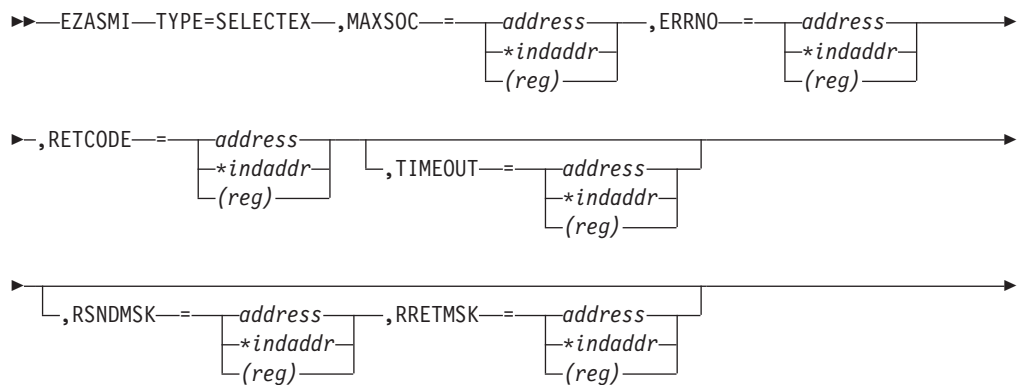
The SELECTEX macro monitors a set of sockets, a time value, and an ECB or list of ECBs. It completes when either one of the sockets has activity, the time value expires, or the ECBs are posted.

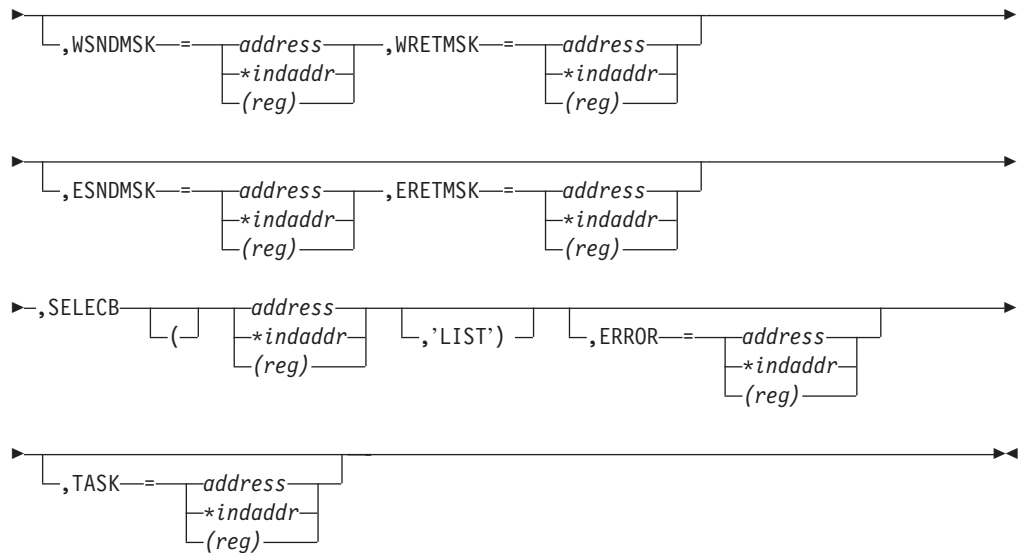
To use the SELECTEX call as a timer in your program, do either of the following:

- Set the read, write, and except 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 “Environmental restrictions and programming requirements” on page 429.
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
MAXSOC	Input parameter. A fullword binary field specifying the largest socket descriptor number being checked.
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 65 535, only 65 353 is reported.
0	Either the SELECTEX time limit has expired (ECB value will be 0) or one of the caller's ECBs has been posted (ECB value will be nonzero and the caller's descriptor sets will be set to 0). The caller must initialize the ECB values to zero before issuing the SELECTEX macro.
-1	Check ERRNO .
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: <ul style="list-style-type: none"> 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 will 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" on page 45 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" on page 45 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 will 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" on page 45 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" on page 45 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 will 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" on page 45 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" on page 45 for more information.
SELECB	<p>Input parameter. An ECB or list of ECB addresses which, if posted, causes completion of the SELECTEX.</p> <p>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.</p> <p>Note: The maximum number of ECBs that can be specified in a list is 1013.</p>
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:

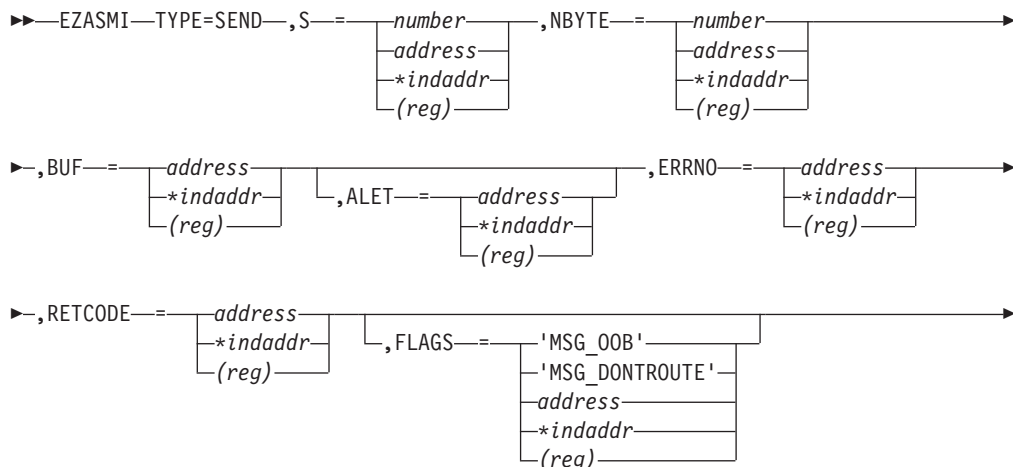
- Send out-of-band data, for example, interrupts, aborts, and data marked urgent. Only stream sockets created in the AF_INET address family support out-of-band data.
- Suppress use of local routing tables. This implies that the caller takes control of routing and writing network software.

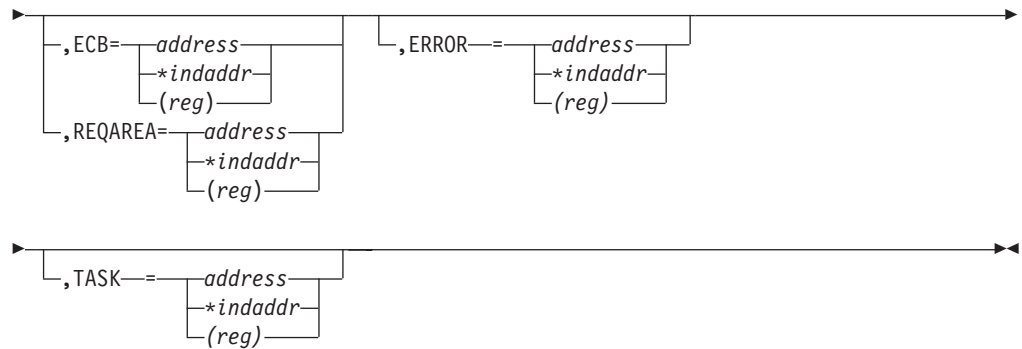
For datagram sockets, SEND transmits the entire datagram if it fits into the receiving buffer. Extra data is discarded.

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 "Environmental restrictions and programming requirements" on page 429.
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 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.
ERRNO	Output parameter. A fullword binary field. If RETCODE is negative, this field contains an error number. See Appendix B, "Return codes," on page 781 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'	1	Send out-of-band data. (Stream sockets only.)
'MSG_DONTROUTE'	4	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

A 100-byte 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

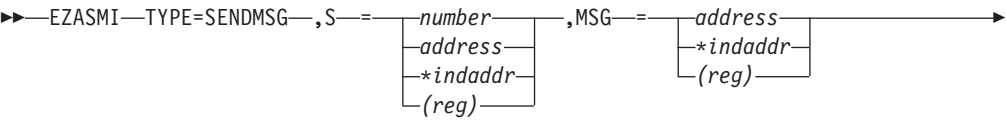
Input parameter. The location of the task storage area in your program.

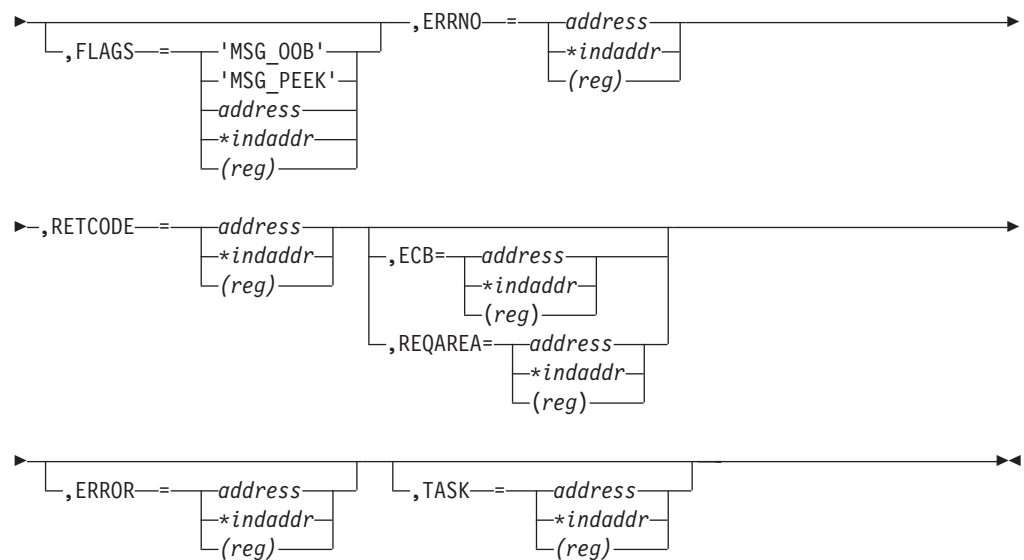
SENDMSG

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 "Environmental restrictions and programming requirements" on page 429.
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.

Field Description

The IPv4 socket address structure contains the following fields:

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 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** may 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 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

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'	1	Send out-of-band data. (Stream sockets only.)
'MSG_DONTROUTE'	4	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

A 100-byte 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:

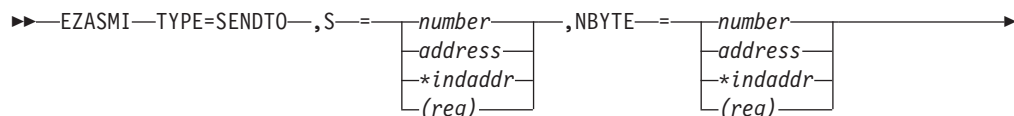
- Send out-of-band data, such as interrupts, aborts, and data marked as urgent.
- Suppress the local routing tables. This implies that the caller takes control of routing, which requires writing network software.

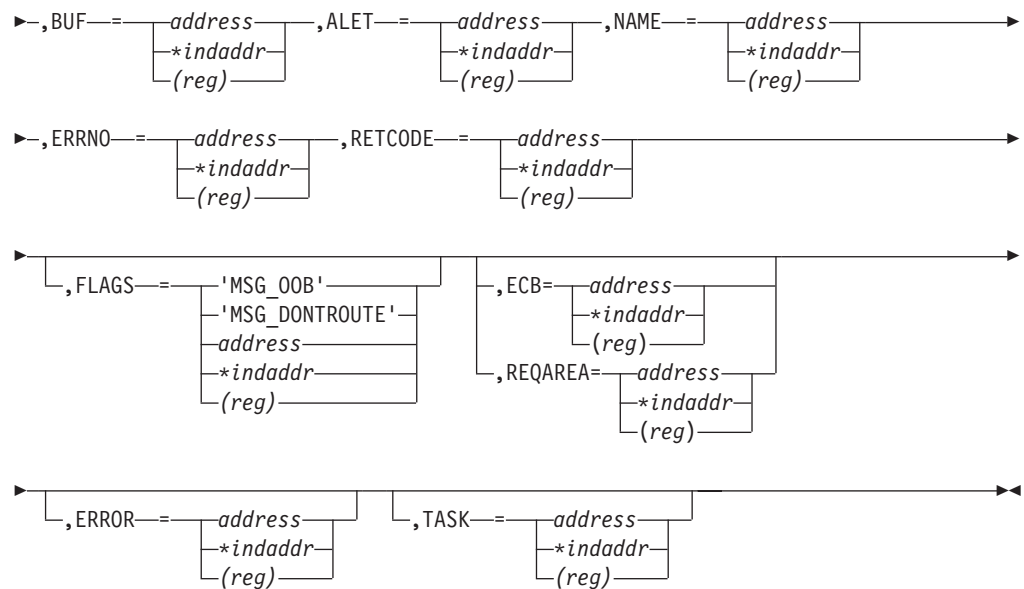
For datagram sockets, the SENDTO macro sends the entire datagram if the datagram fits into the buffer.

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 "Environmental restrictions and programming requirements" on page 429.
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 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.				
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: <table> <tr> <th>Field</th><th>Description</th></tr> <tr> <td>FAMILY</td><td>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.</td></tr> </table>	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 .
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 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 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 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.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, **ERRNO** contains a valid error number. Otherwise, ignore **ERRNO**.

See Appendix B, "Return codes," on page 781 for information about **ERRNO** return codes.

RETCODE

Output parameter. A fullword binary field that returns one of the following:

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'	1	Send out-of-band data. (Stream sockets only.)
'MSG_DONTROUTE'	4	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

A 100-byte 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

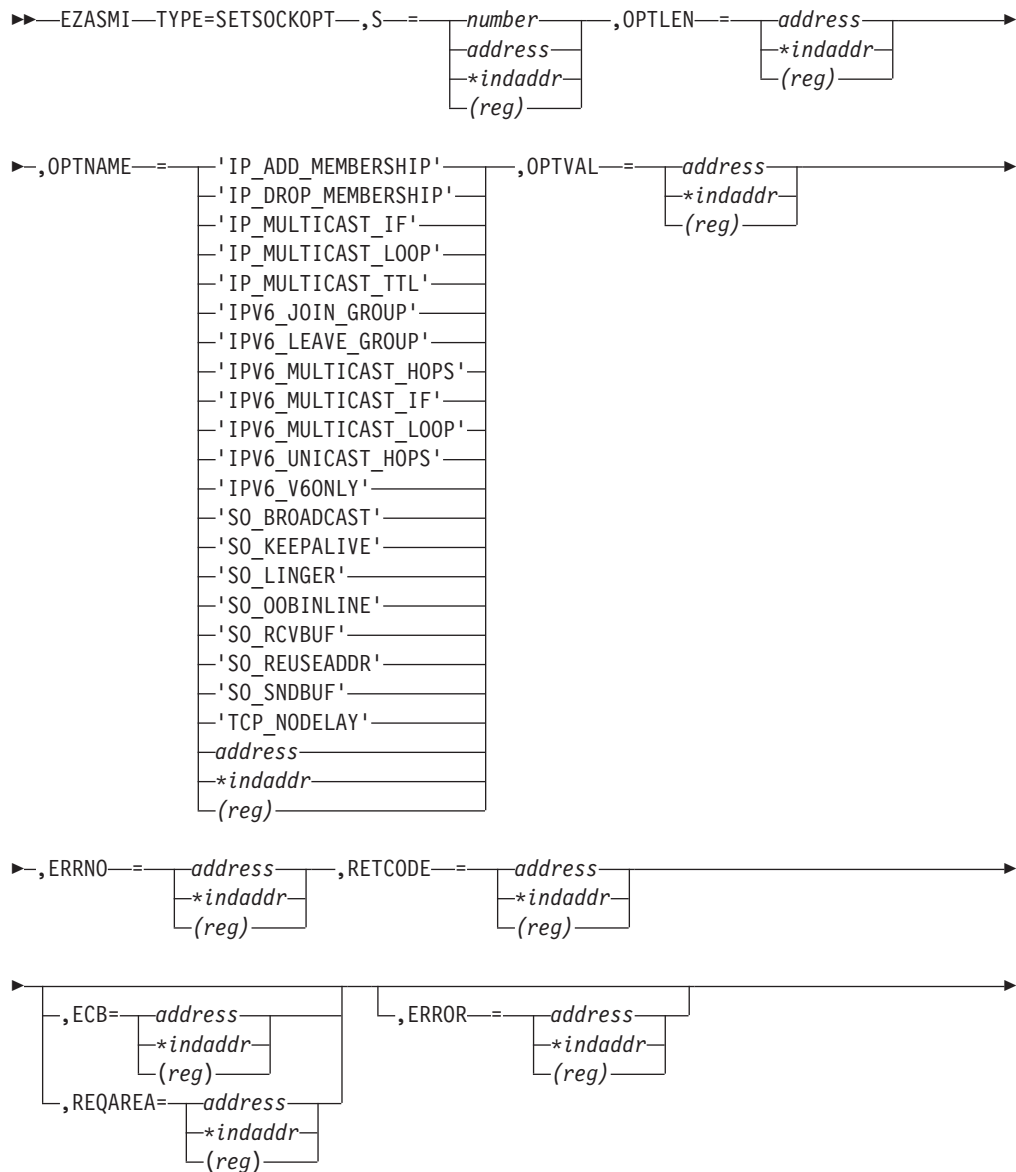
The **SETSOCKOPT** macro sets the options associated with a socket.

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 “Environmental restrictions and programming requirements” on page 429.
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 803 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, "Return codes," on page 781 for information about ERRNO return codes.						
RETCODE	Output parameter. A fullword binary field that returns one of the following: <table> <tr> <th>Value</th><th>Description</th></tr> <tr> <td>0</td><td>Successful call.</td></tr> <tr> <td>-1</td><td>Check ERRNO for an error code.</td></tr> </table>	Value	Description	0	Successful call.	-1	Check ERRNO for an error code.
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: <div> <p>For ECB</p> <p>A 4-byte ECB posted by TCP/IP when the macro completes.</p> <p>For REQAREA</p> <p>A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.</p> <p>For ECB/REQAREA</p> <p>A 100-byte storage field used by the interface to save the state information.</p> <p>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.</p> </div>						
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 18. *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 <i>hlq</i> .SEZAINST(CBLOCK) for the PL/I example of IP_MREQ. The IP_MREQ definition for COBOL: <pre>01 IP-MREQ. 05 IMR-MULTIADDR PIC 9(8) BINARY. 05 IMR-INTERFACE PIC 9(8) BINARY.</pre>	N/A
IP_DROP_MEMBERSHIP Use this option to enable an application to exit a multicast group. 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 <i>hlq</i> .SEZAINST(CBLOCK) for the PL/I example of IP_MREQ. The IP_MREQ definition for COBOL: <pre>01 IP-MREQ. 05 IMR-MULTIADDR PIC 9(8) BINARY. 05 IMR-INTERFACE PIC 9(8) BINARY.</pre>	N/A
IP_MULTICAST_IF Use this option to set or obtain the IPv4 interface address used for sending outbound multicast datagrams from the socket application. This is an IPv4-only socket option. Note: Multicast datagrams can be transmitted only on one interface at a time.	A 4-byte binary field containing an IPv4 interface address.	A 4-byte binary field containing an IPv4 interface address.
IP_MULTICAST_LOOP Use this option to control or determine whether a copy of multicast datagrams are looped back for multicast datagrams sent to a group to which the sending host itself belongs. The default is to loop the datagrams back. This is an IPv4-only socket option.	A 1-byte binary field. To enable, set to 1. To disable, set to 0.	A 1-byte binary field. If enabled, will contain a 1. If disabled, will contain a 0.

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>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 <i>hlq</i>.SEZAINST(CBLOCK) for the PL/I example of IPV6_MREQ.</p> <p>The IPV6_MREQ definition for COBOL:</p> <pre> 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.</pre>	<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 <i>hlq</i>.SEZAINST(CBLOCK) for the PL/I example of IPV6_MREQ.</p> <p>The IPV6_MREQ definition for COBOL:</p> <pre> 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.</pre>	<p>N/A</p>

Table 18. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
IPV6_MULTICAST_HOPS Use to set or obtain the hop limit used for outgoing multicast packets. This is an IPv6-only socket option.	Contains a 4-byte binary value specifying the multicast hops. If not specified, then the default is 1 hop. -1 indicates use stack default. 0 - 255 is the valid hop limit range. 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.	Contains a 4-byte binary value in the range from 0 to 255 indicating the number of multicast hops.
IPV6_MULTICAST_IF Use this option to set or obtain the index of the IPv6 interface used for sending outbound multicast datagrams from the socket application. This is an IPv6-only socket option.	Contains a 4-byte binary field containing an IPv6 interface index number.	Contains a 4-byte binary field containing an IPv6 interface index number.
IPV6_MULTICAST_LOOP 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. This is an IPv6-only socket option.	A 4-byte binary field. To enable, set to 1. To disable, set to 0.	A 4-byte binary field. If enabled, contains a 1. If disabled, contains a 0.
IPV6_UNICAST_HOPS Use this option to set or obtain the hop limit used for outgoing unicast IPv6 packets. This is an IPv6-only socket option.	Contains a 4-byte binary value specifying the unicast hops. If not specified, then the default is 1 hop. -1 indicates use stack default. 0 - 255 is the valid hop limit range. Note: APF authorized applications are permitted to set a hop limit that exceeds the system configured default. CICS applications cannot execute as APF authorized.	Contains a 4-byte binary value in the range from 0 to 255 indicating the number of unicast hops.
IPV6_V6ONLY 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. This is an IPv6-only socket option.	A 4-byte binary field. To enable, set to 1. To disable, set to 0.	A 4-byte binary field. If enabled, contains a 1. If disabled, contains a 0.

Table 18. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
SO_ASCII Use this option to set or determine the translation to ASCII data option. When <code>SO_ASCII</code> is set, data is translated to ASCII. When <code>SO_ASCII</code> is not set, data is not translated to or from ASCII. Note: This is a REXX-only socket option.	To enable, set to ON. To disable, set to OFF. 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.	If enabled, contains ON. If disabled, contains OFF. 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.
SO_BROADCAST 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. Note: This option has no meaning for stream sockets.	A 4-byte binary field. To enable, set to 1 or a positive value. To disable, set to 0.	A 4-byte field. If enabled, contains a 1. If disabled, contains a 0.
SO_DEBUG Use <code>SO_DEBUG</code> 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. Notes: <ol style="list-style-type: none"> 1. This is a REXX-only socket option. 2. This option has meaning only for stream sockets. 	To enable, set to ON. To disable, set to OFF.	If enabled, contains ON. If disabled, contains OFF.
SO_EBCDIC Use this option to set or determine the translation to EBCDIC data option. When <code>SO_EBCDIC</code> is set, data is translated to EBCDIC. When <code>SO_EBCDIC</code> is not set, data is not translated to or from EBCDIC. This option is ignored by EBCDIC hosts. Note: This is a REXX-only socket option.	To enable, set to ON. To disable, set to OFF. 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.	If enabled, contains ON. If disabled, contains OFF. 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.
SO_ERROR 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.	N/A	A 4-byte binary field containing the most recent <code>ERRNO</code> for the socket.

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 keepalive 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 keepalive 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 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 18. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_OOBINLINE</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 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 65 535 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_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, refer to 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 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:</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 65 535 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>
<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>

Table 18. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
TCP_NODELAY Use this option to set or determine whether data sent over the socket is subject to the Nagle algorithm (RFC 896). 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. Note: Use the following to set TCP_NODELAY OPTNAME value for COBOL programs: <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>	A 4-byte binary field. To enable, set to a 0. To disable, set to a 1 or nonzero.	A 4-byte binary field. If enabled, contains a 0. If disabled, contains a 1.

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.

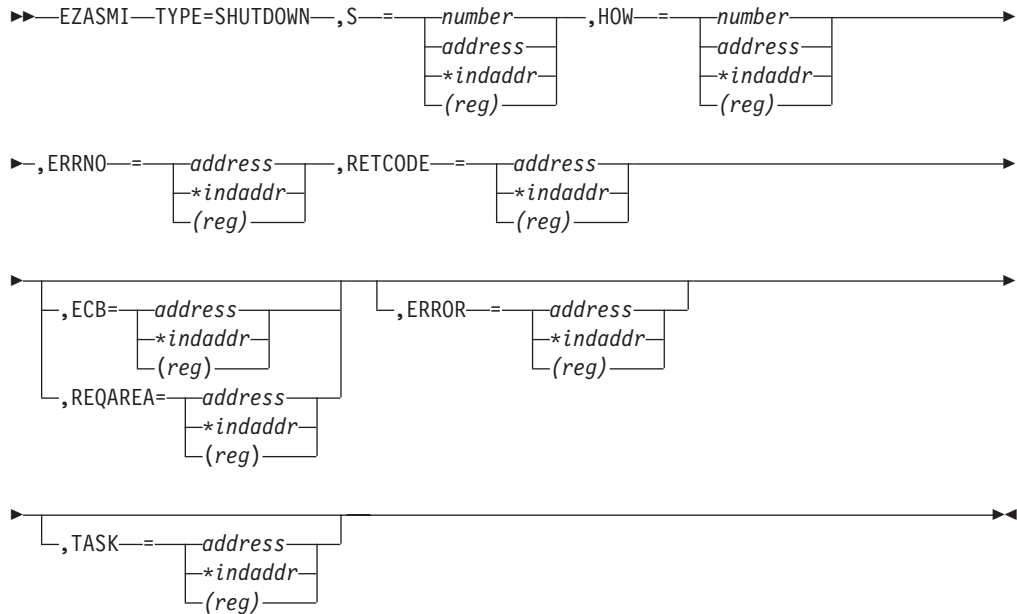
A client program can use the SHUTDOWN macro to reuse a given socket with a different connection.

If you issue SHUTDOWN for a socket that currently has outstanding socket calls pending, see Table 3 on page 37 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 “Environmental restrictions and programming requirements” on page 429.
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. <table border="1"> <thead> <tr> <th>Value</th><th>Description</th></tr> </thead> <tbody> <tr> <td>0</td><td>Ends further receive operations.</td></tr> <tr> <td>1</td><td>Ends further send operations.</td></tr> <tr> <td>2</td><td>Ends further send and receive operations.</td></tr> </tbody> </table>	Value	Description	0	Ends further receive operations.	1	Ends further send operations.	2	Ends further send and receive operations.
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, "Return codes," on page 781 for information about ERRNO return codes.								
RETCODE	Output parameter. A fullword binary field that returns the following: <table border="1"> <thead> <tr> <th>Value</th><th>Description</th></tr> </thead> <tbody> <tr> <td>0</td><td>Successful call.</td></tr> <tr> <td>-1</td><td>Check ERRNO for an error code.</td></tr> </tbody> </table>	Value	Description	0	Successful call.	-1	Check ERRNO for an error code.		
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 <code>APITYPE=3</code> . It points to a 104-byte field containing: <p>For ECB</p> <p>A 4-byte ECB posted by TCP/IP when the macro completes.</p>								

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

A 100-byte 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.

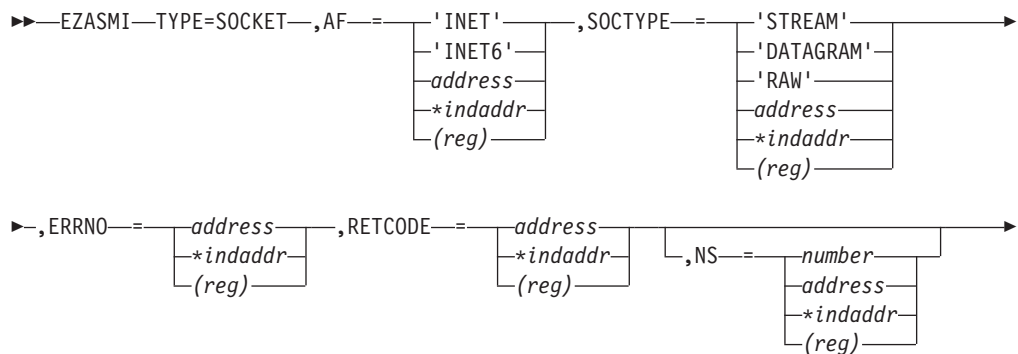
TASK Input parameter. The location of the task storage area in your program.

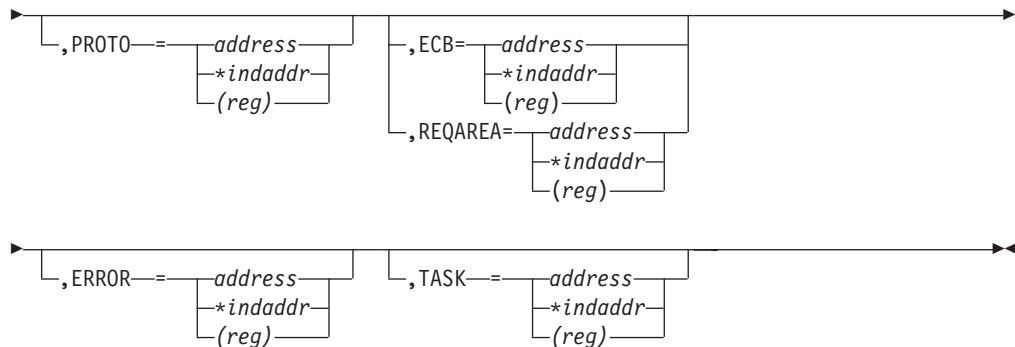
SOCKET

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 “Environmental restrictions and programming requirements” on page 429.
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	<p>Input parameter. Specify one of the following:</p> <table> <tr> <th>Value</th><th>Description</th></tr> <tr> <td>'INET' or a decimal 2</td><td>Indicates the socket being created will use the IPv4 Internet protocol.</td></tr> <tr> <td>'INET6' or decimal 19</td><td>Indicates the socket being created will use the IPv6 Internet protocol.</td></tr> </table> <p>Note: AF can also indicate a fullword binary number specifying the address family.</p>	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.		
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.								
SOCTYPE	<p>Input parameter. A fullword binary field set to the type of socket required. The types are:</p> <table> <tr> <th>Value</th><th>Description</th></tr> <tr> <td>1 or 'STREAM'</td><td>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.</td></tr> <tr> <td>2 or 'DATAGRAM'</td><td>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.</td></tr> <tr> <td>3 or 'RAW'</td><td>Raw sockets provide the interface to internal protocols (such as IP and ICMP).</td></tr> </table> <p>Note: For SOCK_RAW sockets, the application must be APF-authorized.</p>	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).
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).								
ERRNO	Output parameter. A fullword binary field. If RETCODE is negative, this field contains an error number. See Appendix B, "Return codes," on page 781 for information about ERRNO return codes.								
RETCODE	Output parameter. A fullword binary field that returns one of the following:								

	<p>Value Description</p> <p>> or = 0 Contains the new socket descriptor.</p> <p>-1 Check ERRNO for an error code.</p>																						
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	<p>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:</p> <table> <tr> <th>Protocol name</th><th>Numeric value</th></tr> <tr> <td>IPPROTO_HOPOPTS</td><td>0</td></tr> <tr> <td>IPPROTO_TCP</td><td>6</td></tr> <tr> <td>IPPROTO_UDP</td><td>17</td></tr> <tr> <td>IPPROTO_IPV6</td><td>41</td></tr> <tr> <td>IPPROTO_ROUTING</td><td>43</td></tr> <tr> <td>IPPROTO_FRAGMENT</td><td>44</td></tr> <tr> <td>IPPROTO_ESP</td><td>50</td></tr> <tr> <td>IPPROTO_AH</td><td>51</td></tr> <tr> <td>IPPROTO_NONE</td><td>59</td></tr> <tr> <td>IPPROTO_DSTOPTS</td><td>60</td></tr> </table> <p>PROTO numbers are found in the <i>hlq.etc.proto</i> data set.</p>	Protocol name	Numeric value	IPPROTO_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
Protocol name	Numeric value																						
IPPROTO_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																						
ECB or REQAREA	<p>Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:</p> <p>For ECB</p> <p>A 4-byte ECB posted by TCP/IP when the macro completes.</p> <p>For REQAREA</p> <p>A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.</p> <p>For ECB/REQAREA</p> <p>A 100-byte storage field used by the interface to save the state information.</p> <p>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.</p>																						
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

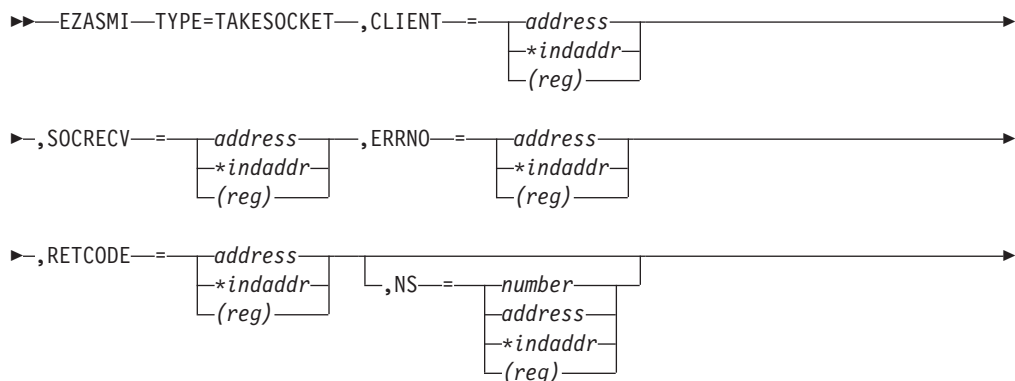
The TAKESOCKET macro acquires a socket from another program and creates a new socket. Typically, a subtask issues this macro using client ID and socket descriptor data that it obtained from the concurrent server.

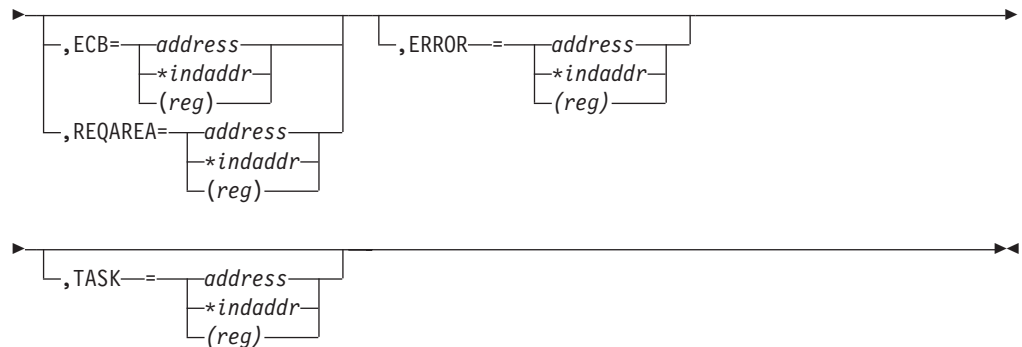
Notes:

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 "Environmental restrictions and programming requirements" on page 429.
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, "Return codes," on page 781 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

A 100-byte 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.

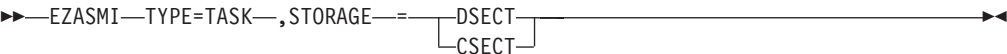
TASK

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 "Environmental restrictions and programming requirements" on page 429.
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
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.

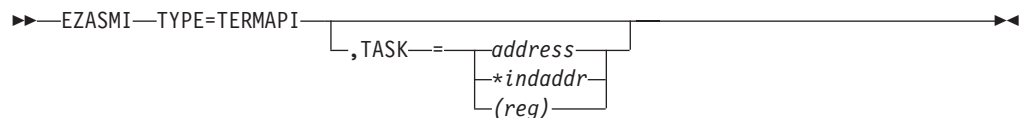
TERMAPI

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 “Environmental restrictions and programming requirements” on page 429.
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.

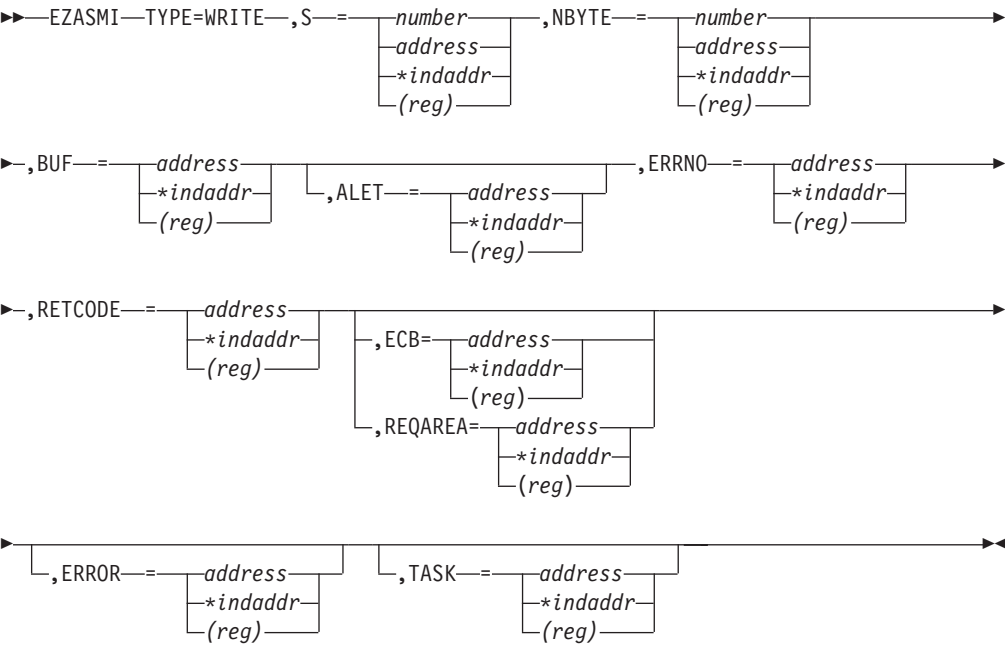
For datagram sockets, this macro writes the entire datagram, if it will fit into one TCP/IP buffer.

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 "Environmental restrictions and programming requirements" on page 429.
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 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, "Return codes," on page 781 for information about ERRNO return codes.								
RETCODE	Output parameter. A fullword binary field.								
	<table> <tr> <th>Value</th><th>Description</th></tr> <tr> <td>>0</td><td>A successful call. The value is set to the number of bytes transmitted.</td></tr> <tr> <td>0</td><td>Connection partner has closed connection.</td></tr> <tr> <td>-1</td><td>Check ERRNO for an error code.</td></tr> </table>	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.
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: <table> <tr> <td>For ECB</td><td>A 4-byte ECB posted by TCP/IP when the macro completes.</td></tr> <tr> <td>For REQAREA</td><td>A 4-byte user token (set by you) that is presented to your exit when the response to this function request is complete.</td></tr> <tr> <td>For ECB/REQAREA</td><td>A 100-byte storage field used by the interface to save the state information.</td></tr> </table> <p>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.</p>	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	A 100-byte storage field used by the interface to save the state 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	A 100-byte storage field used by the interface to save the state information.								
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.								

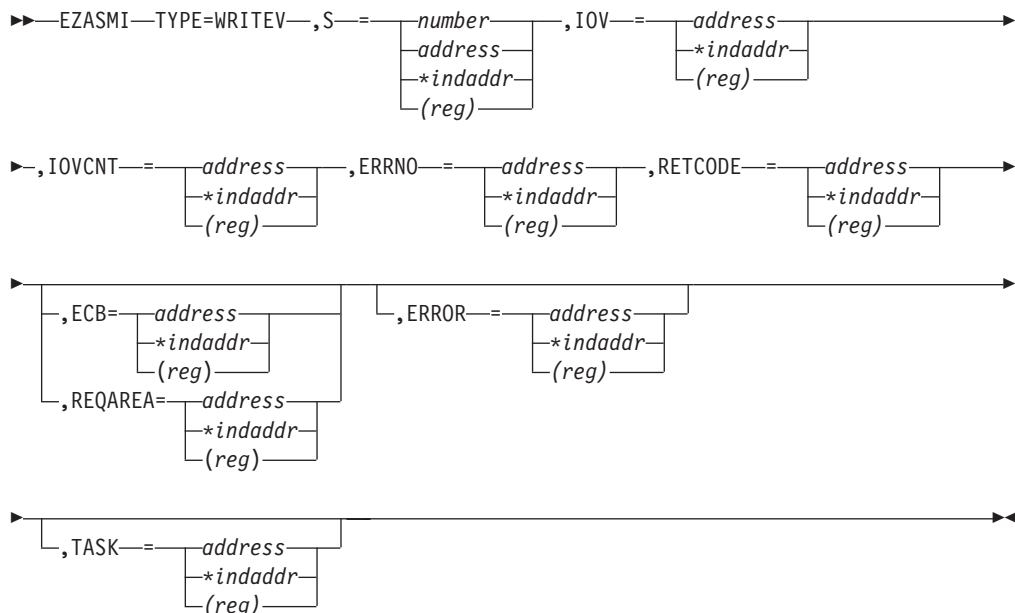
This macro writes up to **NBYTE** bytes of data. If there is not enough available buffer space for the socket data to be transmitted, and the socket is in blocking mode, WRITE blocks the caller until additional buffer space is available. If the socket is in nonblocking mode, WRITE returns a -1 and sets **ERRNO** to 35 (EWOULDBLOCK). See "FCNTL" on page 272 or "IOCTL" on page 319 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 “Environmental restrictions and programming requirements” on page 429.
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 <code>IOVCNT</code> and the format of the structures as follows: <div> <p>Fullword 1 Input parameter. The address of a data buffer.</p> <p>Fullword 2 Input parameter. The ALET for this buffer. If the buffer is in the primary address space, this should be zeros.</p> <p>If a nonzero ALET is specified, the ALET must represent a valid entry in the dispatchable unit access list (DU-AL) for</p> </div>

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 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

A 100-byte 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 section provides sample programs for the macro interface that you can use for assembler language applications. The source code can be found in the *hlq.SEZAINST* data set.

The following sample programs are included:

Program	Description
EZASOKAS	Sample IPv4 macro interface server program

Program	Description
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

```

Figure 61. EZASOKAS sample server program for IPv4 (Part 1 of 10)

```

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,1      MOVE A 1 FOR ASYNC
      MVC  TYPE,INITAPI    MOVE 'INITAPI' TO MESSAGE
*
      EZASMI TYPE=INITAPI, Issue INITAPI Macro
X

```

Figure 61. EZASOKAS sample server program for IPv4 (Part 2 of 10)

```

        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          X
        S=S,                STREAM                X

```

Figure 61. EZASOKAS sample server program for IPv4 (Part 3 of 10)

```

NAME=NAME,          (SOCKET NAME STRUCTURE)          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 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
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 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                      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,    IN CASE WE ARE DOING EXITS OR ECBS X
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                      X
S=SOCDESCA,          ACCEPT SOCKET                    X
NBYTE=NBYTE,        SIZE OF BUFFER                   X
BUF=BUF,             (BUFFER)                         X
ERRNO=ERRNO,        (Specify ERRNO field)             X

```

Figure 61. EZASOKAS sample server program for IPv4 (Part 4 of 10)

```

        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
        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)
        REQAREA=REQAREA,       IN CASE WE ARE DOING EXITS OR ECBS
        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
        S=SOCDESCA,            ACCEPT
        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
*
        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

```

Figure 61. EZASOKAS sample server program for IPv4 (Part 5 of 10)

```

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
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'
NOWAIT WAIT ECB=ECB
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 *
*
```

Figure 61. EZASOKAS sample server program for IPv4 (Part 6 of 10)


```

*-----*
*      FORMAT THE RETCODE= -XXXXXX ERRNO= XXXXXX MSG RESULTS
*      ***> R6 = RETCODE VALUE ON ENTRY
*-----*

NOTM   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   *
        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) 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

```

Figure 61. EZASOKAS sample server program for IPv4 (Part 7 of 10)

```

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'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 *
*-----*
        CNOP 0,4
NAME    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'

```

Figure 61. EZASOKAS sample server program for IPv4 (Part 8 of 10)

```

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
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'
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'

```

Figure 61. EZASOKAS sample server program for IPv4 (Part 9 of 10)

```

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 61. EZASOKAS sample server program for IPv4 (Part 10 of 10)

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

```

Figure 62. EZASOKAC sample client program for IPv4 (Part 1 of 10)

```

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,INITAPI     MOVE 'INITAPI' TO MESSAGE
*
      POST ECB,1            FOLLOWING IS SYNC ONLY
      MVI  SYNFLAG,0        MOVE A 1 FOR ASYNCH

```

Figure 62. EZASOKAC sample client program for IPv4 (Part 2 of 10)

```

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,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,    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

```

Figure 62. EZASOKAC sample client program for IPv4 (Part 3 of 10)

```

        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                        X
        S=S,                STREAM 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,    FOR EXITS (AND ECBS)                X
        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
*****

```

Figure 62. EZASOKAC sample client program for IPv4 (Part 4 of 10)


```

*
*      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!
*
*-----*

```

Figure 62. EZASOKAC sample client program for IPv4 (Part 5 of 10)

```

*      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'
      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 *
*-----*

```

Figure 62. EZASOKAC sample client program for IPv4 (Part 6 of 10)

```

*      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
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 *
      C     R6,=F'-1'              VALID HOSTID MAY LOOK LIKE NEG. RC
                                      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

```

Figure 62. EZASOKAC sample client program for IPv4 (Part 7 of 10)

```

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'

```

Figure 62. EZASOKAC sample client program for IPv4 (Part 8 of 10)

```

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
          DC    CL16'0123456789ABCDEF'
PARMADDR  DC    A(0)          PARM ADDRESS SAVE AREA
          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'
MYNEXT    DC    A(MYCB)      NEXT IN CHAIN FOR MULTIPLES

```

Figure 62. EZASOKAC sample client program for IPv4 (Part 9 of 10)

```

        CNOP    0,8
MYLEN    EQU    *-MYCB
MYCB2    EQU    *
        ORG     **MYLEN
        CNOP    0,8
        DC      CL8'&SYSDATE'
        DC      CL8'&SYSTIME'
        END

```

Figure 62. EZASOKAC sample client program for IPv4 (Part 10 of 10)

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

```

Figure 63. EZAS06AS sample server program for IPv6 (Part 1 of 13)

```

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,0      MOVE A 1 FOR ASYNC
      MVC  TYPE,INITAPI    MOVE 'INITAPI' TO MESSAGE
*
      EZASMI TYPE=INITAPI, Issue INITAPI Macro
X

```

Figure 63. EZASO6AS sample server program for IPv6 (Part 2 of 13)


```

        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
*      ASYNC=('EXIT',MYEXIT), (SPECIFY AN EXIT)                  X
*      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='INET6',          INET, IUCV, INET6              X
        SOCTYPE='STREAM',    STREAM(TCP) DATAGRAM(UDP) or RAW 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
*
*-----*
*      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              X
        AF='INET6',          X
        SRCADDR=PRESENTABLE_ADDR, X
        SRCLEN=PRESENTABLE_ADDR_LEN, X
        DSTADDR=NUMERIC_ADDR, 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

```

Figure 63. EZASO6AS sample server program for IPv6 (Part 3 of 13)

```

MVC ADDRESS,NUMERIC_ADDR
*****
*
* ISSUE GETHOSTNAME CALL
*
*****
MVC TYPE,MGHOSTN 'GETHOSTN' TO MESSAGE
EZASMI TYPE=GETHOSTNAME, X
NAMELEN=HOSTNAME, LENGTH OF HOST NAME FIELD X
NAME=HOSTNAME, HOST NAME 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,=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,SERVNAMEL SAVE THE SERVICE NAME LENGTH
*
EZASMI TYPE=GETADDRINFO, ISSUE GETADDRINFO MACRO X
NODE=NODENAME, NODE GETTING INFORMATION FOR X
NODELEN=NODENAME, LENGTH OF NODE NAME X
SERVICE=SERVNAME, SERVICE GETTING INFORMATION FOR X
SERVLEN=SERVNAME, LENGTH OF SERVICE NAME X
HINTS=HINTS, HINTS FOR FILTERING X
RES=RESULT_ADDRINFO, RETURNED ADDRESS INFORMATION X
CANNLEN=CANNAMEL, LENGTH OF CANONICAL NAME 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
*
* 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.
*

```

Figure 63. EZASO6AS sample server program for IPv6 (Part 4 of 13)

```

        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
*

```

Figure 63. EZASO6AS sample server program for IPv6 (Part 5 of 13)

```

      B      ENDAIS
NOAIS  DS      0H
      XC      MSGRSLT2,MSGRSLT2
      MVC      MSGRSLT2(21),=C'Result not returned. '
      BAL      R14,WTOSUB          SEND TO THE CONSOLE
END AIS DS      0H
*
*****
*
*      Issue BIND socket
*
*****
      MVC      TYPE,MBIND          MOVE 'BIND' TO MESSAGE
      MVC      PORT(2),PORTS      Load STREAM port #
*
      EZASMI TYPE=BIND,          Issue Macro
      S=S,          STREAM
      NAME=NAME,          (SOCKET NAME STRUCTURE)
      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  X
*
      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)
      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
      S=S,          STREAM
      NAME=NAME,          (SOCKET NAME STRUCTURE)
      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  X

```

Figure 63. EZASO6AS sample server program for IPv6 (Part 6 of 13)

```

*
*      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
*      AF='INET6',
*      SRCADDR=NUMERIC_ADDR,
*      DSTADDR=PRESENTABLE_ADDR,
*      DSTLEN=PRESENTABLE_ADDR_LEN,
*      ERRNO=ERRNO,          (SPECIFY ERRNO FIELD)
*      RETCODE=RETCODE,      (SPECIFY RETCODE FIELD)
*      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
*      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
*      MVC   MSGRSLT1,MSGBUFF
*      MVC   MSGRSLT2,BUF
*      BAL   R14,WTOSUB      --> PRINT IT

```

Figure 63. EZASO6AS sample server program for IPv6 (Part 7 of 13)

```

*
*
*****
*
*      Issue WRITE - Write data from buffer
*
*****
MVC   TYPE,MWRITE      MOVE 'WRITE ' TO MESSAGE
*
EZASMI TYPE=WRITE,      Issue Macro
      S=SOCDSCA,        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  X
*
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=SOCDSCA,        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  X
*
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 Termap
*-----*
* Message RESULTS text
MVC   MSGRSLT2,BLANK35
*
BAL   R14,RCHECK       --> CHECK RC
*-----*
*      Issue console message for task termination
*-----*
MVC   TYPE,MSGEND       Move 'ENDED' to message
*

```

Figure 63. EZASO6AS sample server program for IPv6 (Part 8 of 13)

```

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,='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
CNOF 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'
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 -->

```

Figure 63. EZASO6AS sample server program for IPv6 (Part 9 of 13)

```

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   *
          C     R6,=F'-1'          VALID HOSTID MAY LOOK LIKE NEG. RC
                                   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
*-----*

```

Figure 63. EZASO6AS sample server program for IPv6 (Part 10 of 13)


```

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' '          NAME OF TCP TO WHICH CONNECTING
          DC    CL8' '            MY ADDR SPACE NAME
          DC    CL8'SOC401CB'
*-----*
*  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'          SCOPEID
          DC    XL4'00'
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 '

```

Figure 63. EZAS06AS sample server program for IPv6 (Part 11 of 13)

```

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

```

Figure 63. EZAS06AS sample server program for IPv6 (Part 12 of 13)

```

        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 PTOP
*
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
HOSTNAMEL DC   AL4(L'HOSTNAME) LENGTH OF THE HOST NAME
NODENAMEL DC   AL4(L'NODENAME) LENGTH OF THE NODE NAME
SERVNAMEL DC   F'0'          LENGTH OF THE SERVICE NAME
RESULT_ADDRINFO DC F'0'      RETURNED ADDRINFO
CANNAMEL 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 63. EZASO6AS sample server program for IPv6 (Part 13 of 13)

EZASO6AC sample client program for IPv6

The EZASO6AC 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

```

Figure 64. EZAS06AC sample client program for IPv6 (Part 1 of 10)

```

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,INITAPI    MOVE 'INITAPI' TO MESSAGE
*
      POST ECB,1           FOLLOWING IS SYNC ONLY
      MVI  SYNFLAG,0       MOVE A 1 FOR ASYNCH

```

Figure 64. EZASO6AC sample client program for IPv6 (Part 2 of 10)

```

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,MSOCKET     MOVE 'SOCKET' TO MESSAGE
*
      EZASMI TYPE=SOCKET,    Issue SOCKET Macro          X
      AF='INET6',          INET, IUCV, or INET6          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 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 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
*****
*

```

Figure 64. EZASO6AC sample client program for IPv6 (Part 3 of 10)

```

*      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 GETNAMEINFO MACRO                                *
*                                                                 *
*****
MVC   TYPE,MGNAMEI      MOVE 'GETNAMEINFO' TO MESSAGE
LA    R6,NI_NAMEREQD
ST    R6,FLAGS
*
EZASMI TYPE=GETNAMEINFO, ISSUE GETNAMEINFO MACRO          X
      NAME=NAME,                X
      NAMELEN=NAMELEN,          X
      HOST=HOSTNAME,            X
      HOSTLEN=HOSTNAME,        X
      SERVICE=SERVNAME,        X
      SERVLN=SERVNAME,        X
      FLAGS=FLAGS,              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 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                  *

```

Figure 64. EZASO6AC sample client program for IPv6 (Part 4 of 10)

```

*
*****
MVC  TYPE,MWRITE      MOVE 'WRITE ' TO MESSAGE
*
EZASMI TYPE=WRITE,      Issue Macro          X
      S=S,              STREAM 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,  FOR EXITS (AND ECBS)  X
      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          X
      S=S,              STREAM SOCKET        X
      NBYTE=NBYTE,      SIZE OF BUFFER       X
      BUF=BUF2,         (BUFFER)             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
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

```

Figure 64. EZASO6AC sample client program for IPv6 (Part 5 of 10)


```

*****
*
*      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 Termapl
*
BAL    R14,RCCHECK        --> 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,='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
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
*
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 *

```

Figure 64. EZASO6AC sample client program for IPv6 (Part 6 of 10)

```

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= 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
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
*
*

```

Figure 64. EZASO6AC sample client program for IPv6 (Part 7 of 10)

```

        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      ERROFMT
*
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 *
*-----*
        CNOP   0,4
NAME     DC     0CL28' '        SOCKET IPV6 NAME STRUCTURE

```

Figure 64. EZAS06AC sample client program for IPv6 (Part 8 of 10)

```

        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

*-----*

```

Figure 64. EZAS06AC sample client program for IPv6 (Part 9 of 10)

```

* 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
*-----*
*
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'
*
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 64. EZASO6AC sample client program for IPv6 (Part 10 of 10)

Chapter 13. Using the CALL instruction application programming interface (API)

This chapter describes the CALL instruction API for IPv4 or IPv6 socket applications. The following topics are included:

- “Environmental restrictions and programming requirements”
- “CALL instruction application programming interface (API)” on page 431
- “Understanding COBOL, Assembler, and PL/I call formats” on page 431
- “Converting parameter descriptions” on page 432
- “Diagnosing problems in applications using the CALL instruction API” on page 433
- “Error messages and return codes” on page 433
- “Code CALL instructions” on page 433
- “Using data translation programs for socket call interface” on page 548
- “Call interface sample programs” on page 566

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 may only be invoked in TCB mode (task mode).
Cross-memory mode	These APIs may 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.
INITAPI/TERMAPI macros	The INITAPI/TERMAPI macros 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 EZASOKET macro call that is issued after an EZASOKET TYPE=INITAPI with the ASYNC=('ECB') option has been issued.
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.

Function	Restriction
Addressability mode (Amode) considerations	The EZASOKET macro API may 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 UNIX System Services	Address spaces using the EZASOKET API should not use any UNIX System Services facilities such as EZASOKET API with UNIX System Service's BXP1 Assembler Callable Services or EZASOKET API with Language Environment for OS/390 C/C++ API Services. Doing so can yield unpredictable results.

Linkage conventions for the 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** For synchronous calls, it contains the entry point address of EZBSOH03

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.

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 section describes the CALL instruction API for TCP/IP application programs written in the COBOL, PL/1, or System/370 Assembler language. The format and parameters are described for each socket call.

Notes:

1. Unless your program is running in a CICS environment, reentrant code and multithread applications are not supported by this interface.
2. Only one copy of an interface can exist in a single address space.
3. For a PL/1 program, include the following statement before your first call instruction.

```
DCL EZASOKET ENTRY OPTIONS(RETCODE,ASM,INTER) EXT;
```
4. The entry point for the CICS Sockets Extended module (EZASOKET) is within the EZACICAL module. Therefore EZACICAL should be included explicitly in your link-edit JCL. If not included, you could experience problems, such as the CICS region waiting for the socket calls to complete.

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 is 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 is the 'EZASOKET' call format for assembler language programs.

```
▶▶—CALL EZASOKET, (SOC-FUNCTION,—parm1, parm2, ..—ERRNO,RETCODE),VL—▶▶
```

PL/I language call format

The following is 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 `tcpperror()` 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 chapter 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 65 shows examples of storage definition statements for COBOL, PL/I, and assembler language programs.

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 65. 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 on the SOCKAPI trace option, refer to *z/OS Communications Server: IP Diagnosis Guide*.

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, "Return codes," on page 781.

Code CALL instructions

This section 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 “Environmental restrictions and programming requirements” on page 429.
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 66 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 66. ACCEPT call instructions example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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 Internet 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 Internet 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, "Return codes," on page 781 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 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 only interested in receiving new TCP connections or UDP datagrams that are targeted towards 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.

Note: Even if an application specifies a value of 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. This has a similar effect to the application specifying an explicit IP address on the BIND CALL. For more information, refer to 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 "Environmental restrictions and programming requirements" on page 429.
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 67 on page 437 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 67. BIND call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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

Refer to Chapter 3, “Designing an iterative server program,” on page 27 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: The application can call the GETSOCKNAME macro after the BIND macro to discover the assigned port number.

IP-ADDRESS

A fullword binary field that is set to the 32-bit IPv4 Internet 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: The application can call the GETSOCKNAME macro after the BIND macro to discover the assigned port number.

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 Internet 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, "Return codes," on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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 539 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 530.
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 “Environmental restrictions and programming requirements” on page 429.
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 68 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 68. CLOSE call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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, "Return codes," on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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 512.

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 “Environmental restrictions and programming requirements” on page 429.
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 69 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 69. CONNECT call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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 Internet address of the server's host machine in network byte order. For example, if the Internet address is 129.4.5.12 in dotted decimal notation, it would be represented as '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 Internet address of the server's host machine in network byte order. For example, if the IPv6 Internet 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, “Return codes,” on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

Value	Description
0	Successful call.
-1	Check 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 487 for another way to control a socket’s blocking mode.

Values for commands that are supported by the UNIX Systems Services fcntl callable service will also be accepted. Refer to *OpenEdition® MVS 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 “Environmental restrictions and programming requirements” on page 429.
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 70 on page 444 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 70. FCNTL call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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, “Return codes,” on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following.

- 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 “Environmental restrictions and programming requirements” on page 429.
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 71 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 71. **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, “Return codes,” on page 781 for information about **ERRNO** return codes.

RETCODE Output parameter. A fullword binary field that returns one of the following:

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 “Environmental restrictions and programming requirements” on page 429.
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 447 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 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 9(8)  BINARY.
03 FILLER         PIC 9(8)  BINARY.
03 FILLER         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 CANONNAME      USAGE IS POINTER.
03 NAME           USAGE IS POINTER.
03 NEXT           USAGE IS POINTER.

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 72. 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.

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 SERVLLEN. The SERVICE name being queried will consist of up to SERVLLEN or up to the first binary 0.
SERVLLEN	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 to enable your assembler 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 of the bitwise, OR of one or more of the following:

AI-PASSIVE (X'00000001') or a decimal value of 1.

- Specifies how to fill in 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 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 (127.0.0.0) 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 a decimal value of 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 a decimal value of 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 a decimal value of 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 a decimal value of 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. When the **AI-ALL** flag is not also specified, if no IPv6 addresses are found, a query is made for IPv4 addresses. If IPv4 addresses are found, they are returned as IPv4-mapped IPv6 addresses.
- If the **AF** field does not have the value of **AF_INET6** or the **AF** field contains **AF_UNSPEC** but IPv6 is not supported on the system, this flag is ignored.

AI-ALL (X'00000020') or a decimal value of 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). When the **AF** field has a value of **AF_UNSPEC** when the system supports IPv6 and **AI-ALL** is set, the caller accepts IPv6 addresses and either IPv4 address (if **AI-V4MAPPED** is not set), or IPv4-mapped IPv6 addresses (if **AI-V4MAPPED** is set). A query is first made for IPv6 addresses and if successful, the IPv6 addresses are returned. Another query is then made for IPv4 addresses, and any IPv4

addresses found are returned as either IPv4-mapped IPv6 addresses (if AI-V4MAPPED is also specified), or as IPv4 addresses (if AI-V4MAPPED is not specified).

- If the AF field does not have the value of AF_INET6 or does not have the value of AF_UNSPEC when the system supports IPv6, this flag is ignored.

AI-ADDRCONFIG (X'00000040') or a decimal value of

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 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.

Note: 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 are the acceptable socket types:

Type name	Decimal value	Description
SOCK_STREAM	1	for stream socket
SOCK_DGRAM	2	for datagram socket

Type name	Decimal value	Description
SOCK_RAW	3	for raw-protocol interface

Anything else will fail with return code EAI_SOCKTYPE. Note that although SOCK_RAW will be accepted, it will only be valid 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 provision exists 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* means one of the following:

- 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 are 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 provision exists 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* means one of the following:

- 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. On input, this field must be 0.

CANONNAME A fullword binary field. On input, this field must be 0.

NAME A fullword binary field. On input, this field must be 0.

NEXT A fullword binary field. On input, this field must be 0.

RES Initially a fullword binary field. On a successful return, this field will contain a pointer to an addrinfo structure. The addrinfo storage will be allocated in the caller's key. This pointer will also be used as input to the FREEADDRINFO call which must be used to free storage obtained by this call.

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. The length of the NAME socket address structure. The value returned in this field may be used as the arguments for the CONNECT or BIND call with such a socket, according to the AI-PASSIVE flag.

CANONNAME

A fullword binary field. 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 will contain the length of the returned canonical name.

NAME

A fullword binary field. The address of the returned socket address structure. The value returned in this field may be used as the arguments for the CONNECT or BIND 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.

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, "Return codes," on page 781 for information about ERRNO return codes.

RETCODE Output parameter. A fullword binary field that returns one of the following:

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/1 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 482 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 “Environmental restrictions and programming requirements” on page 429.
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 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 73. GETCLIENTID call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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, "Return codes," on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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 Internet address is specified in the call. A given TCP/IP host can have multiple alias names and multiple host IPv4 Internet addresses. The address resolution attempted depends on how the resolver is configured and if any local host tables exist. Refer to *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 "Environmental restrictions and programming requirements" on page 429.
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 456 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 74. GETHOSTBYADDR call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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 Internet address (specified in network byte order) of the host whose name is being sought. See Appendix B, “Return codes,” on page 781 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:

Value	Description
0	Successful call.
-1	Check ERRNO for an error code.

GETHOSTBYADDR returns the **HOSTENT** structure shown in Figure 75 on page 457.

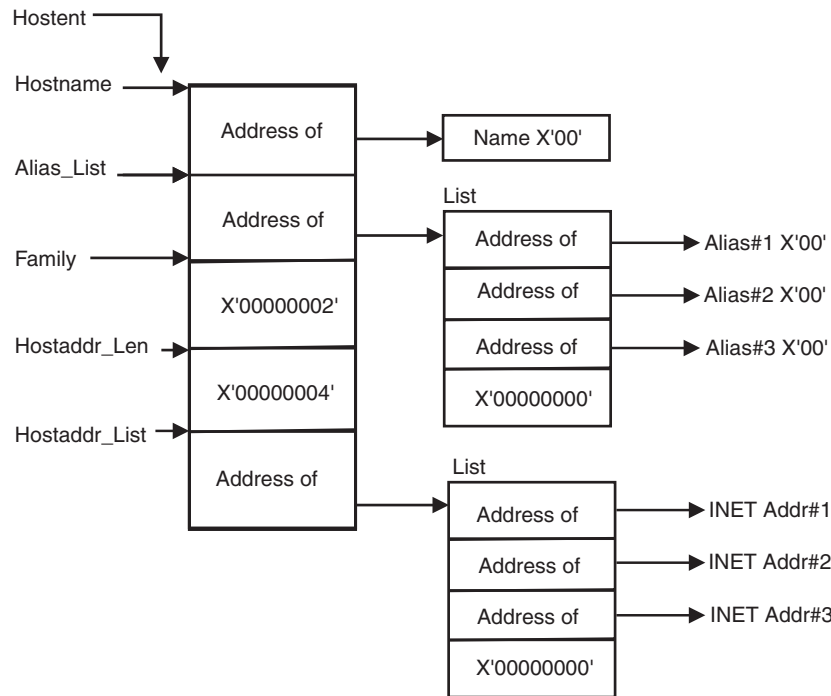


Figure 75. HOSTENT structure returned by the GETHOSTBYADDR call

GETHOSTBYADDR returns the HOSTENT structure shown in figure Figure 75. 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 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 Internet 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 Internet 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 Internet 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 554.

GETHOSTBYNAME

The GETHOSTBYNAME call returns the alias name and the IPv4 Internet 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 Internet addresses.

The name resolution attempted depends on how the resolver is configured and if any local host tables exist. Refer to *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 “Environmental restrictions and programming requirements” on page 429.
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 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 76. GETHOSTBYNAME call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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:

Value	Description
0	Successful call.
-1	An error occurred.

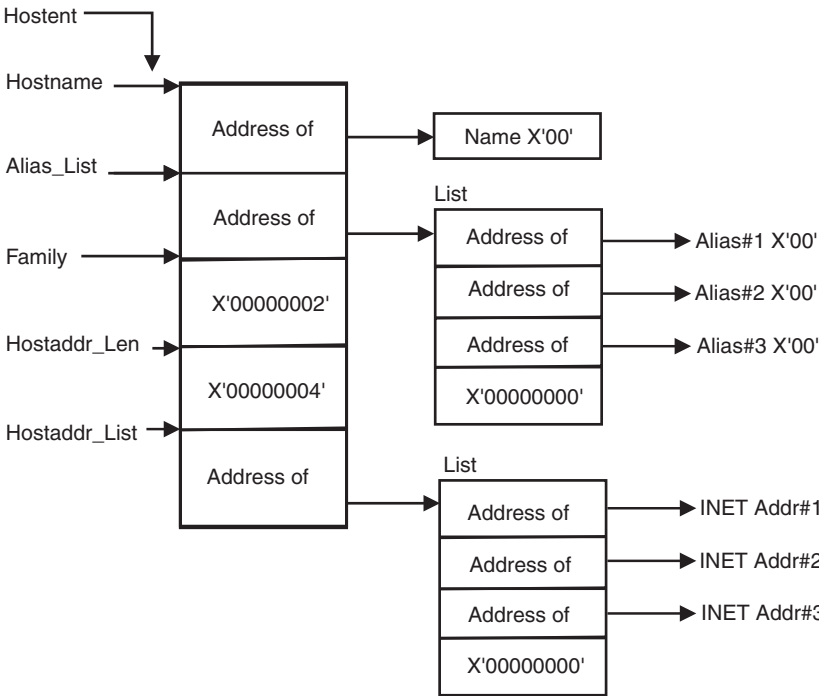


Figure 77. HOSTENT structure returned by the GETHOSTYBYNAME call

GETHOSTBYNAME returns the HOSTENT structure shown in Figure 77. 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 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 Internet 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 Internet 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 Internet 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 554.

GETHOSTID

The GETHOSTID call returns the 32-bit Internet 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 “Environmental restrictions and programming requirements” on page 429.
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 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 78. GETHOSTID call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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 Internet 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 “Environmental restrictions and programming requirements” on page 429.
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 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 79. GETHOSTNAME call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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.

Parameter values returned to the application

NAME

Indicates the receiving field for the host name. TCP/IP Services allows a maximum length of 24 characters. The Internet standard is a maximum name length of 255 characters. The actual length of the NAME field is found in NAMELEN.

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See Appendix B, “Return codes,” on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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.

Note: Images from pre-V3R2 releases of TCP/IP Services are excluded. The GETIBMOPT call is not meaningful for pre-V3R2 releases. 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 it is not used, 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 *z/OS Communications Server: IP Configuration Reference*.

For detailed information about the standard method, refer to *z/OS Communications Server: New Function Summary*.

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 “Environmental restrictions and programming requirements” on page 429.
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 463 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 80. 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 version field is:

Version	Field
TCP/IP OS/390 CS V2R10	X'0510'
TCP/IP z/OS CS V1R2	X'0612'
TCP/IP z/OS CS V1R4	X'0614'
TCP/IP z/OS CS V1R5	X'0615'
TCP/IP z/OS CS V1R6	X'0616'

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 81. Example of name field

ERRNO

A fullword binary field. If RETCODE is negative, this contains an error number. See Appendix B, "Return codes," on page 781 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 “Environmental restrictions and programming requirements” on page 429.
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.

```

```

* 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 82. 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 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**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 Internet 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. 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.

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 specified you must also code **HOSTLEN**. Either **HOST/HOSTLEN** or **SERVICE/SERVLN** parameters, or both parameters, are required. An error occurs if both are omitted.

HOSTLEN

An output parameter. A fullword binary field that contains the length of the **HOST** storage used to contain the returned resolved host name. **HOSTLEN** must be equal to or greater than the length of the longest host name to be returned. **GETNAMEINFO** will return the host name up to the length specified by **HOSTLEN**. On

output, HOSTLEN will contain the length of the returned resolved host name. If HOSTLEN is 0 on input, then the resolved host name will not be returned. This is an optional field but if specified you must also code HOST. Either HOST/HOSTLEN or SERVICE/SERVLEN parameters, or both parameters, are required. An error occurs if both are omitted.

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 specified you must also code SERVLLEN. Either HOST/HOSTLEN or SERVICE/SERVLEN parameters, or both parameters, are required. An error occurs if both are omitted.

SERVLEN 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 specified you must also code SERVICE. Either HOST/HOSTLEN or SERVICE/SERVLEN parameters, or both parameters, are required. An error occurs if both are omitted.

FLAGS An input parameter. A fullword binary field. This is an optional field. The FLAGS field must contain either a binary 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.

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, "Return codes," on page 781 for information about ERRNO return codes.

RETCODE Output parameter. A fullword binary field that returns one of the following:

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 “Environmental restrictions and programming requirements” on page 429.
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 470 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 83. GETPEERNAME call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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 Internet 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 Internet 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, "Return codes," on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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 "Environmental restrictions and programming requirements" on page 429.
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 84 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 84. GETSOCKNAME call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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 Internet address of the local host machine.

RESERVED

Specifies 8 bytes of binary 0s. 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 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

A fullword binary field. If RETCODE is negative, the field contains an error number. See Appendix B, “Return codes,” on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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 “Environmental restrictions and programming requirements” on page 429.
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 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 85. GETSOCKOPT call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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 803 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, “Return codes,” on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

Value	Description
0	Successful call.
-1	Check ERRNO for an error code.

Table 19. 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 <i>hlq</i> .SEZAINST(CBLOCK) for the PL/I example of IP_MREQ. The IP_MREQ definition for COBOL: <pre>01 IP-MREQ. 05 IMR-MULTIADDR PIC 9(8) BINARY. 05 IMR-INTERFACE PIC 9(8) BINARY.</pre>	N/A
IP_DROP_MEMBERSHIP Use this option to enable an application to exit a multicast group. 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 <i>hlq</i> .SEZAINST(CBLOCK) for the PL/I example of IP_MREQ. The IP_MREQ definition for COBOL: <pre>01 IP-MREQ. 05 IMR-MULTIADDR PIC 9(8) BINARY. 05 IMR-INTERFACE PIC 9(8) BINARY.</pre>	N/A
IP_MULTICAST_IF Use this option to set or obtain the IPv4 interface address used for sending outbound multicast datagrams from the socket application. This is an IPv4-only socket option. Note: Multicast datagrams can be transmitted only on one interface at a time.	A 4-byte binary field containing an IPv4 interface address.	A 4-byte binary field containing an IPv4 interface address.

Table 19. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
IP_MULTICAST_LOOP Use this option to control or determine whether a copy of multicast datagrams are looped back for multicast datagrams sent to a group to which the sending host itself belongs. The default is to loop the datagrams back. This is an IPv4-only socket option.	A 1-byte binary field. To enable, set to 1. To disable, set to 0.	A 1-byte binary field. If enabled, will contain a 1. If disabled, will contain a 0.
IP_MULTICAST_TTL 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. This is an IPv4-only socket option.	A 1-byte binary field containing the value of '00'x to 'FF'x.	A 1-byte binary field containing the value of '00'x to 'FF'x.
IPV6_JOIN_GROUP Use this option to control the reception of multicast packets and specify that the socket join a multicast group. This is an IPv6-only socket option.	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. If the interface index number is 0, then the stack chooses the local interface. See the <i>hlq</i> .SEZAINST(CBLOCK) for the PL/I example of IPV6_MREQ. The IPV6_MREQ definition for COBOL: <pre> 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. </pre>	N/A

Table 19. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<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 <i>hlq</i>.SEZAINST(CBLOCK) for the PL/I example of IPV6_MREQ.</p> <p>The IPV6_MREQ definition for COBOL:</p> <pre> 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. </pre>	N/A
<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 from 0 to 255 indicating the number of multicast hops.</p>
<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>

Table 19. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
IPV6_UNICAST_HOPS Use this option to set or obtain the hop limit used for outgoing unicast IPv6 packets. This is an IPv6-only socket option.	Contains a 4-byte binary value specifying the unicast hops. If not specified, then the default is 1 hop. -1 indicates use stack default. 0 - 255 is the valid hop limit range. Note: APF authorized applications are permitted to set a hop limit that exceeds the system configured default. CICS applications cannot execute as APF authorized.	Contains a 4-byte binary value in the range from 0 to 255 indicating the number of unicast hops.
IPV6_V6ONLY 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. This is an IPv6-only socket option.	A 4-byte binary field. To enable, set to 1. To disable, set to 0.	A 4-byte binary field. If enabled, contains a 1. If disabled, contains a 0.
SO_ASCII 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. Note: This is a REXX-only socket option.	To enable, set to ON. To disable, set to OFF. 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.	If enabled, contains ON. If disabled, contains OFF. 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.
SO_BROADCAST 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. Note: This option has no meaning for stream sockets.	A 4-byte binary field. To enable, set to 1 or a positive value. To disable, set to 0.	A 4-byte field. If enabled, contains a 1. If disabled, contains a 0.
SO_DEBUG 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. Notes: <ol style="list-style-type: none"> 1. This is a REXX-only socket option. 2. This option has meaning only for stream sockets. 	To enable, set to ON. To disable, set to OFF.	If enabled, contains ON. If disabled, contains OFF.

Table 19. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
SO_EBCDIC 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. Note: This is a REXX-only socket option.	To enable, set to ON. To disable, set to OFF. 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.	If enabled, contains ON. If disabled, contains OFF. 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.
SO_ERROR 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.	N/A	A 4-byte binary field containing the most recent ERRNO for the socket.
SO_KEEPALIVE Use this option to set or determine whether the keepalive mechanism periodically sends a packet on an otherwise idle connection for a stream socket. The default is disabled. When activated, the keepalive 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.	A 4-byte binary field. To enable, set to 1 or a positive value. To disable, set to 0.	A 4-byte field. If enabled, contains a 1. If disabled, contains a 0.

Table 19. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<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>
<p>SO_OOBINLINE</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 field.</p> <p>If enabled, contains a 1.</p> <p>If disabled, contains a 0.</p>

Table 19. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
SO_RCVBUF Use this option to control or determine the size of the data portion of the TCP/IP receive buffer. The size of the data portion of the receive buffer is protocol-specific, based on the following values prior to any SETSOCKOPT call: <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 65 535 for a raw socket 	A 4-byte binary field. To enable, set to a positive value specifying the size of the data portion of the TCP/IP receive buffer. To disable, set to a 0.	A 4-byte binary field. If enabled, contains a positive value indicating the size of the data portion of the TCP/IP receive buffer. If disabled, contains a 0.
SO_REUSEADDR 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. When this option is enabled, the following situations are supported: <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, refer to the SHAREPORT option on the PORT statement in TCPIP.PROFILE. 	A 4-byte binary field. To enable, set to 1 or a positive value. To disable, set to 0.	A 4-byte field. If enabled, contains a 1. If disabled, contains a 0.

Table 19. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
SO_SNDBUF 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: <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 65 535 for a raw socket 	A 4-byte binary field. To enable, set to a positive value specifying the size of the data portion of the TCP/IP send buffer. To disable, set to a 0.	A 4-byte binary field. If enabled, contains a positive value indicating the size of the data portion of the TCP/IP send buffer. If disabled, contains a 0.
SO_TYPE Use this option to return the socket type.	N/A	A 4-byte binary field indicating the socket type: X'1' indicates SOCK_STREAM. X'2' indicates SOCK_DGRAM. X'3' indicates SOCK_RAW.
TCP_NODELAY Use this option to set or determine whether data sent over the socket is subject to the Nagle algorithm (RFC 896). 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. Note: Use the following to set TCP_NODELAY OPTNAME value for COBOL programs: 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.	A 4-byte binary field. To enable, set to a 0. To disable, set to a 1 or nonzero.	A 4-byte binary field. If enabled, contains a 0. If disabled, contains a 1.

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 "Environmental restrictions and programming requirements" on page 429.
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 484 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 86. GIVESOCKET call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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, 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, “Return codes,” on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

Value	Description
0	Successful 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/1, or assembler language must issue the INITAPI macro before they issue other sockets macros.

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 “Environmental restrictions and programming requirements” on page 429.
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 486 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 87. INITAPI call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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 will be 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. 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, containing 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 will ensure that, if you issue more than one INITAPI command from the same address space, each SUBTASK parameter will be unique.

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, “Return codes,” on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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 macro, you must load a value representing 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 20 on page 492 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 “Environmental restrictions and programming requirements” on page 429.
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 88 on page 488 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 'EZASOKET' USING SOC-FUNCTION S COMMAND REQARG
    RETARG ERRNO RETCODE.

```

Figure 88. IOCTL call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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.

- When the SIOCGHOMEIF6 IOCTL is issued, the REGARQ must contain a Network Configuration Header. The NETCONFHDR is defined in the SYS1.MACLIB(BPXYIOC6) 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 the following table.

```

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-LENTH    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 89. COBOL language example for SIOCGHOMEIF6

SIOCGIFADDR

Requests the IPv4 network interface address for a given interface name. See the NAME field in Figure 90 on page 491 for the address format.

SIOCGIFBRDADDR

Requests the IPv4 network interface broadcast address for a given interface name. See the NAME field in Figure 90 on page 491 for the address format.

SIOCGIFCONF

Requests the IPv4 network interface configuration. The configuration is a variable number of 32-byte structures formatted as shown in Figure 90.

- 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.

```

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).

```

Figure 90. Interface request structure (IFREQ) for the IOCTL call

SIOCGIFDSTADDR

Requests the network interface destination address for a given interface name. (See IFREQ NAME field, Figure 90 for format.)

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.

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 91. COBOL language example for SIOCGIFNAMEINDEX

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 and is used to pass arguments to IOCTL, and RETARG is an output parameter and receives arguments from IOCTL. The REQARG and RETARG parameters are described in Table 20.

Table 20. 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 89 on page 490 NetConfHdr.
SIOCGIFADDR X'C020A70D'	32	First 16 bytes - interface name. Last 16 bytes - not used.	32	Network interface address, see Figure 90 on page 491 for format.
SIOCGIFBRDADDR X'C020A712'	32	First 16 bytes - interface name. Last 16 bytes - not used.	32	Network interface address, see Figure 90 on page 491 for format.
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, see Figure 90 on page 491 for format.
SIOCGIFNAMEINDEX X'4000F603'	4	First 4 bytes size of return buffer.		See Figure 91 IF-NAMEINDEX .

Table 20. IOCTL call arguments (continued)

COMMAND/CODE	SIZE	REQARG	SIZE	RETARG
--------------	------	--------	------	--------

Notes:

- 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.

Parameter values returned to the application

RETARG

Returns an array whose size is based on the value in COMMAND. See Table 20 for information about REQARG and RETARG.

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See Appendix B, "Return codes," on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

Value	Description
0	Successful call.
-1	Check ERRNO for an error code.

The COMMAND SIOGIFCONF returns a variable number of network interface configurations. Figure 92 contains an example of a COBOL II routine that can be used to work with such a structure.

Note: This call can only be programmed in languages that support address pointers. Figure 92 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 92. 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. Once 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 “Environmental restrictions and programming requirements” on page 429.
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 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 93. LISTEN call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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.

Parameter values returned to the application

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an

error number. See Appendix B, “Return codes,” on page 781 for information about `ERRNO` return codes.

RETCODE

A fullword binary field that returns one of the following:

Value	Description
0	Successful call.
-1	Check <code>ERRNO</code> 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 “Environmental restrictions and programming requirements” on page 429.
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 Figure 94 on page 496 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 94. 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, “Return codes,” on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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 “Environmental restrictions and programming requirements” on page 429.
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 Figure 95 on page 498 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 95. 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, “Return codes,” on page 781 for information about ERRNO return codes.						
RETCODE	A fullword binary field that returns one of the following: <table><tr><th>Value</th><th>Description</th></tr><tr><td>0</td><td>Successful call.</td></tr><tr><td>-1</td><td>Check ERRNO for an error code.</td></tr></table>	Value	Description	0	Successful call.	-1	Check ERRNO for an error code.
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 551 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 “Environmental restrictions and programming requirements” on page 429.
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 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 'EZASOKET' USING SOC-FUNCTION S NBYTE BUF
                        ERRNO RETCODE.

```

Figure 96. READ call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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, “Return codes,” on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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 “Environmental restrictions and programming requirements” on page 429.
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 97 on page 502 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 97. READV call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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 filled in 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, “Return codes,” on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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 443 or "IOCTL" on page 487 for a description of how to set nonblocking mode.

For raw sockets, RECV adds a 20-byte header.

Note: See "EZACIC05" on page 551 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 "Environmental restrictions and programming requirements" on page 429.
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 98 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 98. RECV call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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	0	Read data.
OOB	1	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.
PEEK	2	Peek at the data, but do not destroy data. If the peek flag is set, the next RECV call will read the same data.

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, “Return codes,” on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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 443 or "IOCTL" on page 487 for a description of how to set nonblocking mode.

Note: See "EZACIC05" on page 551 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 "Environmental restrictions and programming requirements" on page 429.
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 99 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 99. RECVFROM call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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	0	Read data.
OOB	1	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.
PEEK	2	Peek at the data, but do not destroy data. If the peek flag is set, the next RECVMFROM call will read the same data.

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 Internet 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 Internet 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, “Return codes,” on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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 “Environmental restrictions and programming requirements” on page 429.
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 509 shows an example of RECVMSG call instructions.

```

WORKING-STORAGE SECTION.
01 SOC-FUNCTION      PIC X(16)  VALUE IS 'RCVMSG'.
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 RCVMSG-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 RCVMSG-BUFFER1    PIC X(16).
03 RCVMSG-BUFFER2    PIC X(16).
03 RCVMSG-BUFFER3    PIC X(16).
03 RCVMSG-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.

```

Figure 100. RCVMSG call instruction example (Part 1 of 2)

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 100. RECVMSG call instruction example (Part 2 of 2)

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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 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

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 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.

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.

ACCRLLEN

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	0	Read data.
OOB	1	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.
PEEK	2	Peek at the data, but do not destroy data. If the peek flag is set, the next RECVMSG call will read the same data.

Parameter values returned to the application

ERRNO

A fullword binary field. If RETCODE is negative, this contains an error number. See Appendix B, "Return codes," on page 781 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, do one of the following:

- Set the read, write, and except 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 "Environmental restrictions and programming requirements" on page 429.
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 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.

Each socket descriptor is represented by a bit in a bit string. The bit strings are contained in 32-bit fullwords, numbered from right to left. The rightmost bit represents socket descriptor 0, the leftmost bit represents socket descriptor 31, and so on. If your process uses 32 or fewer sockets, the bit string is 1 fullword. If your process uses 33 sockets, the bit string is 2 fullwords. You define the sockets that you want to test by turning on bits in the string.

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 552.

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. Once 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 in the range 0 through the MAXSOC value.

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 101 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 RSNDSK          PIC X(*).
01 WSNDSK          PIC X(*).
01 ESNDSK          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
                        RSNDSK WSNDSK ESNDSK
                        RRETMSK WRETMSK ERETMSK
                        ERRNO RETCODE.

```

* The bit mask lengths can be determined from the expression:
 $((\text{maximum socket number} + 32) / 32 \text{ (drop the remainder)}) * 4$

Figure 101. 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/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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 set to the largest socket descriptor number that is to be checked plus 1. (Remember to start counting at 0).

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, "Return codes," on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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, do either of the following:

- Set the read, write, and except 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 “Environmental restrictions and programming requirements” on page 429.
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 518 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:
 $((\text{maximum socket number} + 32) / 32 \text{ (drop the remainder)}) * 4$

Figure 102. SELECTEX call instruction example

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 specifying the largest socket descriptor number being checked.

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 SELECT call blocks until a socket becomes ready. 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.

SELECTB

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, "Return codes," on page 781 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 will be 0) or one of the caller's ECBs has been posted (ECB value will be nonzero and the caller's descriptor sets will be set to 0). The caller must initialize the ECB values to 0 before issuing the SELECTEX macro. |
| -1 | Check ERRNO for an error code. |

RRETMSK

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.

WRETMSK

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.

ERETMSK

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.

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 550 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 "Environmental restrictions and programming requirements" on page 429.
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 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 103. SEND call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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	0	No flag is set. The command behaves like a WRITE call.
OOB	1	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.
DONT-ROUTE	4	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, “Return codes,” on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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 “Environmental restrictions and programming requirements” on page 429.
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 523 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.

```

Figure 104. SENDMSG call instruction example (Part 1 of 2)

```

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 104. SENDMSG call instruction example (Part 2 of 2)

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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 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
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 Internet 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.

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:

Literal Value	Binary Value	Description
NO-FLAG	0	No flag is set. The command behaves like a WRITE call.
OOB	1	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.
DONTRROUTE	4	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, "Return codes," on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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 550 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 “Environmental restrictions and programming requirements” on page 429.
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 528 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 105. SENDTO call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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:

Literal Value	Binary Value	Description
NO-FLAG	0	No flag is set. The command behaves like a WRITE call.
OOB	1	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.
DONT-ROUTE	4	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 Internet 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 Internet 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, "Return codes," on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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 “Environmental restrictions and programming requirements” on page 429.
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 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 106. SETSOCKOPT call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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 803 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, “Return codes,” on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

Value	Description
0	Successful call.
-1	Check ERRNO for an error code.

Table 21. *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 <i>hlq</i>.SEZAINST(CBLOCK) for the PL/I example of IP_MREQ.</p> <p>The IP_MREQ definition for COBOL:</p> <pre>01 IP-MREQ. 05 IMR-MULTIADDR PIC 9(8) BINARY. 05 IMR-INTERFACE PIC 9(8) BINARY.</pre>	<p>N/A</p>
<p>IP_DROP_MEMBERSHIP</p> <p>Use this option to enable an application to exit 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 <i>hlq</i>.SEZAINST(CBLOCK) for the PL/I example of IP_MREQ.</p> <p>The IP_MREQ definition for COBOL:</p> <pre>01 IP-MREQ. 05 IMR-MULTIADDR PIC 9(8) BINARY. 05 IMR-INTERFACE PIC 9(8) BINARY.</pre>	<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 are 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 21. *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>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 <i>hlq</i>.SEZAINST(CBLOCK) for the PL/I example of IPV6_MREQ.</p> <p>The IPV6_MREQ definition for COBOL:</p> <pre> 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.</pre>	<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 <i>hlq</i>.SEZAINST(CBLOCK) for the PL/I example of IPV6_MREQ.</p> <p>The IPV6_MREQ definition for COBOL:</p> <pre> 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.</pre>	<p>N/A</p>

Table 21. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
IPV6_MULTICAST_HOPS Use to set or obtain the hop limit used for outgoing multicast packets. This is an IPv6-only socket option.	Contains a 4-byte binary value specifying the multicast hops. If not specified, then the default is 1 hop. -1 indicates use stack default. 0 - 255 is the valid hop limit range. 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.	Contains a 4-byte binary value in the range from 0 to 255 indicating the number of multicast hops.
IPV6_MULTICAST_IF Use this option to set or obtain the index of the IPv6 interface used for sending outbound multicast datagrams from the socket application. This is an IPv6-only socket option.	Contains a 4-byte binary field containing an IPv6 interface index number.	Contains a 4-byte binary field containing an IPv6 interface index number.
IPV6_MULTICAST_LOOP 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. This is an IPv6-only socket option.	A 4-byte binary field. To enable, set to 1. To disable, set to 0.	A 4-byte binary field. If enabled, contains a 1. If disabled, contains a 0.
IPV6_UNICAST_HOPS Use this option to set or obtain the hop limit used for outgoing unicast IPv6 packets. This is an IPv6-only socket option.	Contains a 4-byte binary value specifying the unicast hops. If not specified, then the default is 1 hop. -1 indicates use stack default. 0 - 255 is the valid hop limit range. Note: APF authorized applications are permitted to set a hop limit that exceeds the system configured default. CICS applications cannot execute as APF authorized.	Contains a 4-byte binary value in the range from 0 to 255 indicating the number of unicast hops.
IPV6_V6ONLY 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. This is an IPv6-only socket option.	A 4-byte binary field. To enable, set to 1. To disable, set to 0.	A 4-byte binary field. If enabled, contains a 1. If disabled, contains a 0.

Table 21. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
SO_ASCII 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. Note: This is a REXX-only socket option.	To enable, set to ON. To disable, set to OFF. 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.	If enabled, contains ON. If disabled, contains OFF. 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.
SO_BROADCAST 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. Note: This option has no meaning for stream sockets.	A 4-byte binary field. To enable, set to 1 or a positive value. To disable, set to 0.	A 4-byte field. If enabled, contains a 1. If disabled, contains a 0.
SO_DEBUG 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. Notes: <ol style="list-style-type: none"> 1. This is a REXX-only socket option. 2. This option has meaning only for stream sockets. 	To enable, set to ON. To disable, set to OFF.	If enabled, contains ON. If disabled, contains OFF.
SO_EBCDIC 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. Note: This is a REXX-only socket option.	To enable, set to ON. To disable, set to OFF. 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.	If enabled, contains ON. If disabled, contains OFF. 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.
SO_ERROR 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.	N/A	A 4-byte binary field containing the most recent ERRNO for the socket.

Table 21. *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 keepalive 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 keepalive 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 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 21. 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 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 65 535 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 21. 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, refer to 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 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:</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 65 535 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>
<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>

Table 21. *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 37 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 “Environmental restrictions and programming requirements” on page 429.
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 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 107. SHUTDOWN call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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, “Return codes,” on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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 “Environmental restrictions and programming requirements” on page 429.
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 108 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 108. SOCKET call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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:

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, "Return codes," on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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 482 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 “Environmental restrictions and programming requirements” on page 429.
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 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 109. TAKESOCKET call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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 RETCODE is negative, the field contains an error number. See Appendix B, "Return codes," on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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 “Environmental restrictions and programming requirements” on page 429.
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 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 110. TERMAPI call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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 wishes 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 550 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 “Environmental restrictions and programming requirements” on page 429.
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 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 111. WRITE call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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, “Return codes,” on page 781 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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 “Environmental restrictions and programming requirements” on page 429.
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 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 112. WRITEV call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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, "Return codes," on page 781 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 the following utility programs to translate data:

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 118 on page 562.
- EZACIC15 provides an alternative to EZACIC05 and translates ASCII data to EBCDIC data using the translation table documented in Figure 120 on page 564.

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:

- 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 113 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 113. EZACIC04 call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

OUT-BUFFER

A buffer that contains the following:

- 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 114 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 114. EZACIC05 call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

IN-BUFFER

A buffer that contains the following:

- When called, ASCII data
- Upon return, EBCDIC data

LENGTH

Specifies the length of the data to be translated.

EZACIC06

The SELECT 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 the assembler language program EZACIC06 to translate them to character strings to be used with the SELECT call.

Figure 115 shows an example of EZACIC06 call instructions.

```
WORKING-STORAGE SECTION.  
  01 CHAR-MASK.  
    05 CHAR-STRING          PIC X(nn).  
  
  01 CHAR-ARRAY              REDEFINES CHAR-MASK.  
    05 CHAR-ENTRY-TABLE     OCCURS nn TIMES.  
      10 CHAR-ENTRY         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 115. EZACIC06 call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

TOKEN

Specifies a 16-character identifier. This identifier is required and it must be the first parameter in the list.

CHAR-MASK

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.]

BIT-MASK

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

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 Internet addresses.

TCP/IP uses indirect addressing to connect the variable number of alias names and Internet 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 Internet 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:
 - 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 Internet address type, always 2 for AF_INET
 - The host Internet address length, always 4 for AF_INET
 - The number of host Internet addresses for this host
 - The host Internet address sequence number
 - The host Internet address
3. If the GETHOSTBYADDR or GETHOSTBYNAME call returns more than one alias name or host Internet address (554 or 554 above), the application program should repeat the call to EZACIC08 until all alias names and host Internet addresses have been retrieved.

Figure 116 on page 555 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 116. EZACIC08 call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

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 Internet 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 Internet addresses returned by this instance of the call.

HOSTADDR-SEQ

This halfword binary field contains the sequence number of the host Internet address currently found in HOSTADDR-VALUE.

HOSTADDR-VALUE

This fullword binary field contains a host Internet address.

RETURN-CODE

This fullword binary field contains the EZACIC08 return code:

Value	Description
0	Successful completion.
-1	HOSTENT address is not valid.
-2	Invalid value in HOSTALIAS-SEQ.
-3	Invalid value in HOSTADDR-SEQ.

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:
 - 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 117 on page 558 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-numericserve 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.

```

Figure 117. EZACIC09 call instruction example (Part 1 of 3)


```

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.

```

Figure 117. EZACIC09 call instruction example (Part 2 of 3)

```

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 117. EZACIC09 call instruction example (Part 3 of 3)

For equivalent PL/I and assembler language declarations, see "Converting parameter descriptions".

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 118 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	0C	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	B4	DB	DC	D9	DA	9F

Figure 118. EZACIC14 EBCDIC-to-ASCII table

Figure 119 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 119. EZACIC14 call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

OUT-BUFFER

A buffer that contains the following:

- 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 120 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 120. EZACIC15 ASCII-to-EBCDIC table

Figure 121 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 121. EZACIC15 call instruction example

For equivalent PL/1 and assembler language declarations, see “Converting parameter descriptions” on page 432.

OUT-BUFFER

A buffer that contains the following:

- When called, ASCII data

- Upon return, EBCDIC data

LENGTH

Specifies the length of the data to be translated.

Call interface sample programs

This section 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 available in the *hlq.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
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, 2003              */
/*
/*              US Government Users Restricted Rights -         */
/*              Use, duplication or disclosure restricted by     */
/*              GSA ADP Schedule Contract with IBM Corp.       */
/*
/* Status:       CSV1R5                                          */
/*
*****/
EZASOKPS: PROC OPTIONS(MAIN);

/* INCLUDE CBLOCK - common variables                            */
% include CBLOCK;

ID.TCPNAME = 'TCP/IP';          /* 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                                              */

```

Figure 122. EZASOKPS PL/1 sample server program for IPv4 (Part 1 of 4)

```

/*                                                                    */
/*****                                                                    */

call ezasoket(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;
else sock_stream = retcode;

/*****                                                                    */
/*                                                                    */
/* Execute BIND                                                            */
/*                                                                    */
/*****                                                                    */

name_id.port = 8888;
name_id.address = '01234567'BX;                /* internet address */
call ezasoket(BIND, SOCK_STREAM, NAME_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                                                    */
/*                                                                    */
/*****                                                                    */

name_id.port = 8888;
name_id.address = '01234567'BX;                /* internet address */
call ezasoket(GETSOCKNAME, SOCK_STREAM,
              NAME_ID, ERRNO, RETCODE);
msg = blank;                                /* clear field */
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                                                         */
/*                                                                    */
/*****                                                                    */

```

Figure 122. EZASOKPS PL/1 sample server program for IPv4 (Part 2 of 4)

```

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
/*
/*
*****/

name_id.port = 8888;
name_id.address = '01234567'BX;                /* internet address */
call ezasocket(ACCEPT, SOCK_STREAM,
               NAME_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 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 = length(bufout);
end;

/*****
/*
*****/

```

Figure 122. EZASOKPS PL/1 sample server program for IPv4 (Part 3 of 4)

```

/* 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 ezasokps;

```

Figure 122. EZASOKPS PL/1 sample server program for IPv4 (Part 4 of 4)

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 = 'TCP/IP';          /* 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);

```

Figure 123. EZASOKPC PL/1 sample client program for IPv4 (Part 1 of 3)

```

    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;

```

Figure 123. EZASOKPC PL/1 sample client program for IPv4 (Part 2 of 3)

```

        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 ezasokpc;

```

Figure 123. EZASOKPC PL/I sample client program for IPv4 (Part 3 of 3)

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, 2003            */
/*
/*                US Government Users Restricted Rights -       */
/*                Use, duplication or disclosure restricted by   */
/*                GSA ADP Schedule Contract with IBM Corp.     */
/*
/*  Status:       CSV1R5                                         */
/*
/*****
EZASO6PS: PROC OPTIONS(MAIN);

/* INCLUDE CBLOCK - common variables */
% include CBLOCK;

ID.TCPNAME = 'TCPCS';          /* Set TCP to use */
ID.ADSNAME = 'EZASO6PS';      /* 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 */
/*

```

Figure 124. EZASO6PS PL/1 sample server program for IPv6 (Part 1 of 6)


```

/*****/

call ezasoket(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 ezasoket(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 ezasoket(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 */

```

Figure 124. EZASO6PS PL/1 sample server program for IPv6 (Part 2 of 6)

```

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;

```

Figure 124. EZASO6PS PL/1 sample server program for IPv6 (Part 3 of 6)

```

    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;

```

Figure 124. EZASO6PS PL/1 sample server program for IPv6 (Part 4 of 6)

```

/*****
/*
/* 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 = length(bufout);
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;

```

Figure 124. EZASO6PS PL/1 sample server program for IPv6 (Part 5 of 6)

```

    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 124. EZASO6PS PL/I sample server program for IPv6 (Part 6 of 6)

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:  EZASO6PC - 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                                         */
/*
*****/
EZASO6PC: PROC OPTIONS(MAIN);

/* INCLUDE CBLOCK - common variables                          */
% include CBLOCK;

ID.TCPNAME = 'TCPSC';          /* Set TCP to use          */
ID.ADSNAME = 'EZASO6PS';      /* 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);

```

Figure 125. EZASO6PC PL/1 sample client program for IPv6 (Part 1 of 4)

```

    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
/*
*****/

```

Figure 125. EZASO6PC PL/1 sample client program for IPv6 (Part 2 of 4)

```

/*****/

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                */

```

Figure 125. EZASO6PC PL/1 sample client program for IPv6 (Part 3 of 4)


```

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 125. EZASO6PC PL/1 sample client program for IPv6 (Part 4 of 4)

Common variables used in PL/I sample programs

The CBLOCK common storage area contains the variables that are used in the PL/1 programs in this section.

```

/*****
/*
/*  MODULE NAME:  CBLOCK - SOKET COMMON VARIABLES
/*
/*
/* Copyright:    Licensed Materials - Property of IBM
/*
/*
/*              "Restricted Materials of IBM"
/*
/*
/*              5694-A01
/*
/*
/*              (C) Copyright IBM Corp. 1994, 2003
/*
/*
/*              US Government Users Restricted Rights -
/*              Use, duplication or disclosure restricted by
/*              GSA ADP Schedule Contract with IBM Corp.
/*
/*
/* Status:       CSV1R5
/*
/*****
/*****
/*
/* SOKET COMMON VARIABLES
/*
/*****

DCL ABS      BUILTIN;
DCL ADDR     BUILTIN;
DCL ACCEPT   CHAR(16) INIT('ACCEPT');
DCL ACCPSOCK 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               */
DCL ai_PASSIVE FIXED BIN(31) INIT(1);
                                   /* flag: getaddrinfo hints  */
DCL ai_CANONNAMEOK FIXED BIN(31) INIT(2);
                                   /* flag: getaddrinfo hints  */
DCL ai_NUMERICHOST FIXED BIN(31) INIT(4);
                                   /* flag: getaddrinfo hints  */
DCL ai_NUMERICSERV FIXED BIN(31) INIT(8);
                                   /* flag: getaddrinfo hints  */
DCL ai_V4MAPPED FIXED BIN(31) INIT(10);
                                   /* flag: getaddrinfo hints  */
DCL ai_ALL FIXED BIN(31) INIT(20);
                                   /* flag: getaddrinfo hints  */
DCL ai_ADDRCONFIG FIXED BIN(31) INIT(40);
                                   /* 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 BIT       BUILTIN;
DCL BITZERO   BIT(1);            /* bit zero value            */
DCL BLANK255  CHAR(255) INIT(' '); /*

```

Figure 126. CBLOCK PL/1 common variables (Part 1 of 7)

```

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_SOCK 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 ESNDMSK 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 FIXED BIN(31) INIT(-2147178626); /* flag: nonblocking */
DCL FIONREAD FIXED BIN(31) INIT(+1074046847); /* 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 */
    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 CANONNAME FIXED BIN(31) INIT(0),
    2 NAME FIXED BIN(31) INIT(0),
    2 NEXT FIXED BIN(31) INIT(0);
DCL 1 GAI_ADDRINFO BASED(RES), /* getaddrinfo RES addrinfo */
    2 FLAGS FIXED BIN(31),
    2 AF FIXED BIN(31),
    2 SOCTYPE FIXED BIN(31),

```

Figure 126. CBLOCK PL/1 common variables (Part 2 of 7)

```

2 PROTO      FIXED BIN(31),
2 NAMELEN    FIXED BIN(31), /* RES socket address struct length*/
2 CANONNAME  POINTER,      /* RES canonical name          */
2 NAME       POINTER,      /* RES socket address structure */
2 NEXT       POINTER;      /* RES next addrinfo, zero if none.*/
DCL 1 GAI_NAME_ID BASED(GAI_ADDRINFO.NAME),
2 LEN       BIT(8),
2 FAMILY    BIT(8),
2 PORT      FIXED BIN(15),
2 ADDRESS    FIXED BIN(31),
2 RESERVED1 CHAR(8);
DCL 1 GAI_NAME6_ID BASED(GAI_ADDRINFO.NAME),
2 LEN       BIT(8),
2 FAMILY    BIT(8),
2 PORT      FIXED BIN(15),
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 HINTS POINTER; /*getaddrinfo hints addrinfo pointer*/
DCL 1 HOMEIF, /* Home Interface Structure */
2 ADDRESS CHAR(16); /* Home Interface Address */
DCL HOSTADDR FIXED BIN(31); /* 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 I FIXED BIN(15); /* loop index */
DCL ICMP FIXED BIN(31) INIT(2); /* prototype icmp ??? */
DCL 1 ID, /* */
2 TCPNAME CHAR(8) INIT('TCP/IP'), /* remote address space */
2 ADSNAME 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 */

```

Figure 126. CBLOCK PL/1 common variables (Part 3 of 7)

```

2 IF_NITABLE(10) CHAR(24);
DCL 1 IF_NAMEINDEXENTRY,
2 IF_NIINDEX FIXED BIN(31),          /* Interface Index          */
2 IF_NINAME CHAR(16),                /* Interface Name, blank padded */
2 IF_NIEXT,
3 IF_NINAMETERM CHAR(1),             /* Null for C for Name len=16 */
3 IF_RESERVED CHAR(3);              /* Reserved                  */
DCL IFREQ CHAR(255);                  /* interface structure       */
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_REQ00 FIXED BIN(31);        /* command request argument */
DCL IOCTL_REQ04 FIXED BIN(31);        /* command request argument */
DCL IOCTL_REQ08 FIXED BIN(31);        /* command request argument */
DCL IOCTL_REQ32 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 INITAPI CHAR(16) INIT('INITAPI'); /*                            */
DCL 1 INTERNET,                      /* internet address         */
2 NETID1 FIXED BIN(31) INIT(9), /* network id, part 1       */
2 NETID2 FIXED BIN(31) INIT(67), /* network id, part 2       */
2 SUBNETID FIXED BIN(31) INIT(30), /* subnet id                */
2 HOSTID FIXED BIN(31) INIT(137); /* host id                  */
DCL IP FIXED BIN(31) INIT(1);         /* prototype ip            */
DCL 1 IP_MREQ,
2 IMR_MULTIADDR,                    /* IP multicast addr of group */
3 NETID1 FIXED BIN(31),             /* network id, part 1       */
3 NETID2 FIXED BIN(31),             /* network id, part 2       */
3 SUBNETID FIXED BIN(31),           /* subnet id                */
3 HOSTID FIXED BIN(31),             /* host id                  */
2 IMR_INTERFACE,                    /* local IP addr of interface */
3 NETID1 FIXED BIN(31),             /* network id, part 1       */
3 NETID2 FIXED BIN(31),             /* network id, part 2       */
3 SUBNETID FIXED BIN(31),           /* subnet id                */
3 HOSTID FIXED BIN(31);             /* host id                  */
DCL 1 IPV6_MREQ,
2 IPV6MR_MULTIADDR CHAR(16),
2 IPV6MR_INTERFACE FIXED BIN(31);
DCL IP_MULTICAST_TTL FIXED BIN(31) INIT(1048579);
DCL IP_MULTICAST_LOOP FIXED BIN(31) INIT(1048580);
DCL IP_MULTICAST_IF FIXED BIN(31) INIT(1048583);
DCL IP_ADD_MEMBERSHIP FIXED BIN(31) INIT(1048581);
DCL IP_DROP_MEMBERSHIP FIXED BIN(31) INIT(1048582);
DCL IPRES POINTER;                  /* EZACIC09 RES addrinfo ptr */
DCL IPV6_JOIN_GROUP FIXED BIN(31) INIT(65541);
DCL IPV6_LEAVE_GROUP FIXED BIN(31) INIT(65542);
DCL IPV6_LOOPBACK CHAR(3) INIT('::1');
DCL IPV6_MULTICAST_HOPS FIXED BIN(31) INIT(65545);
DCL IPV6_MULTICAST_IF FIXED BIN(31) INIT(65543);
DCL IPV6_MULTICAST_LOOP FIXED BIN(31) INIT(65540);
DCL IPV6_UNICAST_HOPS FIXED BIN(31) INIT(65539);
DCL IPV6_V6ONLY FIXED BIN(31) INIT(65546);
DCL J FIXED BIN(15);                /* loop index               */

```

Figure 126. CBLOCK PL/1 common variables (Part 4 of 7)

```

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 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    FIXED BIN(15), /* system assigned port # */
    2 ADDRESS FIXED BIN(31), /* 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    FIXED BIN(15), /* 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(10);
    /* 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;

```

Figure 126. CBLOCK PL/1 common variables (Part 5 of 7)

```

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 RECVR FROM CHAR(16) INIT('RECVR FROM');
DCL RECVMMSG CHAR(16) INIT('RECVMMSG');
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 RSNDMSK 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 SELECTCB 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 SETSOCKOPT CHAR(16) INIT('SETSOCKOPT');
DCL SHUTDOWN CHAR(16) INIT('SHUTDOWN');
DCL SIOCADDRT FIXED BIN(31) INIT(-2144295158);
DCL SIOCATMARK FIXED BIN(31) INIT(+1074046727); /* flag: add routing entry */
DCL SIOCDELRT FIXED BIN(31) INIT(-2144295157); /* flag: out-of-band data */
DCL SIOCGIFADDR FIXED BIN(31) INIT(-1071601907); /* flag: delete routing */
DCL SIOCGIFADDR FIXED BIN(31) INIT(-1071601907); /* flag: network int addr */

```

Figure 126. CBLOCK PL/1 common variables (Part 6 of 7)

```

DCL SIOCGHOMEIF6 BIT(32) INIT('C014F608'BX);
/* flag: netw int config */
DCL SIOCGIFBRDADDR FIXED BIN(31) INIT(-1071601902);
/*flag net broadcast*/
DCL SIOCGIFCONF FIXED BIN(31) INIT(-1073174764);
/* flag: netw int config*/
DCL SIOCGIFDSTADDR FIXED BIN(31) INIT(-1071601905);
/* flag: net des addr*/
DCL SIOCGIFFLAGS FIXED BIN(31) INIT(-1071601903);
/* flag: net intf flags*/
DCL SIOCGIFMETRIC FIXED BIN(31) INIT(-1071601897);
/* flag: get rout metr*/
DCL SIOCGIFNAMEINDEX BIT(32) INIT('4000F603'BX);
/* flag: name and indexes */
DCL SIOCGIFNETMASK FIXED BIN(31) INIT(-1071601899);
/* flag: network mask*/
DCL SIOCGIFNONSENSE FIXED BIN(31) INIT(-1234567890);
/* flag: nonsense */
DCL SIOCSIFMETRIC FIXED BIN(31) INIT(-2145343720);
/* flag: set rout metr*/
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_KEEPAIVE FIXED BIN(31) INIT(8); /* request status of stream*/
DCL SO_LINGER FIXED BIN(31) INIT(128); /* toggle, linger on close */
DCL SO_OOBINLINE FIXED BIN(31) INIT(256); /*toggle, out-of-bound data*/
DCL SO_REUSEADDR FIXED
BIN(31) INIT(4); /* toggle, local address reuse*/
DCL SO_SNDBUF FIXED BIN(31) INIT(4097);
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 TYPE_DATAGRAM FIXED BIN(31) INIT(2); /*fixed lengthconnectionless*/
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 WSDMSK CHAR(4); /*check for pending write events */
DCL TCP_NODELAY FIXED BIN(31) INIT(-2147483647);

```

Figure 126. CBLOCK PL/1 common variables (Part 7 of 7)

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
*
*                  (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:     COBOL II
*
*****
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 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  '.

```

Figure 127. EZAS06CS COBOL call interface sample IPv6 server program (Part 1 of 13)

```

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-termapapi       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 'EZASO6CS'.
01 maxsno                 pic 9(8) Binary Value 1.
*-----*
* Variables returned by the GETCLIENTID Call              *
*-----*

```

Figure 127. EZASO6CS COBOL call interface sample IPv6 server program (Part 2 of 13)

```

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-numericerv               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.
*-----*

```

Figure 127. EZASO6CS COBOL call interface sample IPv6 server program (Part 3 of 13)

```

* 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.
*-----*

```

Figure 127. EZASO6CS COBOL call interface sample IPv6 server program (Part 4 of 13)

```

* 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

```

Figure 127. EZASO6CS COBOL call interface sample IPv6 server program (Part 5 of 13)

[illegible]

```

*-----*
Get-ClientID.
  move soket-getclientid to ezaerror-function.
  Call 'EZASOKET' 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 'EZASOKET' 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 '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.

*-----*
* 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.

```

Figure 127. EZASO6CS COBOL call interface sample IPv6 server program (Part 7 of 13)


```

    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-unstrung
        count in host-name-char-count.
    string host-name-unstrung 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

```

Figure 127. EZASO6CS COBOL call interface sample IPv6 server program (Part 8 of 13)

```

        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 address'

```

Figure 127. EZASO6CS COBOL call interface sample IPv6 server program (Part 9 of 13)

```

        move output-ipv4-ipaddr to numeric-addr
        move 16 to presentable-addr-len
    else
        display 'Formatting ipv6 address'
        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 soket-listen to ezaerror-function.
    Call 'EZASOKET' using soket-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.

```

Figure 127. EZASO6CS COBOL call interface sample IPv6 server program (Part 10 of 13)

```

*-----*
*  Accept a connection request                                     *
*-----*
Accept-Connection.
  Move socket-accept to ezaerror-function.
  Call 'EZASOKET' 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 'EZASOKET' 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 'EZASOKET' 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 'EZASOKET' using socket-write socket-descriptor-new

```

Figure 127. EZAS06CS COBOL call interface sample IPv6 server program (Part 11 of 13)

```

        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 soket-close to ezaerror-function
    Call 'EZASOKET' using soket-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.

*-----*

```

Figure 127. EZAS06CS COBOL call interface sample IPv6 server program (Part 12 of 13)

```

* 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 ' EZASO6CS: ' 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 127. EZASO6CS COBOL call interface sample IPv6 server program (Part 13 of 13)

COBOL call interface sample IPv6 client program

The EZASO6CC 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
*
*                 (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:     COBOL II
*
*****

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 '.

```

Figure 128. EZAS06CC COBOL call interface sample IPv6 client program (Part 1 of 9)

```

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 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).

```

Figure 128. EZASO6CC COBOL call interface sample IPv6 client program (Part 2 of 9)


```

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 'EZSO6CC'.
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.

```

Figure 128. EZASO6CC COBOL call interface sample IPv6 client program (Part 3 of 9)

```

    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='.

```

Figure 128. EZASO6CC COBOL call interface sample IPv6 client program (Part 4 of 9)

```

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-getclntid to ezaerror-function.
    Call 'EZASOKET' using soket-getclntid 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 socket-socket to ezaerror-function.
    Call 'EZASOCKET' using socket-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 socket-pton to ezaerror-function.
    move IN6ADDR-LOOPBACK to presentable-addr.
    Call 'EZASOCKET' using socket-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 socket-connect to ezaerror-function.
    CALL 'EZASOCKET' 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.

```

Figure 128. EZASO6CC COBOL call interface sample IPv6 client program (Part 6 of 9)

```

*-----*
* Use NTOP to display the IP address.                                     *
*-----*
Numeric-To-Presentation.
  move socket-ntop to ezaerror-function.
  move server-ipaddr to numeric-addr.
  move socket-ntop to ezaerror-function.
  Call 'EZASOKET' using socket-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 socket-getnameinfo to ezaerror-function.
  Call 'EZASOKET' using socket-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 socket-write to ezaerror-function.
  Move 'Message from EZAS06CC' to buf.
  Call 'EZASOKET' using socket-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.

```

Figure 128. EZAS06CC COBOL call interface sample IPv6 client program (Part 7 of 9)

```

Exit.

*-----*
* Shutdown to pipe *
*-----*
Shutdown-Send.
  Move socket-shutdown to ezaerror-function.
  move 1 to how.
  Call 'EZASOCKET' using socket-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 socket-read to ezaerror-function.
  Move spaces to buf.
  Call 'EZASOCKET' using socket-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 socket-shutdown to ezaerror-function.
  move 0 to how.
  Call 'EZASOCKET' using socket-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 socket-close to ezaerror-function.
  Call 'EZASOCKET' using socket-close socket-descriptor
    errno retcode.

```

Figure 128. EZASO6CC COBOL call interface sample IPv6 client program (Part 8 of 9)

```

        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 128. EZAS06CC COBOL call interface sample IPv6 client program (Part 9 of 9)

Chapter 14. REXX socket application programming interface (API)

This chapter describes the application program interface (API) for IPv4 or IPv6 socket call instructions written in REXX for TCP/IP z/OS CS environment.

The REXX socket program uses the REXX built-in function RXSOCKET to access the TCP/IP socket interface. The program maps the socket calls from the C programming language to the REXX programming language. This allows you to use REXX to implement and test TCP/IP applications. Examples of the corresponding C socket call are included where they apply.

This chapter describes the following:

- REXX socket initialization
- REXX socket programming hints and tips
- Coding socket built-in function
- Error messages and return codes
- Coding calls to process socket sets
- Coding calls to initialize, change, and close sockets
- Coding calls to exchange data for sockets
- Coding calls to resolve names for REXX sockets
- Coding calls to manage configuration, options, and modes
- REXX socket sample programs

REXX socket initialization

The member RXSOCKET in the TCP/IP load library is called when the REXX built-in function socket is called. RXSOCKET must be part of the step library when your REXX program calls the socket built-in function.

REXX socket programming hints and tips

This section contains information that you might find useful if you plan to use the REXX socket interface to TCP/IP.

- To use the socket calls contained in the socket function, one socket set must be active. The Initialize call creates a socket set and can support multiple calls. The *subtaskid* for a socket set identifies the socket set and normally corresponds to the application name.
- The *name* parameter contains *domain*, *portid*, and *ipaddress* for an IPv4 socket address and *domain*, *portid*, *flowinfo*, *ipaddress*, and *scopeid* for an IPv6 socket address. If specified as an input parameter for socket calls, you can specify *ipaddress* as a name that is resolved by the name server. For example, you can enter 'IBMVM1' or 'IBMVM1.IBM.COM'. When returned as a result, *ipaddress* is specified as either the IPv4 dotted decimal format or the IPv6 colon-hex format. For example, '128.228.1.2' or '12f9:0:0:c30:123:457:9cb:bef' can be returned.
- A socket can be in blocking or nonblocking mode. In blocking mode, calls such as send and recv block the caller until the operation completes successfully or an error occurs. In nonblocking mode, the caller is not blocked, but the operation ends immediately with the return code 35 (EWOULDBLOCK) or 36 (EINPROGRESS). You can use the fcntl or Ioctl calls to switch between blocking and nonblocking mode.

- When a socket is in nonblocking mode, you can use the select call to wait for socket events. Data arriving at a socket for a read or recv call is a possible event. If the socket is not ready to send data because buffer space for the transmitted message is not available at the receiving socket, your REXX program can wait until the socket is ready for sending data.
- If your application uses the givesock and takesock calls to transfer a socket from a master program to a slave program, both the master and slave programs need to agree on a mechanism for exchanging client IDs and the socket ID to be transferred. The slave program must also signal the master program when the takesocket call is successfully completed. The master program can then close the socket.
- The socket options SO_ASCII and SO_EBCDIC identify the socket's data type for use by the REXX/RXSOCKET program. Setting SO_EBCDIC on has no effect, and setting SO_ASCII 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. REXX/RXSOCKET uses the following hierarchy of translation tables:

```

user_prefix.subtaskid.TCPXLBIN
user_prefix.userid.TCPXLBIN
system_prefix.STANDARD.TCPXLBIN
system_prefix.RXSOCKET.TCPXLBIN
Internal tables

```

Note: The user_prefix is either the user ID or jobname of the REXX program.
The system_prefix is either TCPIP or the DATASETPREFIX value from the *hlq.TCPIP.DATA* data set.

The first four tables are data sets and they are searched in the order in which they are listed. If no files are found, the internal tables corresponding to the ISO standard are used. You can change the system_prefix parameter to match your site convention.

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.

Coding the socket built-in function

The socket built-in function describes an interface that allows you to use the socket calls associated with the C language in a REXX application. Calls are included to:

- Process socket sets
- Initialize, change, and close sockets
- Exchange data for sockets
- Resolve names for sockets
- Manage configurations, options, and modes for sockets

The first parameter in the Socket built-in function identifies the function of the call. The corresponding C socket call is included where applicable.

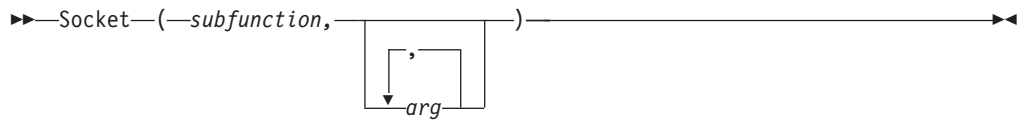
Two REXX sample programs for a client/server application are described in “REXX socket sample programs” on page 690.

The socket calls are ordered by function. For example, the initialize and terminate calls are described before the calls that process sockets.

Socket

This call provides access to the TCP/IP socket interface. The subfunction specifies the socket call. Each call contains the name socket, *subfunction*, and optional parameters (*arg*). For a description of each call, see:

- “Coding calls to process socket sets” on page 619
- “Coding calls to initialize, change, and close sockets” on page 625
- “Coding calls to exchange data for sockets” on page 637
- “Coding calls to resolve names for REXX sockets” on page 646
- “Coding calls to manage configuration, options, and modes” on page 666



Parameters

subfunction

The name of the socket call

arg

The parameters for the socket call

Return Values

A character string containing a return code is returned.

The returned string contains several values separated by a blank to allow you to parse it using REXX. The first value is a return code. If the return code is 0, the values following the return code are returned by the subfunction called. If the return code is not 0, the second value indicates an error code and the rest of the string is the corresponding error message.

```
Socket('GetHostId')      ==  '0 9.4.3.2'
Socket('Recv',socket)    ==  '35 EWOULDBLOCK Operation would block'
```

Error messages and return codes

For information about error messages, refer to *z/OS Communications Server: IP Messages Volume 1 (EZA)*.

The REXX built-in function `socket` returns a return code in the first token slot in the result string. For more information about return codes, see Appendix B, “Return codes,” on page 781.

Coding calls to process socket sets

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.

Initialize

This call preallocates the number of sockets specified and returns the *subtaskid* for the socket set that is active when this Initialize call is issued. It also returns the maximum number of preallocated sockets and the name of the TCP/IP service provider. If this call is successful, the socket set identified by *subtaskid* automatically becomes the active socket set.

►►—Socket—(—'Initialize'—,—*subtaskid*—,—maxdesc—,—service—)—►►

Parameters

subtaskid

A name for a socket set. The name can be as many as eight printable characters and cannot contain blanks.

maxdesc

The number of sockets that can be opened in a socket set. The number can be in the range 1-65535. The default is 40.

service

The name of the TCP/IP service. The service name should match the TYPE operand that was specified on the FILESYSTYPE statement, or the NAME operand of the SUBFILESYSTYPE statement that defined this physical file system in the BPXPRMxx PARMLIB member. In an INET environment, any service name will be accepted. However, if the service name is not correct, the INITIALIZE will complete successfully, but the service name returned will be changed to '*INET' to indicate to the caller that an INET environment exists.

Return Values

A string containing a return code, *subtaskid*, *maxdesc*, and *service* is returned.

Example

```
Socket('Initialize','myId') == '0 myId 40 TCPIP'
```

Socketsetlist

This call returns a list of the subtask IDs for all available socket sets in the current order of the stack.

►►—Socket—(—'Socketsetlist'—)—————►◄

Return Values

A string containing a return code and the active *subtaskid* is returned. If this call changes the active *subtaskid*, the previous *subtaskid* is also returned.

Example

```
Socket('SocketSetList')      ==  '0 myId firstId'
```

Socketset

This call returns the *subtaskid* of the active socket set, and optionally makes the specified socket set the active socket set.

►►—Socket—(—'Socketset'—, —*subtaskid*—)——►►

Parameters

subtaskid

The name for a socket set. The name can be as many as eight printable nonblank characters.

Return Values

A string containing a return code and a subtask ID is returned.

Example

```
Socket('SocketSet','firstId') == '0 myId'
```


Socketsetstatus

This call returns the status of the socket set. If the socket set is connected, the call returns the number of free and the number of allocated sockets in the socket set. If the set is severed, the reason for the TCP/IP sever is also returned. If the *subtaskid* parameter is not specified, the active socket set is used. Initialized socket sets should be in connected status, and uninitialized socket sets should be in free status.

A socket set that is initialized and is not in connected status must be terminated before the *subtaskid* can be reused.

►►Socket—('Socketsetstatus', subtaskid)—►►

Parameters

subtaskid

The name for a socket set. The name can be as many as eight printable nonblank characters.

Return Values

The string containing a return code, *subtaskid* and status is returned. The call optionally returns connect and sever information.

Example

```
Socket('SocketSetStatus') == '0 myId Connected Free 17 Used 23'
```

Terminate

This call closes all sockets in the socket set, releases the socket set, and returns the *subtaskid* for the socket set. If the *subtaskid* is not specified, the active socket set is ended. If the active socket set is ended, the next socket set in the stack becomes the active socket set.

» Socket (— 'Terminate' —, subtaskid) —

Parameters

subtaskid

A name for a socket set. The name can be as many as eight printable characters and cannot contain blanks.

Return Values

A string containing a return code and a *subtaskid* is returned.

Example

```
Socket('Terminate','myId') == '0 myId'
```

Coding calls to initialize, change, and close sockets

A socket is an endpoint for communication that can be named and addressed in a network. A socket is represented by a socket identifier (*socketid*). A socket ID used in a Socket call must be in the active socket set.

Accept

This call is used by a server to accept a connection request from a client. It accepts the first connection on the queue of pending connections. It creates a new *socketid* with the same properties as the given *socketid*. If the queue has no pending connection requests, accept blocks the caller unless the socket is in nonblocking mode. If no connection request is pending and the socket is in nonblocking mode, accept ends with the return code 35 (EWOULDBLOCK). The original socket remains available to accept more connection requests.

►►—Socket—('Accept',—*socketid*—)————►◄

Parameters

socketid

The socket descriptor.

Return Values

A string containing a return code, new socket ID, and name is returned. For an IPv6 name, 0s will be returned for scopeid and flowinfo.

Example

```
IPv4 example (socket descriptor 5 is AF_INET)
Socket('Accept',5) == '0 6 AF_INET 5678 9.4.3.2'

IPv6 example (socket descriptor 8 is AF_INET6)
Socket('Accept',8) == '0 6 AF_INET6 5678 0 12ab:0:0:cd30:123:4567:89ab:cedf 0'

C socket call: accept(s, name, namelen)
```

Bind

This call binds a unique local name to the socket with the given *socketid*. After calling *socket*, a socket does not have a name associated with it, but it does belong to an addressing family. The form of the name depends on the addressing family. The *bind* call also allows servers to specify the network interfaces from which they wish to receive UDP packets and TCP connection requests.

The *Bind* call 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* call. 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 only interested in receiving new TCP connections or UDP datagrams that are targeted towards 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.

Note: Even if an application specifies a value of 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. This has a similar effect to the application specifying an explicit IP address on the *Bind* call. For more information, refer to the *z/OS Communications Server: IP Configuration Reference*.

►►—Socket—(—'Bind'—,—*socketid*—,—*name*—)—————►►

Parameters

socketid

The socket descriptor that is to be bound.

name

Refer to Chapter 3, “Designing an iterative server program,” on page 27 for more information.

The IPv4 network address consists of:

domain Must be set to decimal 2 for AF_INET.

portid Set to the port number to which the socket must bind.

ipaddress

Set to the IPv4 address to which the socket must bind. If the *ipaddress* is set to 0, the system selects the source address for the connection.

The IPv6 network address consists of:

domain Must be set to decimal 19 for AF_INET6.

portid Set to the port number to which the socket must bind.

flowinfo

Set to the traffic class and flow label. This field must be set to 0.

ipaddress

Set to the IPv6 address to which the socket must bind. If *ipaddress* is set to IN6ADDR_ANY,::0, the system selects the source address for the connection.

scopeid

Identifies a set of interfaces as appropriate for the scope of the address carried in the *ipaddress* field. A value of 0 indicates the *scopeid* 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 *ipaddress*, *scopeid* may specify a link index which identifies a set of interfaces. For all other address scopes, *scopeid* must be set to 0.

Return Values

A string containing a return code is returned.

Example

```
IPv4 example (socket descriptor 5 is AF_INET)
Socket('Bind',5,'AF_INET 1234 128.228.1.2') == '0'

IPv6 example (socket descriptor 8 is AF_INET6)
Socket('Bind',8,'AF_INET6 1234 0 12f9:0:0:cd30:1823:4567:89ab:FEDF 0') == '0'

C socket call: accept(s, name, namelen)
```

Close

This call shuts down the socket associated with the *socketid*, and frees resources allocated to the socket. If the *socketid* 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 rather than being cleanly closed.

►►—Socket—('Close',—*socketid*—)——►◄

Parameters

socketid

The socket descriptor.

Return Values

A string containing a return code is returned.

Example

```
Socket('Close',6) == '0'
```

C socket call: close(s)

Connect

For stream sockets, the connect call completes the bind for a socket, if bind has not been called, and tries to establish a connection to another socket. For datagram sockets, this call specifies the peer for a socket. If the socket is in blocking mode, this function blocks the caller until the connection is established, or until an error is received. If the socket is in nonblocking mode, this function ends with the return code 36 (EINPROGRESS), or another return code indicating an error.

►►—Socket—(—'Connect'—,—*socketid*—,—*name*—)—►►

Parameters

socketid

The socket descriptor.

name

The IPv4 network address consists of:

domain Must be set to decimal 2 for AF_INET.

portid Set to the port number to which the socket must bind.

ipaddress

Set to the IPv4 address to which the socket must bind.

The IPv6 network address consists of:

domain Must be set to decimal 19 for AF_INET6.

portid Set to the port number to which the socket must bind.

flowinfo

Set to the traffic class and flow label. This field must be set to 0.

ipaddress

Set to the IPv6 address to which the socket must bind.

scopeid Identifies a set of interfaces as appropriate for the scope of the address carried in the *ipaddress* field. A value of 0 indicates the *scopeid* 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 *ipaddress*, *scopeid* may specify a link index which identifies a set of interfaces. For all other address scopes, *scopeid* must be set to 0.

Return Values

A string containing a return code is returned.

Example

```
IPv4 example (socket descriptor 5 is AF_INET)
Socket('Connect',5,'AF_INET 1234 128.228.1.2')      ==  '0'
Socket('Connect',5,'AF_INET 1234 CUNYVM')           ==  '0'
Socket('Connect',5,'AF_INET 1234 CUNYVM.CUNY.EDU')  ==  '0'

IPv6 example (socket descriptor 8 is AF_INET6)
Socket('Connect',8,'AF_INET6 1234 0 12F9:0:0:cd30:1823:4567:89Cb:FEDF
0')          ==  '0'
Socket('Connect',8,'AF_INET6 1234 0 CUNYVM 0')      ==  '0'
Socket('Connect',8,'AF_INET6 1234 0 CUNYVM.CUNY.EDU 0') ==  '0'
```

C socket call: connect(s, name, namelen)

Givesocket

This call transfers the socket with the given socket descriptor to another application. Givesocket makes the specified socket available to a takesocket call issued by another application running on the same host. Any connected stream socket can be given. Normally, givesocket is used by a master program that obtains sockets using the accept call and gives them to slave programs that handle one socket at a time.

►►—Socket—(—GIVESOCKET—,—socketid—,—clientid—)————►◄

Parameters

socketid

The socket descriptor.

clientid

The identifier for another application. The format is *domain*, *userid*, and *subtaskid*. The domain field must be set to a decimal 2 to indicate AF_INET for IPv4 or a decimal 19 to indicate AF_INET6 for IPv6.

Note: A socket given by GIVESOCKET can only be taken by a TAKESOCKET with the same DOMAIN, address family (AF_INET or AF_INET6).

Return Values

A string containing a return code is returned.

Example

```
IPv4 example (socket descriptor 6 is AF_INET)
Socket('GiveSocket',6,'AF_INET USERID2 hisId') == '0'

IPv6 example (socket descriptor 8 is AF_INET6)
Socket('GiveSocket',8,'AF_INET6 USERID2 hisId') == '0'

C socket call: givesocket(s, clientid)
```

Listen

This call applies only to stream sockets and performs two tasks. If Bind has not been called, it completes the bind for a socket. It also creates a connection request queue, whose length is specified as *backlog*, for incoming connection requests. If the queue is full, the connection request is ignored.

►►—Socket—('Listen',—*socketid*—,—*backlog*—)—►►

Parameters

socketid

The socket descriptor.

backlog

The number of pending connect requests. The number is an integer in the range of 0 to the maximum number that can be specified with the SOMAXCONN parameter in TCPIP.PROFILE. The default is 10. The minimum size to the connection request queue (number of pending connect requests) is effectively two. Therefore, if BACKLOG is specified as 0, 1, or 2, then the number of pending connect requests is 2.

Return Values

A string containing a return code is returned.

Example

```
Socket('Listen',5,10) == '0'
```

C socket call: listen(s, backlog)

Shutdown

This call shuts down all or part of a duplex connection. The *how* parameter sets the condition for shutting down the connection to the socket.

►►Socket(—('Shutdown'—,—*socketid*—,—*how*)—)◄◄

If you issue SHUTDOWN for a socket that currently has outstanding socket calls pending, see Table 3 on page 37 to determine the effects of this operation on the outstanding socket calls.

Parameters

socketid

The socket descriptor.

how

The choices are as follows:

- 'SEND', 'SENDING', 'TO', 'WRITE', 'WRITING'—Ends further send operations.
- 'FROM', 'READ', 'READING', 'RECEIVE', 'RECEIVING'—Ends further receive operations.
- 'BOTH' (default)—Ends further send and receive operations.

Return Values

A string containing a return code is returned.

Example

```
Socket('ShutDown',6,'BOTH') == '0'
```

C socket call: shutdown(s, how)

Socket

This call creates an IPv4 or IPv6 socket in the active socket set that is an endpoint for communication, and returns a socket identification.

►► Socket (—'Socket'—, —domain—, —type—, —protocol—) —►►

Parameters

domain

The addressing family must be set to a decimal 2 for AF_INET, or a decimal 19 for AF_INET6.

type

One of the following socket types. SOCK_STREAM or STREAM is the default.

- 'SOCK_STREAM'
- 'STREAM'
- 'SOCK_DGRAM'
- 'DATAGRAM'
- 'SOCK_RAW'
- 'RAW'

Note: If you select the type SOCK_RAW or RAW, the application must be an APF authorized application.

protocol

One of the following protocol names. The *protocol* field should be set to 0 to allow TCP/IP to assign the default protocol for the domain and socket type selected. The protocol defaults are TCP for stream sockets and UDP for datagram sockets. There is no default for raw sockets.

- '0'
- 'IPPROTO_UDP'
- 'UDP'
- 'IPPROTO_TCP'
- 'TCP'

When the socket type is raw, the following values are valid for the *protocol* field.

- 'IPPROTO_ICMP'
- 'IPPROTO_ICMPV6'
- 'IPPROTO_RAW'
- 'RAW'

The following cannot be specified for the *protocol* field when requesting a RAW IPv6 socket:

- 'IPPROTO_HOPOPTS'
- 'IPPROTO_TCP'
- 'IPPROTO_UDP'
- 'IPPROTO_IPV6'
- 'IPPROTO_ROUTING'
- 'IPPROTO_FRAGMENT'
- 'IPPROTO_ESP'
- 'IPPROTO_AH'
- 'IPPROTO_NONE'
- 'IPPROTO_DSTOPTS'

Return Values

A string containing a return code and a new socket ID is returned.

Example

IPv4 example (socket descriptor 5 is AF_INET): `Socket('Socket') == '0 5'`

IPv6 example (socket descriptor 6 is AF_INET6): `Socket('Socket' 'AF_INET6' 'SOCK_STREAM' '0') == '0 6'`

C socket call: `socket(domain, type, protocol)`

Takesocket

This call acquires a socket from another program that obtains the other program's *clientid* and *socketid* by a method not defined by TCP/IP. After a successful call to *takesocket*, the other application must close the socket.

►►—Socket—(—'Takesocket'—,—*clientid*—,—*socketid*—)—►►

Parameters

clientid

The identifier for another application. The identifier contains a domain, userid, and subtaskid. The domain field must be set to a decimal 2 to indicate AF_INET for IPv4 or a decimal 19 to indicate AF_INET6 for IPv6.

Note: The TAKESOCKET can only acquire a socket of the same DOMAIN, address family from a GIVESOCKET.

socketid

The socket descriptor belonging to another *clientid*.

Return Values

A string containing a return code and a new socket ID is returned.

Example

```
IPv4 example (socket descriptor 6 is AF_INET)
Socket('TakeSocket','AF_INET USERID1 myId',6) == '0 7'

IPv6 example (socket descriptor 8 is AF_INET6)
Socket('TakeSocket','AF_INET6 USERID1 myId',8) == '0 7'

C socket call: takesocket(clientid, hisdesc)
```

Coding calls to exchange data for sockets

On a connected stream socket and on datagram sockets, you can send and receive data. You can use the calls in this section to send, receive, read, and write data.

Read

This call reads up to *maxlength* bytes of data. This is the conventional TCP/IP read data operation. If less than the number of bytes requested is available, the call returns the number currently available. If data is not available at the socket, the call waits for data to arrive and blocks the caller, unless the socket is in nonblocking mode. If the length is 0, the other side of the call closed the stream socket.

For datagram sockets, Read returns the entire datagram that was sent, providing that the datagram fits into the specified buffer.

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 the return values string. Therefore, programs using stream sockets should place this call in a loop that repeats until all data has been received.

►►Socket—('Read',—*socketid*—,—*maxlength*)—►►

Parameters

socketid

The socket descriptor.

maxlength

The maximum data length. The length is a number in the range 1—100 000. The default is 10 000.

Return Values

A string containing a return code, the data length, and the data read is returned.

Example

```
Socket('Read',6)           ==  '0 21 This is the data line'
```

```
C socket call: read(s, buf, len)
```


Recv

This call receives up to *maxlength* bytes of data from the incoming message. RECV applies only to connected sockets. For additional control of the incoming data, you can use RECV to:

- Peek at the incoming data without having it removed from the buffer.
- Read out-of-band data.

If more than the number of bytes requested is available on a datagram socket, the call discards excess bytes. If less than the number of bytes requested is available, the call returns the number of bytes currently available. If data is not available at the socket, the call waits for data to arrive and blocks the caller, unless the socket is in nonblocking mode. If the length is 0, the other side closed the stream socket.

►►Socket(—('Recv',—*socketid*—,—*maxlength*—,—*recvflags*—)—►►

Parameters

socketid

The socket descriptor.

maxlength

The maximum data length. The length is a number in the range 1—100 000. The default is 10 000.

recvflags

Specifies the following receive flags:

'MSG_OOB', 'OOB', 'OUT_OF_BAND'

Read out-of-band data on the socket. Only stream sockets created in the AF_INET domain support out-of-band data.

'MSG_PEEK', 'PEEK'

Look at the data on the socket but do not change or destroy it. The next Recv call can read the same data.

" (default)

Receive the data. No flag is set.

Return Values

A string containing a return code, the data length, and the data received is returned.

Example

```
Socket('Recv',6)           == '0 21 This is the data line'
Socket('Recv',6,, 'PEEK OOB') == '0 24 This is out-of-band data'
```

C socket call: `recv(s, buf, len, flags)`

Recvfrom

This call receives the incoming message, up to *maxlength* bytes of data. If more than the number of bytes requested is available on a datagram socket, the call discards excess bytes. If less than the number of bytes requested is available, the call returns the number of bytes available. If data is not available at the socket, Recvfrom waits for data to arrive and blocks the caller, unless the socket is in nonblocking mode.

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 UDP sockets, RECVFROM returns the entire datagram that was sent if it will fit in the buffer. If the datagram packet is too large to fit in the buffer, the excess bytes are discarded.

For datagram protocols, recvfrom() and recvmsg() return 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 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).

►►—Socket—('Recvfrom',—*socketid*—, —*maxlength*—, —*recvflags*—) —►►

Parameters

socketid

The socket descriptor.

maxlength

The maximum data length. The length is a number in the range of 1 to 100 000. The default is 10 000.

recvflags

Specifies the following receive flags:

'MSG_OOB', 'OOB', 'OUT_OF_BAND'

Read out-of-band data on the socket. Only stream sockets created in the AF_INET domain support out-of-band data.

'MSG_PEEK', 'PEEK'

Look at the data on the socket but do not change or destroy it.

" (default)

Receive the data. No flag is set.

Return Values

A string containing a return code, a name, a length, and the data is returned. For an IPv6 name, 0s will be returned for scopeid and flowinfo.

Example

```
IPv4 example (socket descriptor 6 is AF_INET)
Socket('RecvFrom',6)      ==  '0 AF_INET 5678 9.4.3.2 9 Data line'

IPv6 example (socket descriptor 8 is AF_INET6)
Socket('RecvFrom',8)      ==  '0 AF_INET6 5678 0 12f9:0:0:c30:123:457:9Cb:bef 0 9 Data line'

C socket call: recvfrom(s, buf, len, flags, name, namelen)
```

Send

This call sends the outgoing data message to a connected socket. If Send cannot send the number of bytes of data that is requested, it waits until sending is possible. This blocks the caller, unless the socket is in nonblocking mode. For datagram sockets, the socket should not be in blocking mode.

►►—Socket—('Send'—,—*socketid*—,—*data*—,—*sendflags*—)—►►

Parameters

socketid

The socket descriptor.

data

The message string for data exchange.

sendflags

Specifies the following send flags:

'MSG_OOB', 'OOB', 'OUT_OF_BAND'

Sends out-of-band data on sockets that support it. Only stream sockets created in the AF_INET domain support out-of-band data.

'MSG_DONTROUTE', 'DONTROUTE'

Do not route the data. Routing is handled by the calling program.

" (default)

Send the data. No flag is set.

Return Values

A string containing a return code and the length of the data sent is returned.

Example

```
Socket('Send',6,'Some text')           == '0 9'  
Socket('Send',6,'Out-of-band data','OOB') == '0 16'
```

C socket call: send(s, buf, len, flags)

Sendto

Sendto is similar to Send, except that it includes the destination address parameter. You can use the destination address if you want to use the Sendto call to send datagrams on a UDP socket, regardless of whether or not the socket is connected. For datagram sockets, the socket should not be in blocking mode.

Use the FLAGS parameter 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 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 the return values. Therefore, programs using stream sockets should place Sendto in a loop that repeats the call until all data has been sent.

►►Socket—('Sendto'—,—socketid—,—data—,—sendflags—,—name—)——►►

Parameters

socketid

The socket descriptor.

data

The message string for data exchange.

sendflags

Specifies the following send flags:

'MSG_DONTROUTE', 'DONTROUTE'

Do not route the data. Routing is handled by the calling program.

" (default)

Send the data. No flag is set.

'MSG_OOB', 'OOB', 'OUT_OF_BAND'

Sends out-of-band data on sockets that support it. Only stream sockets created in the AF_INET domain support out-of-band data.

name

The IPv4 network address in the format:

domain Must be set to a decimal 2 for AF_INET.

portid Set to the port number to which the socket must bind.

ipaddress

Set to the IP address to which the socket must bind.

The IPv6 network address consists of:

domain Must be set to decimal 19 for AF_INET6.

portid Set to the port number to which the socket must bind.

flowinfo

Set to the traffic class and flow label. This field must be set to 0.

ipaddress

Set to the IPv6 address to which the socket must bind.

scopeid Identifies a set of interfaces as appropriate for the scope of the address carried in the *ipaddress* field. A value of 0 indicates the *scopeid* 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 *ipaddress*, *scopeid* may specify a link index which identifies a set of interfaces. For all other address scopes, *scopeid* must be set to 0.

Return Values

A string containing a return code and a length is returned.

Example

```
IPv4 example (socket descriptor 6 is AF_INET)
Socket('SendTo',6,'Some text',,AF_INET 5678 9.4.3.2'           == '0 9'
Socket('SendTo',6,'Some text',,AF_INET 5678 ZURLVM1'           == '0 9'
Socket('SendTo',6,'Some text',,AF_INET 5678 ZURLVM1.ZURICH.IBM.COM' == '0 9'

IPv6 example (socket descriptor 8 is AF_INET6)
Socket('SendTo',8,'Some text',,AF_INET6 5678 0 12f9:0:0:c30:123:457:9Cb:bef 0'
== '0 9'
Socket('SendTo',8,'Some text',,AF_INET6 5678 0 ZURLVM1 0'      == '0 9'
Socket('SendTo',8,'Some text',,AF_INET6 5678 0 ZURLVM1.ZURICH.IBM.COM 0' == '0 9'

C socket call: sendto(s, buf, len, flags, name, namelen)
```

Write

This call writes the given number of bytes of data for a connected socket. This call is similar to SEND, except that it lacks the control flags available with SEND.

Write waits until conditions are suitable for writing data. This blocks the caller, unless the socket is in nonblocking mode. For datagram sockets, the socket should not be in blocking mode.

►►—Socket—('Write',—*socketid*—,—*data*—)—————►◄

Parameters

socketid

The socket descriptor.

data

The message string for data exchange.

Return Values

A string containing a return code and the length of the data written is returned.

Example

```
Socket('Write',6,'Some text') == '0 9'
```

C socket call: write(s, buf, len)

Coding calls to resolve names for REXX sockets

You can use the name resolution calls to get information such as name, address, client identification, and host name. You can also use the name resolution calls to resolve an Internet protocol address (*ipaddress*) to a symbolic name or a symbolic name to an *ipaddress*.

Getaddrinfo

This 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.

```
➤➤Socket—('GETADDRINFO' —, —————)————→
                        |node| |,—service| |,—flags|
➤—————)————→
|,—family| |,—socktype| |,—protocol|
```

Parameters

node

An input parameter. Storage up to 255 bytes long that contains the host name being queried. If the `AI_NUMERICHOST` flag is specified, then *node* should contain the queried hosts IP address. Both *node* and *service* are optional, but one or both must be specified.

service

An input parameter. Storage up to 32 bytes long that contains the service name being queried. If the `AI_NUMERICSERV` flag is specified, then *service* should contain the queried port number. Both *node* and *service* are optional, however one or both must be specified.

FLAGS

Must have the value of 0 or the bitwise, OR of one or more of the following:

AI_PASSIVE

- Specifies how to fill in the returned socket address structures in the returned address information. 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 pointwill be set to `INADDR_ANY` for an IPv4 address or `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-node protocol) or for a `CONNECT`, or `SENDTO` 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 will be set to the default loopback address for an IPv4 address (127.0.0.0) or the default loopback address for an IPv6 address (::1).
- This flag is ignored if the *node* argument is specified.

AI_CANONNAMEOK

- If this flag is specified and the *node* name argument is specified, then the `GETADDRINFO` call attempts to determine the canonical name corresponding to the *node* argument.

AI_NUMERICHOST

- 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

- 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

- If this flag is specified along with the *family* field with the value of AF_INET6, or a value of AF_UNSPEC when IPv6 is supported on the system, the caller will accept IPv4-mapped IPv6 addresses. When the AI_ALL flag is not also specified and if no IPv6 addresses are found, a query is made for IPv4 addresses. If any are found they are returned as IPv4-mapped IPv6 addresses.
- If the *family* field does not have the value of AF_INET6 or the *family* field contains AF_UNSPEC but IPv6 is not supported on the system, this flag is ignored.

AI_ALL

- When the *family* field has a value of AF_INET6 and AI_ALL is set, the AI_V4MAPPED flag must also be set to indicate the caller accepts all addresses (IPv6 and IPv4-mapped IPv6 addresses). When the *family* field has a value of AF_UNSPEC when the system supports IPv6 and AI_ALL is set, the caller will accept IPv6 addresses and either IPv4 addresses (if AI_MAPPED is not specified) or IPv4-mapped IPv6 addresses (if AI_V4MAPPED is specified). A query is first made for IPv6 addresses and if successful, the IPv6 addresses are returned. Another query is then made for IPv4 addresses, and any IPv4 addresses found are returned as either IPv4-mapped IPv6 addresses (if AI_V4MAPPED is also specified) or as IPv4 addresses (if AI_V4MAPPED is not specified).
- If the *family* field does not have the value of AF_INET6 or does not have the value of AF_UNSPEC when the system supports IPv6, this flag is ignored.

AI_ADDRCONFIG

If this flag is specified, then a query for IPv6 on the *node* will occur if the Resolver determines whether either of the following is true:

- If the system is IPv6 enabled and has at least one IPv6 interface, 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, the Resolver will make a query for IPv4 (A DNS) records.

The loopback address is not considered in this case as a valid interface.

family

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 0 indicates AF_UNSPEC. The value of 2 indicates AF_INET and the value of 19 indicates AF_INET6.

socktype

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 are the acceptable socket types:

Type name	Decimal value	Description
SOCK_STREAM	1	stream socket
SOCK_DGRAM	2	datagram socket
SOCK_RAW	3	raw-protocol interface

Anything else will fail with return code EAI_SOCKTYPE. Note that although SOCK_RAW will be accepted, it will only be valid when *service* is numeric (*service*=23). A lookup for a *service* name will never occur in the services file (for example, *hlq.ETC.SERVICES*) using any *protocol* value other than SOCK_STREAM or SOCK_DGRAM.

If *socktype* and *protocol* are both specified as 0, then GETADDRINFO will proceed as follows:

- If *service* is null, or if *service* is numeric, then any returned addinfos will default to a specification of *socktype* as SOCK_STREAM.
- If *service* is specified as a service name (for example, FTP), then GETADDRINFO call will search the appropriate services file (for example, *hlq.ETC.SERVICES*) twice. The first search will use the SOCK_STREAM as the protocol, and the second search will use SOCK_DGRAM as the protocol. No default socket type provision exists in this case.

If both *socktype* and *protocol* are specified as nonzero then they should be compatible, regardless of the value specified by *service*. In this context, *compatible* means one of the following:

- *socktype*=SOCK_STREAM and *protocol*=IPPROTO_TCP
- *socktype*=SOCK_DGRAM and *protocol*=IPPROTO_UDP
- *socktype*=SOCK_RAW, in which case *protocol* can be anything

protocol

Used to limit the returned information to a specific protocol. A value of 0 means that the caller will accept any protocol.

The following are acceptable protocols:

Protocol name	Decimal value	Description
IPPROTO_TCP	6	tcp
IPPROTO_UDP	17	user datagram

If *socktype* is 0 and *protocol* is not 0, the only acceptable input values for *protocol* are IPPROTO_TCP and IPPROTO_UDP; otherwise, the GETADDRINFO call will be failed with return code of EAI_BADFLAGS.

If *protocol* and *socktype* are both specified as 0, then GETADDRINFO will proceed as follows:

- If *service* is null, or if *service* is numeric, then any returned addinfos will default to a specification of *socktype* as SOCK_STREAM.
- If *service* is specified as a service name (for example, FTP), then GETADDRINFO 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 provision exists in this case.

If both *protocol* and *socktype* are specified as nonzero, they should be compatible, regardless of the value specified by *service*. In this context, *compatible* means one of the following:

- *socktype*=SOCK_STREAM and *protocol*=IPPROTO_TCP
- *socktype*=SOCK_DGRAM and *protocol*=IPPROTO_UDP
- *socktype*=SOCK_RAW, in which case *protocol* can be anything.

If the lookup for the value specified in *service* fails [for example, the service name does not appear in the appropriate services file (such as, *hlq.ETC.SERVICES*) using the input protocol], then the GETADDRINFO call will be failed with return code of EAI_SERVICE.

Return Values

A string containing a return code, canon name and a name or list of names. Zeros will be returned for scopeid and flowinfo.

Example

```
Socket('GetAddrInfo','ZURLVM1.ZURICH.IBM.COM','daytime',, 'AF_INET6',
      'SOCK_STREAM','IPROTO_IPV6') ==
'0 "" AF_INET6 1234 0 12F9:0:0:c30:123:457:9Cb:ace 0'
```

```
Socket('GetAddrInfo','ZURLVM1.ZURICH.IBM.COM',, 'AF_INET6','SOCK_STREAM',
      'IPROTO_IPV6') ==
'0 "" AF_INET6 0 12F9:0:0:c30:123:457:9Cb:1111 0
AF_INET6 1234 0 12F9:0:0:c30:123:457:9Cb:1112 0
AF_INET6 1234 0 12F9:0:0:c30:123:457:9Cb:1113 0
AF_INET6 1234 0 12F9:0:0:c30:123:457:9Cb:1114 0'
```

```
Socket('GetAddrInfo','ZURLVM1',,,,,) ==
'0 "" AF_INET6 1234 0 1F9:0:0:c30:123:457:9Cb:abc 0'
```

```
Socket('GetAddrInfo','ZURLVM1.ZURICH.IBM.COM',, 'AI_CANNONNAMEOK',
      'AF_INET6','SOCK_STREAM','IPROTO_IPV6') ==
'0 "ZVM1.ZURICH.IBM.COM" AF_INET6 1234 0
12F9:0:0:c30:123:457:9Cb:ace 0'
```

C socket call: getaddrinfo(nodename, servname, hints, res)

Getclientid

This call returns the calling program's TCP/IP virtual machine identifier.

►►—Socket—('Getclientid',—domain—)————►►

Parameters

domain

On input, this is an optional parameter for AF_INET and a required parameter for AF_INET6. It is necessary to specify the domain of the client. The name must be AF_INET for IPv4 or AF_INET6 for IPv6.

Return Values

A string containing a return code and a client identification is returned.

Example

```
IPv4 example
Socket('GetClientId')          ==  '0 AF_INET USERID1 myId'

IPv6 example
Socket('GetClientId')          ==  '0 AF_INET6 USERID1 myId'

C socket call: getclientid(domain, clientid)
```

Getdomainname

This call returns the domain name for the processor running the program.

►►—Socket—('Getdomainname'—)—————►◄

Parameters

None

Return Values

A string containing a return code and a domain name is returned.

Example

```
Socket('GetDomainName')      ==  '0 ZURICH.IBM.COM'
```

C socket call: getdomainname(name, namelen)

Gethostbyaddr

This call resolves the IPv4 IP address through a name server, if one is present.

►►—Socket—('Gethostbyaddr'—,—*ipaddress*—*domain*)—►►

Parameters

ipaddress

The *ipaddress* in dotted-decimal notation.

domain

The socket domain. The name must be AF_INET.

Note: If the *ipaddress* is not found through the name server, the resolver searches the local hosts table.

Return Values

A string containing a return code and a full host name is returned.

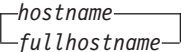
Example

```
Socket('GetHostByAddr','128.228.1.2') == '0 CUNYVM.CUNY.EDU'
```

```
C socket call: gethostbyaddr(addr, addrlen, domain)
```

Gethostbyname

This call tries to resolve the host name through a name server, if one is present. Gethostbyname returns all *ipaddresses* for multihome hosts. Any trailing blanks will be removed from the specified name prior to trying to resolve it to an IPv4 IP address. The addresses are separated by blanks.

►►—Socket—(—'Gethostbyname'—, ——)——►►

Parameters

hostname

The host processor name as a character string. The maximum length is 255 characters.

fullhostname

A fully qualified host name in the form hostname.domainname. The maximum length is 255 characters.

Note: If the hostname or fullhostname is not found through the name server, the resolver searches the local hosts table. For more information, refer to *z/OS Communications Server: IP Configuration Reference*.

Return Values

A string containing a return code and an *ipaddress* list is returned.

Example

```
Socket('GetHostByName','CUNYVM')          == '0 128.228.1.2'
Socket('GetHostByName','CUNYVM.CUNY.EDU') == '0 128.228.1.2'
```

C socket call: gethostbyname(name)

Gethostid

This call returns the *ipaddress* for the current host. This address is the default home *ipaddress*.

►►—Socket—(—'Gethostid'—)—————►◄

Parameters

None

Return Values

A string containing a return code and the *ipaddress* is returned.

Example

```
Socket('Gethostid')      ==    '0 9.4.3.2'
```

C socket call: gethostid()

Gethostname

This call returns the name of the host processor for the program.

Note: The hostname returned is the hostname the TCPIP stack learned at startup from the TCPIP.DATA file that was found.

►►—Socket—(—'Gethostname'—)—————►◄

Parameters

None

Return Values

A string containing a return code and the host name is returned.

Example

```
Socket('GetHostName')      ==  '0 ZURLVM1'
```

C socket call: gethostname(name, namelen)

Getnameinfo

This call returns the node name and service location of an IPv6 socket address.

```
►► Socket ( ('Getnameinfo', —name— [flags] ) )
```

Parameters

name

The IPv4 network address in the following format:

domain

Must be set to a decimal 2 for AF_INET.

portid Set to the port number to which the socket must bind.

ipaddress

Set to the IP address to which the socket must bind.

The IPv6 network address consists of:

domain

Must be set to a decimal 19 for AF_INET6.

flowinfo

Set to the traffic class and flow label. This field is currently not implemented.

ipaddress

Set to the IPv6 address to which the socket must bind.

scopeid

Set to the link scope for an IPv6 address. If *scopeid* is specified and the destination is not link local, the socket call will fail.

flags

This is an optional field.

'NI_NOFQDN'

Return the host name portion of the fully qualified domain name.

'NI_NUMERICHOST'

Only return the numeric form of host's address.

'NI_NAMEREQD'

Return an error if the host's name cannot be located.

'NI_NUMERICSERV'

Only return the numeric form of the service address.

'NI_DGRAM'

Indicates that the service is a datagram service. The default behavior is to assume that the service is a stream service.

Return Values

A string containing a return code host name and service name is returned.

Example

```
Socket('GetNameInfo','AF_INET6 1234 0 12F9:0:0:c30:123:457:9Cb:ace 0','')  
==      '0 CUNYVM.CUNY.EDU quote'
```

```
Socket('GetNameInfo','AF_INET6 1234 0 12F9:0:0:c30:123:457:9Cb:ace 0',  
      'NI_NUMERICSERV') ==      '0 CUNYVM.CUNY.EDU 17'
```

C socket call: getnameinfo(name,namelen,host,hostlen,serv,servlen,flags)

Getpeername

This call returns the name of the peer that is connected to the given socket.

►►—Socket—('Getpeername',—*socketid*—)—————►◄

Parameters

socketid

The socket descriptor.

Return Values

A string containing a return code and a peer name is returned. For an IPv6 name, 0's will be returned for scopeid and flowinfo.

Example

IPv4 example (socket descriptor 6 is AF_INET)

```
Socket('GetPeerName',6) == '0 AF_INET 1234 128.228.1.2'
```

IPv6 example (socket descriptor 8 is AF_INET6)

```
Socket('GetPeerName',8) == '0 AF_INET6 1234 0  
12F9:0:0:cd30:1823:4567:89Cb:FEDF 0'
```

C socket call: `getpeername(s, name, namelen)`

Getprotobyname

This call returns the number of a network protocol when you specify a protocol name.

►►—Socket—(—'Getprotobyname'—,—*protocolname*—)—————►◄

Parameters

protocolname

The name of a network protocol. The names TCP, UDP, and ICMP are valid.

Return Values

A string containing a return code and a protocol number is returned. If the protocol name specified on the call is incorrect, a 0 return code and a 0 for the protocol number is returned.

Example

```
Socket('GetProtoByName','TCP')           ==    '0 6'
```

C socket call: getprotobyname(name)

Getprotobynumber

This call returns the name of a network protocol when you specify a protocol number.

►►—Socket—(—'Getprotobynumber'—,—*protocolnumber*—)————►◄

Parameters

protocolnumber

A network protocol number. The number is a positive integer.

Return Values

A string containing a return code and a protocol name is returned. If the protocol number specified on the call is incorrect, a 0 return code and a null protocol number string is returned.

Example

```
Socket('GetProtoByNumber',6)           ==    '0 TCP'
```

C socket call: getprotobynumber(name)

Getservbyname

This call returns the service name, the port, and the network protocol name.

►►—Socket—(—'Getservbyname'—,—*servicename*—,—*protocolname*)—►►

Parameters

servicename

A service name such as FTP.

protocolname

A network protocol name, such as TCP, UDP, or ICMP.

Return Values

A string containing a return code, service name, port ID, and protocol name is returned. If the service name specified on the call is incorrect, a 0 return code and a null service name string are returned.

Example

```
Socket('GetServByName','ftp','tcp')      ==  '0 FTP 21 TCP'
```

C socket call: getservbyname(name, proto)

Getservbyport

This call returns the name of a service, port, and network protocol. If the port ID specified on the call is incorrect, a 0 return code and a null port ID string is returned. If the port ID and the protocol name are correct, but the protocol name does not exist for the port ID specified, the call returns the proper protocol service name and port ID.

If both port ID and protocolname are correct, but the protocolname does not exist for the specified port ID, the call returns the proper protocolname, port ID, and service name.

►►—Socket—('Getservbyport'—,—*portid*—,—*protocolname*—)—►►

Parameters

portid

A port number. The number must be an integer in the range of 0 to 65535.

protocolname

A network protocol name, such as TCP, UDP, or ICMP.

Return Values

A string containing a return code, service name, port ID, and protocol name is returned.

Example

```
Socket('GetServByPort',21,'tcp')           ==   '0 FTP 21 TCP'
```

```
C socket call: getservbyport(name, proto)
```

Getsockname

This call returns the name to which the given socket was bound. Stream sockets are not assigned a name until after a successful call to bind, connect, or accept.

►►—Socket—(—'Getsockname'—,—*socketid*—)—►►

Parameters

socketid

The socket descriptor

Return Values

A string containing a return code and a socket name is returned. For an IPv6 name, 0's will be returned for scopeid and flowinfo.

Example

```
IPv4 example (socket descriptor 6 is AF_INET)
Socket('GetSockName',6)      ==  '0 AF_INET 5678 9.4.3.2'
```

```
IPv4 example (socket descriptor 8 is AF_INET6)
Socket('GetSockName',8)      ==  '0 AF_INET6 5678 0 12F9:0:0:cd30:1823:4567:89Cb:FEDF 0'
```

```
C socket call: getsockname(s,name,namelen)
```

Resolve

This call resolves the host name through a name server, if a name server is present.

►► Socket(—('Resolve'—, —
 | *ipaddress* —
 | *hostname* —
 | *fullhostname* —) —————►

Parameters

ipaddress

The *ipaddress* in dotted decimal notation.

hostname

The name of a host processor. The maximum length is 255 characters.

fullhostname

A fully qualified host name in the form *hostname.domainname*. The maximum length is 255 characters.

Note: If an *ipaddress*, *hostname*, or *fullhostname* is not present, the resolver searches the local hosts table.

Return Values

A string containing a return code, an *ipaddress*, and a full host name is returned.

Example

```
Socket('Resolve','128.228.1.2')    == '0 128.228.1.2 CUNYVM.CUNY.EDU'  
Socket('Resolve','CUNYVM')         == '0 128.228.1.2 CUNYVM.CUNY.EDU'  
Socket('Resolve','CUNYVM.CUNY.EDU') == '0 128.228.1.2 CUNYVM.CUNY.EDU'
```

Coding calls to manage configuration, options, and modes

Use this group of calls to obtain the version number of the REXX/SOCKETS function package, get socket options, set socket options, or socket mode of operation. There are also socket calls to determine the network configuration.

Fcntl

This call allows you to control the operating characteristics of a socket. You can use Fcntl to set the blocking or nonblocking mode for a socket.

►►Socket—('Fcntl'—,—*socketid*—,—*cmd*—,—*fvalue*)—►►

Parameters

socketid

The socket descriptor for this socket.

cmd

The command. The options are F_SETFL or F_GETFL.

'F_SETFL'

Sets the status flags for the socket. One flag, FNDELAY, can be set.

'F_GETFL'

Gets the status for the socket. One flag, FNDELAY, can be retrieved.

The FNDELAY flag marks the socket as being in nonblocking mode. If data is not present on calls that can block, such as read, readv, and recv, fcntl returns error code 35 (EWOULDBLOCK).

fvalue

The following operating characteristic values:

- 'BLOCKING '
- '0'
- 'NON-BLOCKING'
- 'FNDELAY'

Return Values

A string containing a return code and *fvalue* for F_GETFL is returned.

Example

```
Socket('Fcntl',5,'F_SETFL','NON-BLOCKING') == '0'
Socket('Fcntl',5,'F_GETFL') == '0 NON-BLOCKING'
```

C socket call: fcntl(s, cmd, data)

Getsockopt

The Getsockopt call gets the active options for a socket that were set with the Setsockopt call.

Multiple options can be associated with each socket. These options are described below. You must specify the option that you want when you issue the Getsockopt call.

►►—Socket—(—'Getsockopt'—,—*socketid*—,—*level*—,—*optname*—)—►►

Parameters

socketid

The socket descriptor for the socket requiring options.

level

The protocol level for which the socket is being set. The levels are SOL_SOCKET, IPPROTO_TCP, IPPROTO_IP, and IPPROTO_IPV6.

- All commands beginning with SO_ are for protocol level SOL_SOCKET.
- All commands beginning with TCP_ are for protocol level IPPROTO_TCP.
- All commands beginning with IP_ are for protocol level IPPROTO_IP.
- All commands beginning with IPV6_ are for protocol level IPPROTO_IPV6.

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 803 for the numeric values of **optname**.

Return Values

A string containing a return code and an option value is returned.

Example

```
Example (socket descriptor 5 is AF_INET)
Socket('GetSockOpt',5,'Sol_Socket','So_ASCII')      == '0 0n STANDARD'
Socket('GetSockOpt',5,'Sol_Socket','So_Broadcast')  == '0 0n'
Socket('GetSockOpt',5,'Sol_Socket','So_Error')       == '0 0'
Socket('GetSockOpt',5,'Sol_Socket','So_Linger')      == '0 0n 60'
Socket('GetSockOpt',5,'Sol_Socket','So_Sndbuf')      == '0 8192'
Socket('GetSockOpt',5,'Sol_Socket','So_Type')        == '0 SOCK_STREAM'
```

```
Example (socket descriptor 8 is AF_INET6)
Socket('GetSockOpt',8,'IPPROTO_IPV6','IPV6_MULTICAST_HOPS') == '0 7'
Socket('GetSockOpt',8,'IPPROTO_IPV6','IPV6_MULTICAST_IF')  == '0 4'
Socket('GetSockOpt',8,'IPPROTO_IPV6','IPV6_MULTICAST_LOOP') == '0 1'
Socket('GetSockOpt',8,'IPPROTO_IPV6','IPV6_UNICAST_HOPS')  == '0 10'
Socket('GetSockOpt',8,'IPPROTO_IPV6','IPV6_V6ONLY')        == '0 1'
```

C socket call: getsockopt(s, level, optname, optval, optlen)

Note: OPTLEN is not a REXX parameter. All references to OPTLEN or the length of data in *optval* in the table below should be ignored.

Table 22. 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 <i>hlq</i>.SEZAINST(CBLOCK) for the PL/I example of IP_MREQ.</p> <p>The IP_MREQ definition for COBOL:</p> <pre>01 IP-MREQ. 05 IMR-MULTIADDR PIC 9(8) BINARY. 05 IMR-INTERFACE PIC 9(8) BINARY.</pre>	<p>N/A</p>
<p>IP_DROP_MEMBERSHIP</p> <p>Use this option to enable an application to exit 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 <i>hlq</i>.SEZAINST(CBLOCK) for the PL/I example of IP_MREQ.</p> <p>The IP_MREQ definition for COBOL:</p> <pre>01 IP-MREQ. 05 IMR-MULTIADDR PIC 9(8) BINARY. 05 IMR-INTERFACE PIC 9(8) BINARY.</pre>	<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 are 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 22. *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>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 <i>hlq</i>.SEZAINST(CBLOCK) for the PL/I example of IPV6_MREQ.</p> <p>The IPV6_MREQ definition for COBOL:</p> <pre> 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. </pre>	<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 <i>hlq</i>.SEZAINST(CBLOCK) for the PL/I example of IPV6_MREQ.</p> <p>The IPV6_MREQ definition for COBOL:</p> <pre> 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. </pre>	<p>N/A</p>

Table 22. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
IPV6_MULTICAST_HOPS Use to set or obtain the hop limit used for outgoing multicast packets. This is an IPv6-only socket option.	Contains a 4-byte binary value specifying the multicast hops. If not specified, then the default is 1 hop. -1 indicates use stack default. 0 - 255 is the valid hop limit range. 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.	Contains a 4-byte binary value in the range from 0 to 255 indicating the number of multicast hops.
IPV6_MULTICAST_IF Use this option to set or obtain the index of the IPv6 interface used for sending outbound multicast datagrams from the socket application. This is an IPv6-only socket option.	Contains a 4-byte binary field containing an IPv6 interface index number.	Contains a 4-byte binary field containing an IPv6 interface index number.
IPV6_MULTICAST_LOOP 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. This is an IPv6-only socket option.	A 4-byte binary field. To enable, set to 1. To disable, set to 0.	A 4-byte binary field. If enabled, contains a 1. If disabled, contains a 0.
IPV6_UNICAST_HOPS Use this option to set or obtain the hop limit used for outgoing unicast IPv6 packets. This is an IPv6-only socket option.	Contains a 4-byte binary value specifying the unicast hops. If not specified, then the default is 1 hop. -1 indicates use stack default. 0 - 255 is the valid hop limit range. Note: APF authorized applications are permitted to set a hop limit that exceeds the system configured default. CICS applications cannot execute as APF authorized.	Contains a 4-byte binary value in the range from 0 to 255 indicating the number of unicast hops.
IPV6_V6ONLY 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. This is an IPv6-only socket option.	A 4-byte binary field. To enable, set to 1. To disable, set to 0.	A 4-byte binary field. If enabled, contains a 1. If disabled, contains a 0.

Table 22. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
SO_ASCII 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. Note: This is a REXX-only socket option.	To enable, set to ON. To disable, set to OFF. 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.	If enabled, contains ON. If disabled, contains OFF. 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.
SO_BROADCAST 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. Note: This option has no meaning for stream sockets.	A 4-byte binary field. To enable, set to 1 or a positive value. To disable, set to 0.	A 4-byte field. If enabled, contains a 1. If disabled, contains a 0.
SO_DEBUG 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. Notes: <ol style="list-style-type: none"> 1. This is a REXX-only socket option. 2. This option has meaning only for stream sockets. 	To enable, set to ON. To disable, set to OFF.	If enabled, contains ON. If disabled, contains OFF.
SO_EBCDIC 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. Note: This is a REXX-only socket option.	To enable, set to ON. To disable, set to OFF. 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.	If enabled, contains ON. If disabled, contains OFF. 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.
SO_ERROR 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.	N/A	A 4-byte binary field containing the most recent ERRNO for the socket.

Table 22. *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 keepalive 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 keepalive 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 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 22. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_OOBLIN</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 <i>RECV</i> or a <i>RECVFROM</i> even if the OOB flag is not set in the <i>RECV</i> or the <i>RECVFROM</i>.</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 <i>RECV</i> or a <i>RECVFROM</i> only when the OOB flag is set in the <i>RECV</i> or the <i>RECVFROM</i>.</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_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 <i>SETSOCKOPT</i> call:</p> <ul style="list-style-type: none"> • <i>TCPRCVBufsize</i> keyword on the <i>TCPCONFIG</i> statement in the <i>PROFILE.TCPIP</i> data set for a TCP Socket • <i>UDPRCVBufsize</i> keyword on the <i>UDPCONFIG</i> statement in the <i>PROFILE.TCPIP</i> data set for a UDP Socket • The default of 65 535 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 22. 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, refer to 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 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:</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 65 535 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>
<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>

Table 22. *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>

ioctl

This call allows you to control the operating characteristics of a socket. You can use this call to set blocking or nonblocking mode and to control the operations characteristics determined by the *icmd* command.

►►Socket—('Ioctl'—,—*socketid*—,—*icmd*—,—*ivalue*)—►►

Parameters

socketid

The socket descriptor for this socket.

icmd

The following operating characteristics commands:

'FIONBIO'

Use FIONBIO to set or clear nonblocking for socket I/O. Set or clear is determined by the value in *ivalue*. The *ivalue* parameter can be ON or OFF.

'FIONREAD'

Use FIONREAD to get the number of immediately readable bytes of data for the socket and return it in *ivalue*.

'SIOCATMARK'

Use SIOCATMARK to determine if the current location in the input data is pointing to out-of-band data. The command returns YES or NO in the *ivalue* field.

'SIOCGIFADDR'

Use SIOCGIFADDR to get the IPv4 network interface address. The *ivalue* parameter returns the address in the format interface, domain, port, and IP address.

'SIOCGIFBRDADDR'

Use SIOCGIFBRDADDR to get the IPv4 network interface broadcast address. The *ivalue* parameter returns the address in the format interface, domain, port, and Internet address.

'SIOCGIFCONF'

Use SIOCGIFCONF to get the IPv4 network interface configuration. The *ivalue* parameter contains the maximum number of interfaces that is returned. The call returns a list of interfaces in the format interface, domain, port, and Internet address.

'SIOCGIFDSTADDR'

Use SIOCGIFDSTADDR to get the IPv4 network interface destination address. The *ivalue* parameter returns the address in the format interface, domain, port, and Internet address.

'SIOCGIFNAMEINDEX'

Use the SIOCGIFNAMEINDEX IOCTL to get all the interface names and indexes including local loopback but excluding VIPAs. Upon successful completion of this call, the stack will return the interface indexes and names found.

ivalue

The operating characteristics value. The operating characteristics value depends on the value specified for *icmd*. The *ivalue* parameter can be used as input or output or both on the same call.

Return Values

A string containing a return code and a list of interface indexes and names is returned.

Example

```
Socket('Ioctl',5,'FionBio','0n')      == '0'
Socket('Ioctl',5,'FionRead')          == '0 8192'
Socket('Ioctl',5,'SiocAtMark')        == '0 No'
Socket('Ioctl',5,'SiocGifConf',2)     == '0 TR1 AF_INET 0 9.4.3.2 TR2
AF_INET 0 9.4.3.3'
Socket('Ioctl',5,'SiocGifAddr','TR1') == '0 TR1 AF_INET 0 9.4.3.2'
```

```
Example (socket descriptor 8 is AF_INET6)
Socket('Ioctl',8,'SiocGIFNameIndex') == '0 0 1e0 1 1e1'
```

C socket call: `ioctl(s, cmd, data)`

Select

Use this call to wait for socket-related events. Select returns all active socket IDs that have completed events when it is called. It does not check for order of completion.

A close on the other side of a socket connection is not reported as an exception, but as a read event that returns 0 bytes of data. When connect is called with a socket in nonblocking mode, the connect call ends and returns the code 36 (EINPROGRESS). The connection setup completion is then reported as a write event on the socket. When accept is called with a socket in nonblocking mode, the accept call ends and returns the code, 35 (EWOULDBLOCK). The availability of the connection request is reported as a Read event on the original socket, and accept should be called only after the read has been reported.

```
►►Socket—('Select', 'Read'—socketidlist—'Write'—socketidlist—►
►'Exception'—socketidlist—, —timeout.—)——►
```

Parameters

socketidlist

A list of socket descriptors.

timeout

A positive integer indicating the maximum wait time in seconds. The default is FOREVER.

Return Values

A string containing a return code, count, read socket ID list, write socket ID list, and exception socket ID list is returned.

Example

```
Socket('Select','Read 5 Write Exception',10) == '0 1 READ 5 WRITE EXCEPTION'
```

```
C socket call: select(nfds, readfds, writefds, exceptfds, timeout)
```

Setsockopt

Setsockopt sets the options associated with a socket. Setsockopt can be called only for sockets in the AF_INET or AF_INET6 domain.

The *optvalue* parameter is used to pass data used by the particular set command. The *optvalue* parameter points to a buffer containing the data needed by the set command. The *optvalue* parameter is optional and can be set to 0 if data is not needed by the command.

►►Socket—('Setsockopt',—socketid—,—level—,—optname—,—optvalue—)►►

Parameters

socketid

The socket descriptor for the socket setting options.

level

The protocol level for which the socket is being set. The levels are SOL_SOCKET, IPPROTO_TCP, IPPROTO_IP, and IPPROTO_IPV6.

- All commands beginning with SO_ are for protocol level SOL_SOCKET.
- All commands beginning with TCP_ are for protocol level IPPROTO_TCP.
- All commands beginning with IP_ are for protocol level IPPROTO_IP.
- All commands beginning with IPV6_ are for protocol level IPPROTO_IPV6.

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 803 for the numeric values of *optname*.

optvalue

For the SETSOCKOPT API, *optvalue* will be an input parameter. See the table below for a list of the options and their unique requirements.

Return Values

A string containing a return code is returned.

Example

Example (socket descriptor 5 is AF_INET)

```
Socket('SetSockOpt',5,'Sol_Socket','So_ASCII','0n')      == '0'
Socket('SetSockOpt',5,'Sol_Socket','So_Broadcast','0n')   == '0'
Socket('SetSockOpt',5,'Sol_Socket','So_Linger',60)        == '0'
```

Example (socket descriptor 8 is AF_INET6)

```
Socket('SetSockOpt',8,'IPPROTO_IPV6','IPV6_JOIN_GROUP','mreq') == '0'
Socket('SetSockOpt',8,'IPPROTO_IPV6','IPV6_LEAVE_GROUP','mreq') == '0'
Socket('SetSockOpt',8,'IPPROTO_IPV6','IPV6_MULTICAST_HOPS','14') == '0'
Socket('SetSockOpt',8,'IPPROTO_IPV6','IPV6_MULTICAST_IF','5') == '0'
Socket('SetSockOpt',8,'IPPROTO_IPV6','IPV6_MULTICAST_LOOP','1') == '0'
Socket('SetSockOpt',8,'IPPROTO_IPV6','IPV6_UNICAST_HOPS','8') == '0'
Socket('SetSockOpt',8,'IPPROTO_IPV6','IPV6_V6ONLY','1') == '0'
```

C socket call: setsockopt(s, level, optname, optval, optlen)

Note: OPTLEN is not a REXX parameter. All references to OPTLEN or the length of data in *optvall* in the table below should be ignored.

Table 23. 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 <i>hlq</i>.SEZAINST(CBLOCK) for the PL/I example of IP_MREQ.</p> <p>The IP_MREQ definition for COBOL:</p> <pre>01 IP-MREQ. 05 IMR-MULTIADDR PIC 9(8) BINARY. 05 IMR-INTERFACE PIC 9(8) BINARY.</pre>	<p>N/A</p>
<p>IP_DROP_MEMBERSHIP</p> <p>Use this option to enable an application to exit 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 <i>hlq</i>.SEZAINST(CBLOCK) for the PL/I example of IP_MREQ.</p> <p>The IP_MREQ definition for COBOL:</p> <pre>01 IP-MREQ. 05 IMR-MULTIADDR PIC 9(8) BINARY. 05 IMR-INTERFACE PIC 9(8) BINARY.</pre>	<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 are 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 23. *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>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 <i>hlq</i>.SEZAINST(CBLOCK) for the PL/I example of IPV6_MREQ.</p> <p>The IPV6_MREQ definition for COBOL:</p> <pre> 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.</pre>	<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 <i>hlq</i>.SEZAINST(CBLOCK) for the PL/I example of IPV6_MREQ.</p> <p>The IPV6_MREQ definition for COBOL:</p> <pre> 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.</pre>	<p>N/A</p>

Table 23. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
IPV6_MULTICAST_HOPS Use to set or obtain the hop limit used for outgoing multicast packets. This is an IPv6-only socket option.	Contains a 4-byte binary value specifying the multicast hops. If not specified, then the default is 1 hop. -1 indicates use stack default. 0 - 255 is the valid hop limit range. 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.	Contains a 4-byte binary value in the range from 0 to 255 indicating the number of multicast hops.
IPV6_MULTICAST_IF Use this option to set or obtain the index of the IPv6 interface used for sending outbound multicast datagrams from the socket application. This is an IPv6-only socket option.	Contains a 4-byte binary field containing an IPv6 interface index number.	Contains a 4-byte binary field containing an IPv6 interface index number.
IPV6_MULTICAST_LOOP 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. This is an IPv6-only socket option.	A 4-byte binary field. To enable, set to 1. To disable, set to 0.	A 4-byte binary field. If enabled, contains a 1. If disabled, contains a 0.
IPV6_UNICAST_HOPS Use this option to set or obtain the hop limit used for outgoing unicast IPv6 packets. This is an IPv6-only socket option.	Contains a 4-byte binary value specifying the unicast hops. If not specified, then the default is 1 hop. -1 indicates use stack default. 0 - 255 is the valid hop limit range. Note: APF authorized applications are permitted to set a hop limit that exceeds the system configured default. CICS applications cannot execute as APF authorized.	Contains a 4-byte binary value in the range from 0 to 255 indicating the number of unicast hops.
IPV6_V6ONLY 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. This is an IPv6-only socket option.	A 4-byte binary field. To enable, set to 1. To disable, set to 0.	A 4-byte binary field. If enabled, contains a 1. If disabled, contains a 0.

Table 23. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
SO_ASCII Use this option to set or determine the translation to ASCII data option. When <code>SO_ASCII</code> is set, data is translated to ASCII. When <code>SO_ASCII</code> is not set, data is not translated to or from ASCII. Note: This is a REXX-only socket option.	To enable, set to ON. To disable, set to OFF. 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.	If enabled, contains ON. If disabled, contains OFF. 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.
SO_BROADCAST 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. Note: This option has no meaning for stream sockets.	A 4-byte binary field. To enable, set to 1 or a positive value. To disable, set to 0.	A 4-byte field. If enabled, contains a 1. If disabled, contains a 0.
SO_DEBUG Use <code>SO_DEBUG</code> 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. Notes: <ol style="list-style-type: none"> 1. This is a REXX-only socket option. 2. This option has meaning only for stream sockets. 	To enable, set to ON. To disable, set to OFF.	If enabled, contains ON. If disabled, contains OFF.
SO_EBCDIC Use this option to set or determine the translation to EBCDIC data option. When <code>SO_EBCDIC</code> is set, data is translated to EBCDIC. When <code>SO_EBCDIC</code> is not set, data is not translated to or from EBCDIC. This option is ignored by EBCDIC hosts. Note: This is a REXX-only socket option.	To enable, set to ON. To disable, set to OFF. 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.	If enabled, contains ON. If disabled, contains OFF. 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.
SO_ERROR 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.	N/A	A 4-byte binary field containing the most recent <code>ERRNO</code> for the socket.

Table 23. *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 keepalive 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 keepalive 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 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 23. *OPTNAME* options for *GETSOCKOPT* and *SETSOCKOPT* (continued)

OPTNAME options (input)	SETSOCKOPT, OPTVAL (input)	GETSOCKOPT, OPTVAL (output)
<p>SO_OOINLINE</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 <i>RECV</i> or a <i>RECVFROM</i> even if the OOB flag is not set in the <i>RECV</i> or the <i>RECVFROM</i>.</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 <i>RECV</i> or a <i>RECVFROM</i> only when the OOB flag is set in the <i>RECV</i> or the <i>RECVFROM</i>.</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_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 <i>SETSOCKOPT</i> call:</p> <ul style="list-style-type: none"> • <i>TCPRCVBufsize</i> keyword on the <i>TCPCONFIG</i> statement in the <i>PROFILE.TCPIP</i> data set for a TCP Socket • <i>UDPRCVBufsize</i> keyword on the <i>UDPCONFIG</i> statement in the <i>PROFILE.TCPIP</i> data set for a UDP Socket • The default of 65 535 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 23. 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, refer to 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 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:</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 65 535 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>
<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>

Table 23. *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>

Version

This call returns the name REXX/SOCKETS, version number, and version date for the REXX Socket service.

►►—Socket—(—'Version'—)—————►◄

Parameters

None

Return Values

A string containing a return code and REXX/SOCKETS version data is returned.

Example

```
| Socket('Version')           == '0 REXX/SOCKETS z/OS V1R6 January 5, 2004'
```

REXX socket sample programs

This section provides sample REXX socket programs.

The following are the sample REXX socket programs available in the *hlq.SEZAINST* data set:

Program	Description
REXX-EXEC RSCLIENT	Client IPv4 sample program
REXX-EXEC RSSERVER	Server IPv4 sample program
REXX-EXEC R6CLIENT	Client IPv6 sample program
REXX-EXEC R6SERVER	Server IPv6 sample program

Before you start the client program, you must start the server program in another address space. The two programs can run on different hosts, but the internet address of the host running the server program must be entered with the command starting the client program, and the hosts must be connected on the same network using TCP/IP.

The REXX-EXEC RSCLIENT sample program for IPv4

The client sample program is a REXX socket program that shows you how to use the calls provided by REXX/SOCKETS. 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 obtains a socket set using the initialize call and a socket using the socket call. 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 and displays it on the screen until the data length is 0, indicating that the server has closed the connection. If an error occurs, the client program displays 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 client does not display partially received records.

```

/* REXX */
/*****
/*
/* Communications Server for OS/390, Version 2, Release 6
/*
/*
/* 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
/*
/*
*****/

/*****
/*- RSCLIENT -- Example demonstrating the usage of REXX/SOCKETS -----*/
*****/

trace o
signal on halt
signal on syntax

/* Set error code values
initialized = 0

parse arg argstring
argstring = strip(argstring)
if substr(argstring,1,1) = '?' then do
    say 'RSSERVER and RSCLIENT are a pair of programs which provide an'
    say 'example of how to use REXX/SOCKETS to implement a service. The'
    say 'server must be started before the clients get started.'
    say '
    say 'The RSSERVER program runs on a dedicated TSO USERID.
    say 'It returns a number of data lines as requested to the client.
    say 'It is started with the command:  RSSERVER
    say 'and terminated with the ATTN KEY and the immediate command HI.'
    say 'Alternate methods of running the RSSERVER are TSO/E Background'
    say '(IKJEFT01) or MVS Batch (IRXJCL).
    say '
    say 'The RSCLIENT program is used to request a number of arbitrary'
    say 'data lines from the server. One or more clients can access
    say 'the server until it is terminated.
    say 'It is started with the command:  RSCLIENT number <server>
    say 'where "number" is the number of data lines to be requested and
    say '"server" is the ipaddress of the service virtual machine. (The'

```

Figure 129. REXX-EXEC RSCLIENT sample program for IPv4 (Part 1 of 3)

```

    say 'default ipaddress is the one of the host  on which RSCLIENT is'
    say 'running, assuming that RSSERVER runs on the same host.)'
    say '
    exit 100
end

/* Split arguments into parameters and options */
parse upper var argstring parameters '(' options ')' rest

/* Parse the parameters */
parse var parameters lines server rest
if ^datatype(lines,'W') then call error 'E', 24, 'Invalid number'
lines = lines + 0
if rest^='' then call error 'E', 24, 'Invalid parameters'

/* Parse the options */
do forever
    parse var options token options
    select
        when token='' then leave
        otherwise call error 'E', 20, 'Invalid option "'token'"
    end
end

/* Initialize control information */
port = '1952' /* The port used by the server */

/* Initialize */
call Socket 'Initialize', 'RSCLIENT'
if src=0 then initialized = 1
else call error 'E', 200, 'Unable to initialize SOCKET'
if server='' then do
    server = Socket('GetHostId')
    if src^=0 then call error 'E', 200, 'Cannot get the local ipaddress'
end
ipaddress = server

/* Initialize for receiving lines sent by the server */
s = Socket('Socket')
if src^=0 then call error 'E', 32, 'SOCKET(SOCKET) rc='src
hostname = translate(Socket('GetHostName'))
if src^=0 then call error 'E', 32, 'SOCKET(GETHOSTNAME) rc='src
call Socket 'Connect', s, 'AF_INET' port ipaddress
if src^=0 then call error 'E', 32, 'SOCKET(CONNECT) rc='src
call Socket 'Write', s, hostname userid() lines
if src^=0 then call error 'E', 32, 'SOCKET(WRITE) rc='src

/* Wait for lines sent by the server */
dataline = ''
num = 0
do forever

    /* Receive a line and display it */
    parse value Socket('Read', s) with len newline
    if src^=0 | len<=0'' then leave

```

Figure 129. REXX-EXEC RSCLIENT sample program for IPv4 (Part 2 of 3)

```

    dataline = dataline || newline
  do forever
    if pos('15'x,dataline)=0 then leave
    parse var dataline nextline '15'x dataline
    num = num + 1
    say right(num,5)':' nextline
  end
end

/* Terminate and exit */
call Socket 'Terminate'
exit 0

/* Calling the real SOCKET function */
socket: procedure expose src
  a0 = arg(1)
  a1 = arg(2)
  a2 = arg(3)
  a3 = arg(4)
  a4 = arg(5)
  a5 = arg(6)
  a6 = arg(7)
  a7 = arg(8)
  a8 = arg(9)
  a9 = arg(10)
  parse value 'SOCKET'(a0,a1,a2,a3,a4,a5,a6,a7,a8,a9) with src res
  return res

/* Syntax error routine */
syntax:
  call error 'E', rc, '==> REXX Error No.' 20000+rc
  return

/* Halt processing routine */
halt:
  call error 'E', 4, '==> REXX Interrupted'
  return

/* Error message and exit routine */
error:
  type = arg(1)
  retc = arg(2)
  text = arg(3)
  ecretc = right(retc,3,'0')
  ectype = translate(type)
  ecfull = 'RXSCLI' || ecretc || ectype
  say '==> Error:' ecfull text
  if type^='E' then return
  if initialized
    then do
      parse value Socket('SocketSetStatus') with . status severreason
      if status^='Connected'
        then say 'The status of the socket set is' status severreason
    end
  call Socket 'Terminate'
  exit retc

```

Figure 129. REXX-EXEC RSCLIENT sample program for IPv4 (Part 3 of 3)

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 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 reported by the select call. 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 only occur 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 events from a single socket.


```

/* REXX */
/*****
/*
/* Communications Server for OS/390, Version 2, Release 6
/*
/*
/* 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
/*
*****/

/*****
/*- RSSERVER -- Example demonstrating the usage of REXX/SOCKETS -----*/
*****/

trace o
signal on syntax
signal on halt

/* Set error code values
initialized = 0

parse arg argstring
argstring = strip(argstring)
if substr(argstring,1,1) = '?' then do
    say 'RSSERVER and RSCLIENT are a pair of programs which provide an'
    say 'example of how to use REXX/SOCKETS to implement a service. The'
    say 'server must be started before the clients get started.'
    say '
    say 'The RSSERVER program runs on a dedicated TSO USERID.
    say 'It returns a number of data lines as requested to the client.
    say 'It is started with the command: RSSERVER
    say 'and terminated with the ATTN key and the immediate command HI.'
    say 'Alternate methods of running the RSSERVER are TSO/E Background'
    say '(IKJEFT01) or MVS Batch (IRXJCL).
    say '
    say 'The RSCLIENT program is used to request a number of arbitrary'
    say 'data lines from the server. One or more clients can access
    say 'the server until it is terminated.
    say 'It is started with the command: RSCLIENT number <server>
    say 'where "number" is the number of data lines to be requested and
    say '"server" is the ipaddress of the service virtual machine. (The'
    say 'default ipaddress is the one of the host on which RSCLIENT is'

```

Figure 130. REXX-EXEC RSSERVER sample program for IPv4 (Part 1 of 5)

```

    say 'running, assuming that RSSERVER runs on the same host.)'
    say '
exit 100
end

/* Split arguments into parameters and options */
parse upper var argstring parameters '(' options ')' rest

/* Parse the parameters */
parse var parameters rest
if rest^='' then call error 'E', 24, 'Invalid parameters specified'

/* Parse the options */
do forever
    parse var options token options
    select
        when token='' then leave
        otherwise call error 'E', 20, 'Invalid option "'token'"
    end
end

/* Initialize control information */
port = '1952' /* The port used for the service */

/* Initialize */
say 'RSSERVER: Initializing'
call Socket 'Initialize', 'RSSERVER'
if src=0 then initialized = 1
else call error 'E', 200, 'Unable to initialize SOCKET'
ipaddress = Socket('GetHostId')
if src^=0 then call error 'E', 200, 'Unable to get the local ipaddress'
say 'RSSERVER: Initialized: ipaddress='ipaddress 'port='port

/* Initialize for accepting connection requests */
s = Socket('Socket')
if src^=0 then call error 'E', 32, 'SOCKET(SOCKET) rc='src
call Socket 'Bind', s, 'AF_INET' port ipaddress
if src^=0 then call error 'E', 32, 'SOCKET(BIND) rc='src
call Socket 'Listen', s, 10
if src^=0 then call error 'E', 32, 'SOCKET(LISTEN) rc='src
call Socket 'Ioctl', s, 'FIONBIO', 'ON'
if src^=0 then call error 'E', 36, 'Cannot set mode of socket' s

/* Wait for new connections and send lines */
timeout = 60
linecount. = 0
wlist = ''
do forever

    /* Wait for an event */
    if wlist^='' then sockevtlist = 'Write'wlist 'Read * Exception'
    else sockevtlist = 'Write Read * Exception'
    sellist = Socket('Select',sockevtlist,timeout)
    if src^=0 then call error 'E', 36, 'SOCKET(SELECT) rc='src
    parse upper var sellist . 'READ' orlist 'WRITE' owlist 'EXCEPTION' .

```

Figure 130. REXX-EXEC RSSERVER sample program for IPv4 (Part 2 of 5)

```

if orlist^='' | owlist^='' then do
    event = 'SOCKET'
    if orlist^='' then do
        parse var orlist orsocket .
        rest = 'READ' orsocket
    end
    else do
        parse var owlist owsocket .
        rest = 'WRITE' owsocket
    end
end
else event = 'TIME'

select

/* Accept connections from clients, receive and send messages */
when event='SOCKET' then do
    parse var rest keyword ts .

    /* Accept new connections from clients */
    if keyword='READ' & ts=s then do
        nsn = Socket('Accept',s)
        if src=0 then do
            parse var nsn ns . np nia .
            say 'RSSERVER: Connected by' nia 'on port' np 'and socket' ns
        end
    end

    /* Get nodeid, userid and number of lines to be sent */
    if keyword='READ' & ts^=s then do
        parse value Socket('Recv',ts) with len nid uid count .
        if src=0 & len>0 & datatype(count,'W') then do
            if count<0 then count = 0
            if count>5000 then count = 5000
            ra = 'by' uid 'at' nid
            say 'RSSERVER: Request for' count 'lines on socket' ts ra
            linecount.ts = linecount.ts + count
            call addsock(ts)
        end
        else do
            call Socket 'Close',ts
            linecount.ts = 0
            call delsock(ts)
            say 'RSSERVER: Disconnected socket' ts
        end
    end

    /* Get nodeid, userid and number of lines to be sent */
    if keyword='WRITE' then do
        if linecount.ts>0 then do
            num = random(1,sourceline()) /* Return random-selected */
            msg = sourceline(num) || '15'x /* line of this program */
            call Socket 'Send',ts,msg
            if src=0 then linecount.ts = linecount.ts - 1
            else linecount.ts = 0
        end
    end
end

```

Figure 130. REXX-EXEC RSSERVER sample program for IPv4 (Part 3 of 5)

```

        end
        else do
            call Socket 'Close',ts
            linecount.ts = 0
            call delsock(ts)
            say 'RSSERVER: Disconnected socket' ts
        end
    end

end

/* Unknown event (should not occur) */
otherwise nop
end
end

/* Terminate and exit */
call Socket 'Terminate'
say 'RSSERVER: Terminated'
exit 0

/* Procedure to add a socket to the write socket list */
addsock: procedure expose wlist
    s = arg(1)
    p = wordpos(s,wlist)
    if p=0 then wlist = wlist s
return

/* Procedure to del a socket from the write socket list */
delsock: procedure expose wlist
    s = arg(1)
    p = wordpos(s,wlist)
    if p>0 then do
        templist = ''
        do i=1 to words(wlist)
            if i^=p then templist = templist word(wlist,i)
        end
        wlist = templist
    end
return

/* Calling the real SOCKET function */
socket: procedure expose initialized src
    a0 = arg(1)
    a1 = arg(2)
    a2 = arg(3)
    a3 = arg(4)
    a4 = arg(5)
    a5 = arg(6)
    a6 = arg(7)
    a7 = arg(8)
    a8 = arg(9)
    a9 = arg(10)
    parse value 'SOCKET'(a0,a1,a2,a3,a4,a5,a6,a7,a8,a9) with src res
return res

```

Figure 130. REXX-EXEC RSSERVER sample program for IPv4 (Part 4 of 5)

```

/* Syntax error routine */
syntax:
    call error 'E', rc, '==> REXX Error No.' 20000+rc
return

/* Halt exit routine */
halt:
    call error 'E', 4, '==> REXX Interrupted'
return

/* Error message and exit routine */
error:
    type = arg(1)
    retc = arg(2)
    text = arg(3)
    ecretc = right(retc,3,'0')
    ectype = translate(type)
    ecfull = 'RXSSRV' || ecretc || ectype
    say '==> Error:' ecfull text
    if type^='E' then return
    if initialized
    then do
        parse value Socket('SocketSetStatus') with . status severreason
        if status^='Connected'
        then say 'The status of the socket set is' status severreason
    end
    call Socket 'Terminate'
exit retc

```

Figure 130. REXX-EXEC RSSERVER sample program for IPv4 (Part 5 of 5)

The REXX-EXEC R6CLIENT sample program for IPv6

The client sample program is a REXX socket program that shows you how to use the calls provided by REXX/SOCKETS. 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 obtains a socket set using the initialize call and a socket using the socket call. 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 and displays it on the screen until the data length is 0, indicating that the server has closed the connection. If an error occurs, the client program displays 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 client does not display partially received records.

```

/* REXX */
/*****
/*
/* Communications Server for z/OS, Version 1, Release 4
/*
/*
/* 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
/*
/* CHANGE-ACTIVITY =
/* CFD LIST:
/*
/* $N0=D314.13 CSV1R4 010831 KMP: IPV6 CLIENT EXAMPLE
/* $N1=MV24392 D314.13 011004 KMP: Added a getaddrinfo to connect
/*              with.
/*
/* END CFD LIST:
/*
*****/

/*****
/*- R6CLIENT -- Example demonstrating the usage of REXX/SOCKETS -----*/
*****/

trace o
signal on halt
signal on syntax

/* Set error code values
initialized = 0

parse arg argstring
argstring = strip(argstring)
if substr(argstring,1,1) = '?' then do
    say 'R6SERVER and R6CLIENT are a pair of programs which provide an'
    say 'example of how to use REXX/SOCKETS to implement a service. The'
    say 'server must be started before the clients get started.'
    say '
    say 'The R6SERVER program runs on a dedicated TSO USERID.
    say 'It returns a number of data lines as requested to the client.'
    say 'It is started with the command: R6SERVER
    say 'and terminated with the ATTN KEY and the immediate command HI.'
    say 'Alternate methods of running the R6SERVER are TSO/E Background'

```

Figure 131. REXX-EXEC R6CLIENT sample program for IPv6 (Part 1 of 4)

```

say '(IKJEFT01) or MVS Batch (IRXJCL).
say '
say 'The R6CLIENT program is used to request a number of arbitrary'
say 'data lines from the server. One or more clients can access '
say 'the server until it is terminated.
say 'It is started with the command: R6CLIENT number <server>
say 'where "number" is the number of data lines to be requested and'
say '"server" is the ipaddress of the service virtual machine. (The'
say 'default ipaddress is the one of the host on which R6CLIENT is'
say 'running, assuming that R6SERVER runs on the same host.)
say '
exit 100
end

/* Split arguments into parameters and options */
parse upper var argstring parameters '(' options ')' rest

/* Parse the parameters */
parse var parameters lines server rest
if ^datatype(lines,'W') then call error 'E', 24, 'Invalid number'
lines = lines + 0
if rest^='' then call error 'E', 24, 'Invalid parameters'

/* Parse the options */
do forever
  parse var options token options
  select
    when token='' then leave
    otherwise call error 'E', 20, 'Invalid option "'token'"
  end
end

/* Initialize control information */
family='AF_INET6'
port = '1952' /* The port used by the server */
server='::1'
flowinfo='0'
scopeid='0'

/* Initialize */
call Socket 'Initialize', 'R6CLIENT'
if src=0 then initialized = 1
else call error 'E', 200, 'Unable to initialize SOCKET'
if server='' then do
  server = Socket('GetHostId')
  if src^=0 then call error 'E', 200, 'Cannot get the local ipaddress'
end
ipaddress = server

/* Initialize for receiving lines sent by the server */
domain='AF_INET6'
s = Socket('Socket', domain)
if src^=0 then call error 'E', 32, 'SOCKET(SOCKET) rc='src
/* hostname = translate(Socket('GetHostName')) */
node = Socket('GetHostname')

```

Figure 131. REXX-EXEC R6CLIENT sample program for IPv6 (Part 2 of 4)

```

if src^=0 then call error 'E', 32, 'SOCKET(GETHOSTNAME) rc='src

/* Getaddrinfo                                                                    */
service=''
flags='ai_canonnameok'
family='AF_INET6'
socktype='SOCK_STREAM'
protocol=''
res=Socket('GetAddrinfo', node, service, flags, family, socktype, protocol)
if src^=0 then call error 'E', 200, 'Unable to get the local ipaddress'
if src = 0 then do
    parse var res . domain .
    say '          domain='domain
    if domain='AF_INET' then do
        parse var res . . . ipaddress
        say 'R6CLIENT: Initialized: ipaddress='ipaddress 'port='port
    end
    else do
        if domain='AF_INET6' then do
            parse var res . . . . ipaddress .
            say 'R6CLIENT: Initialized: ipaddress='ipaddress 'port='port
        end
    end
end
end

call Socket 'Connect', s, family port flowinfo ipaddress scopeid
if src^=0 then call error 'E', 32, 'SOCKET(CONNECT) rc='src
call Socket 'Write', s, hostname userid() lines
if src^=0 then call error 'E', 32, 'SOCKET(WRITE) rc='src

/* Wait for lines sent by the server                                              */
dataline = ''
num = 0
do forever

    /* Receive a line and display it                                             */
    parse value Socket('Read', s) with len newline
    if src^=0 | len<=0'' then leave
    dataline = dataline || newline
    do forever
        if pos('15'x,dataline)=0 then leave
        parse var dataline nextline '15'x dataline
        num = num + 1
        say right(num,5)':' nextline
    end
end

/* Terminate and exit                                                            */
call Socket('Terminate')
exit 0

/* Calling the real SOCKET function                                              */
socket: procedure expose src
    a0 = arg(1)
    a1 = arg(2)

```

Figure 131. REXX-EXEC R6CLIENT sample program for IPv6 (Part 3 of 4)


```

a2 = arg(3)
a3 = arg(4)
a4 = arg(5)
a5 = arg(6)
a6 = arg(7)
a7 = arg(8)
a8 = arg(9)
a9 = arg(10)
parse value 'SOCKET'(a0,a1,a2,a3,a4,a5,a6,a7,a8,a9) with src res
return res

/* Syntax error routine */
syntax:
  call error 'E', rc, '==> REXX Error No.' 20000+rc
return

/* Halt processing routine */
halt:
  call error 'E', 4, '==> REXX Interrupted'
return

/* Error message and exit routine */
error:
  type = arg(1)
  retc = arg(2)
  text = arg(3)
  ecretc = right(retc,3,'0')
  ectype = translate(type)
  ecfull = 'RXSCLI' || ecretc || ectype
  say '==> Error:' ecfull text
  if type^='E' then return
  if initialized
    then do
      parse value Socket('SocketSetStatus') with . status severreason
      if status^='Connected'
        then say 'The status of the socket set is' status severreason
    end
  say 'Performing Terminate from error:'
  call Socket('Terminate')
exit retc

```

Figure 131. REXX-EXEC R6CLIENT sample program for IPv6 (Part 4 of 4)

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 reported by the select call. 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 only occur 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 events from a single socket.

```

/* REXX */
/*****
/*
/* Communications Server for z/OS, Version 1, Release 4
/*
/*
/* 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
/*
/* CHANGE-ACTIVITY =
/* CFD List:
/*
/* $N0=D314.13 CSV1R4 010831 KMP: IPv6 Server example
/* $N1=MV24392 D314.13 011004 KMP: Fixed getaddrinfo error message.
/*
/* End CFD List:
*****/

/*****
/*- R6SERVER -- Example demonstrating the usage of REXX/SOCKETS -----*/
*****/

trace o
signal on syntax
signal on halt

/* Set error code values
initialized = 0

parse arg argstring
argstring = strip(argstring)
if substr(argstring,1,1) = '?' then do
  say 'R6SERVER and R6CLIENT are a pair of programs which provide an'
  say 'example of how to use REXX/SOCKETS to implement a service. The'
  say 'server must be started before the clients get started.'
  say '
  say 'The R6SERVER program runs on a dedicated TSO USERID.'
  say 'It returns a number of data lines as requested to the client.'
  say 'It is started with the command: R6SERVER'
  say 'and terminated with the ATTN key and the immediate command HI.'
  say 'Alternate methods of running the R6SERVER are TSO/E Background'
  say '(IKJEFT01) or MVS Batch (IRXJCL).'
  say '

```

Figure 132. REXX-EXEC R6SERVER sample program for IPv6 (Part 1 of 6)

```

say 'The R6CLIENT program is used to request a number of arbitrary'
say 'data lines from the server. One or more clients can access '
say 'the server until it is terminated.'
say 'It is started with the command: R6CLIENT number <server>'
say 'where "number" is the number of data lines to be requested and'
say '"server" is the ipaddress of the service virtual machine. (The'
say 'default ipaddress is the one of the host on which R6CLIENT is'
say 'running, assuming that R6SERVER runs on the same host.)'
say '
exit 100
end

/* Split arguments into parameters and options */
parse upper var argstring parameters '(' options ')' rest

/* Parse the parameters */
parse var parameters rest
if rest^='' then call error 'E', 24, 'Invalid parameters specified'

/* Parse the options */
do forever
  parse var options token options
  select
    when token='' then leave
    otherwise call error 'E', 20, 'Invalid option "'token'"'
  end
end

/* Initialize control information */
port = '1952' /* The port used for the service */
ipaddress = '::1'
flowinfo = '0'
scopeid = '0'

/* Initialize */
say 'R6SERVER: Initializing'
call Socket 'Initialize', 'R6SERVER'
if src=0 then initialized = 1
else call error 'E', 200, 'Unable to initialize SOCKET'
node = Socket('GetHostname')
if src^=0 then call error 'E', 200, 'Unable to get the local host name'

/* Getaddrinfo */
service=''
flags='ai_canonnameok'
family='AF_INET6'
socktype='SOCK_STREAM'
protocol=''
res=Socket('GetAddrinfo', node, service, flags, family, socktype, protocol)
if src^=0 then call error 'E', 200, 'Unable to get the local ipaddress'
if src = 0 then do
  parse var res . domain .
  say ' domain='domain
  if domain='AF_INET' then do
    parse var res . . . ipaddress
  end
end

```

Figure 132. REXX-EXEC R6SERVER sample program for IPv6 (Part 2 of 6)

```

        say 'R6SERVER: Initialized: ipaddress='ipaddress 'port='port
    end
else do
    if domain='AF_INET6' then do
        parse var res . . . . ipaddress .
        say 'R6SERVER: Initialized: ipaddress='ipaddress 'port='port
    end
end
end
end

/* Initialize for accepting connection requests */
s = Socket('Socket', AF_INET6)
if src^=0 then call error 'E', 32, 'SOCKET(SOCKET) rc='src
call Socket 'Bind', s, AF_INET6 port flowinfo ipaddress scopeid
if src^=0 then call error 'E', 32, 'SOCKET(BIND) rc='src
call Socket 'Listen', s, 10
if src^=0 then call error 'E', 32, 'SOCKET(LISTEN) rc='src
call Socket 'Ioctl', s, 'FIONBIO', 'ON'
if src^=0 then call error 'E', 36, 'Cannot set mode of socket' s

/* Wait for new connections and send lines */
timeout = 60
linecount. = 0
wlist = ''
do forever

    /* Wait for an event */
    if wlist^='' then sockevtlist = 'Write'wlist 'Read * Exception'
    else sockevtlist = 'Write Read * Exception'
    sellist = Socket('Select',sockevtlist,timeout)
    if src^=0 then call error 'E', 36, 'SOCKET(SELECT) rc='src
    parse upper var sellist . 'READ' orlist 'WRITE' owlist 'EXCEPTION' .
    if orlist^='' | owlist^='' then do
        event = 'SOCKET'
        if orlist^='' then do
            parse var orlist orsocket .
            rest = 'READ' orsocket
        end
        else do
            parse var owlist owsocket .
            rest = 'WRITE' owsocket
        end
    end
    end
    else event = 'TIME'

select

    /* Accept connections from clients, receive and send messages */
    when event='SOCKET' then do
        parse var rest keyword ts .

        /* Accept new connections from clients */
        if keyword='READ' & ts=s then do
            nsn = Socket('Accept',s)
            if src=0 then do

```

Figure 132. REXX-EXEC R6SERVER sample program for IPv6 (Part 3 of 6)

```

    parse var nsn ns . np . nia .
    say 'R6SERVER: Connected by' nia 'on port' np 'and socket' ns
    parse var nsn . nname

    /* GetNameInfo */
    flags=''
    host_service = Socket('Getnameinfo', nname, flags)
    if src = 0 then do
        parse var host_service host service
        say 'R6SERVER: as host 'host ' by the way of service ' service
    end
end

/* Get nodeid, userid and number of lines to be sent */
if keyword='READ' & ts^=s then do
    parse value Socket('Recv',ts) with len nid uid count .
    if src=0 & len>0 & datatype(count,'W') then do
        if count<0 then count = 0
        if count>5000 then count = 5000
        ra = 'by' uid 'at' nid
        say 'R6SERVER: Request for' count 'lines on socket' ts ra
        linecount.ts = linecount.ts + count
        call addsock(ts)
    end
    else do
        call Socket 'Close',ts
        linecount.ts = 0
        call delsock(ts)
        say 'R6SERVER: Disconnected socket' ts
    end
end

/* Get nodeid, userid and number of lines to be sent */
if keyword='WRITE' then do
    if linecount.ts>0 then do
        num = random(1,sourceline()) /* Return random-selected */
        msg = sourceline(num) || '15'x /* line of this program */
        call Socket 'Send',ts,msg
        if src=0 then linecount.ts = linecount.ts - 1
        else linecount.ts = 0
    end
    else do
        call Socket 'Close',ts
        linecount.ts = 0
        call delsock(ts)
        say 'R6SERVER: Disconnected socket' ts
    end
end

/* Unknown event (should not occur) */
otherwise nop
end

```

Figure 132. REXX-EXEC R6SERVER sample program for IPv6 (Part 4 of 6)

```

end

/* Terminate and exit */
call Socket 'Terminate'
say 'R6SERVER: Terminated'
exit 0

/* Procedure to add a socket to the write socket list */
addsock: procedure expose wlist
  s = arg(1)
  p = wordpos(s,wlist)
  if p=0 then wlist = wlist s
return

/* Procedure to del a socket from the write socket list */
delsock: procedure expose wlist
  s = arg(1)
  p = wordpos(s,wlist)
  if p>0 then do
    templist = ''
    do i=1 to words(wlist)
      if i^=p then templist = templist word(wlist,i)
    end
    wlist = templist
  end
return

/* Calling the real SOCKET function */
socket: procedure expose initialized src
  a0 = arg(1)
  a1 = arg(2)
  a2 = arg(3)
  a3 = arg(4)
  a4 = arg(5)
  a5 = arg(6)
  a6 = arg(7)
  a7 = arg(8)
  a8 = arg(9)
  a9 = arg(10)
  parse value 'SOCKET'(a0,a1,a2,a3,a4,a5,a6,a7,a8,a9) with src res
return res

/* Syntax error routine */
syntax:
  call error 'E', rc, '==> REXX Error No.' 20000+rc
return

/* Halt exit routine */
halt:
  call error 'E', 4, '==> REXX Interrupted'
return

/* Error message and exit routine */
error:

```

Figure 132. REXX-EXEC R6SERVER sample program for IPv6 (Part 5 of 6)

```

type = arg(1)
retc = arg(2)
text = arg(3)
ecretc = right(retc,3,'0')
ectype = translate(type)
ecfull = 'RXSSRV' || ecretc || ectype
say '==> Error:' ecfull text
if type^='E' then return
if initialized
  then do
    parse value Socket('SocketSetStatus') with . status severreason
    if status^='Connected'
      then say 'The status of the socket set is' status severreason
  end
call Socket 'Terminate'
exit retc

```

Figure 132. REXX-EXEC R6SERVER sample program for IPv6 (Part 6 of 6)

Chapter 15. Pascal application programming interface (API)

This chapter describes the Pascal language for IPv4 socket application program interface (API) 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 will not be enhanced for IPv6 support.

Steps for procedure calls

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.

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 will block until buffer space becomes available or an error is detected. This is the only condition under which TcpFSend will block.
-
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 using TcpClose, you must wait for the connection to terminate.
-
8. Terminate the communication service (EndTcpIp).
-

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.

A sample program is supplied with TCP/IP. See "Sample Pascal program" on page 754, for a listing of the sample program.

Software requirements

To develop programs in Pascal that interface directly to the TCP, UDP, and IP protocol boundaries, you must have the IBM VS Pascal Compiler and Library (5668-767).

Pascal API header files

The following is a list of the headers used by Pascal applications:

- cmclien
- cmcomm
- cminter
- cmresglb

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.

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 *hlq.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 *hlq.SEZACMTX* before the Pascal libraries when linking TCP/IP programs.

Connection state

ConnectionState is the current state of the connection. See Figure 133 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 24 on page 714 for the mapping between TCP states and ConnectionStateType.

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

Figure 133. 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 24 lists the TCP connection states.

Table 24. 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 134.

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

Figure 134. 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 135 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 135 shows the Pascal declaration of a socket type.

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

Figure 135. 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 136 on page 716 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
        );
    end case;
  end record

```

Figure 136. Notification record (Part 1 of 2)

```

FSENDresponse:
(
  SendTurnCode: CallReturnCodeType;
  SendRequestErr: Boolean;
);
PINGresponse:
(
  PingTurnCode: CallReturnCodeType;
  ElapsedTime: TimeStampType
);
RAWIPpacketsDELIVERED:
(
  RawIpDataLength: integer;
  RawIpFullLength: integer;
);
RAWIPspaceAVAILABLE:
(
  RawIpSpaceInBytes: integer;
);
MSGreceived: ();
TIMERexpired:
(
  Datum: integer;
  AssociatedTimer: TimerPointerType
);
UDPdatagramDELIVERED:
(
  DataLength: integer;
  ForeignSocket: SocketType;
  FullLength: integer
);
UDPdatagramSPACEavailable: ();
URGENTpending:
(
  BytesToRead: integer;
  UrgentSpan: integer
);
USERdefinedNOTIFICATION:
(
  UserData: UserNotificationDataType
);
end;

```

Figure 136. Notification record (Part 2 of 2)

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 is 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.

SMSGreceived

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 137 on page 721.

```

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;
  end;

```

Figure 137. Pascal declaration of file specification record

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 25 on page 723 for an explanation of the return codes.

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

Notifications

The 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 `BeginTcpIp`.

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 136 on page 716 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 `hlq.SEZACMAC` data set. The host lookup routine `GetHostResol` is declared in the `CMRESGLB` member of the `hlq.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 737) 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 25 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 25. Pascal language return codes

Return Value	Return Code	Message Text
OK	0	OK.
ABNORMALcondition	–1	Abnormal condition due to CSA storage shortage.
ALREADYclosing	–2	Connection is already closing.
BADlengthARGUMENT	–3	Length specified that is not valid.
CANNOTsendDATA	–4	Cannot send data.
CLIENTrestart	–5	Client reinitialized TCP/IP service.
CONNECTIONalreadyEXISTS	–7	Connection already 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.
NOutstandingNOTIFICATIONS	–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.

Table 25. Pascal language return codes (continued)

Return Value	Return Code	Message Text
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.
TIMEOUTopen	-40	Foreign host did not respond within OPEN timeout.
TOOmanyOPENS	-41	Too many open connections exist already.
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.

Procedure calls

This section 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 138 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 138. 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 139.

```

procedure BeginTcpIp
(
    var ReturnCode: integer
);
external;

```

Figure 139. 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 25 on page 723.

ClearTimer

This procedure resets the timer to prevent it timing out as shown in Figure 140.

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

Figure 140. 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 740 to activate the timer. Figure 141 shows an example.

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

Figure 141. 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 142 shows an example.

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

Figure 142. 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 143 on page 727 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 143. EndTcpIp example

GetHostNumber

The GetHostNumber procedure resolves a host name into an internet address. This is shown in Figure 144.

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 144. 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 145 on page 728 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: InetAddressType
);
external;

```

Figure 145. *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 146.

```

procedure GetHostString
(
  Address: InetAddressType;
  var     Name: SiteNameType
);
external;

```

Figure 146. *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`. Refer to *z/OS Communications Server: IP Configuration Reference* for information about `TCPIP.DATA` search order.

Figure 147 shows the GetIdentity procedure.

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

Figure 147. 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 148 shows an example of the GetNextNote procedure.

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

Figure 148. 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: <ul style="list-style-type: none">• OK• NOoutstandingNOTIFICATIONS• NOTyetBEGUN

For a description of Pascal return codes, see Table 25 on page 723.

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 149 shows an example of the GetSmsg procedure.

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

Figure 149. 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 150. 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 136 on page 716.

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

Figure 150. 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: <ul style="list-style-type: none">• OK• NOTyetBEGUN

For a description of Pascal return codes, see Table 25 on page 723.

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 151 on page 731 shows an example.

```

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

```

Figure 151. 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: <ul style="list-style-type: none"> • OK • NONlocalADDRESS • TCPIPshutdown • FATALerror • SOFTWAREerror

For a description of Pascal return codes, see Table 25 on page 723.

IsLocalHost

This procedure returns the correct host class for Name, which can be a host name or a dotted decimal address. Figure 152 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 152. 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.

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

Figure 153. 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: <ul style="list-style-type: none">• 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 154.

```
MonQueryRecordType =
record
case QueryType: MonQueryType of
    QUERYhomeONLY: ();
end; { MonQueryRecordType }
```

Figure 154. 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 25 on page 723.

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;
var
    ReturnCode: integer
);
external;
```

Figure 155. 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: <ul style="list-style-type: none">• OK• ABNORMALcondition• BADlengthARGUMENT• CONNECTIONalreadyEXISTS• VIRTUALmemoryTOOsmall• NOTyetBEGUN• TIMEOUTopen• PARAMtimeout• SOFTWAREerror• TCPipSHUTDOWN• UNAUTHORIZEDuser

For a description of Pascal return codes, see Table 25 on page 723.

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 156 on page 734 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 156. RawIpClose example

Parameter	Description
ProtocolNo	The number of the Internet protocol.
ReturnCode	Indicates the success or failure of a call. Possible return values are: <ul style="list-style-type: none"> • OK • NOsuchCONNECTION • NOTyetBEGUN • SOFTWAREerror • TCPIPshutdown • UNAUTHORIZEDuser

For a description of Pascal return codes, see Table 25 on page 723.

RawIpOpen

This procedure tells the TCP/IP address space that the client wants to send and receive packets of the specified protocol. Figure 157 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 157. RawIpOpen example

Parameter	Description
ProtocolNo	The number of the Internet protocol.
ReturnCode	Indicates success or failure of a call. Possible return values are: <ul style="list-style-type: none"> • OK • LOCALportNOTavailable • NObufferSPACE • NOTyetBEGUN • SOFTWAREerror • TCPIPshutdown • UNAUTHORIZEDuser

For a description of Pascal return codes, see Table 25 on page 723.

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 158 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 158. 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: <ul style="list-style-type: none">• OK• NOsuchCONNECTION• NOTyetBEGUN• SOFTWAREerror• TCPIPshutdown• UNAUTHORIZEDuser• INVALIDvirtualADDRESS

For a description of Pascal return codes, see Table 25 on page 723.

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 159. 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: <ul style="list-style-type: none"> • 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 25 on page 723.

ReadXlateTable

The procedure shown in Figure 160 on page 737 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 160. ReadXlateTable example

Parameter	Description
TableName	The name of the translate table. ReadXlateTable tries to read <i>user_id</i> .TableName.TCPXLBIN. If that data set exists but it has an incorrect format, ReadXlateTable returns with a ReturnCode FILEformatINVALID. If <i>user_id</i> .TableName.TCPXLBIN does not exist, ReadXlateTable tries to read <i>hlq</i> .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: <ul style="list-style-type: none"> • OK • ERRORopeningORreadingFILE • FILEformatINVALID

SayCalRe

This function returns a printable string describing the return code passed in CallReturn. Figure 161 shows an example.

```

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

```

Figure 161. 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 162 on page 738 shows an example of this procedure.

```

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

```

Figure 162. SayConSt example

Parameter	Description
State	The connection state to be described.

SayIntAd

This function converts the Internet 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. Refer to *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 163 shows an example of this procedure.

```

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

```

Figure 163. 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 164.

```

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

```

Figure 164. 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 165. 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 166 shows an example of this function.

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

Figure 166. 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 167 shows an example of this function.

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

Figure 167. SayProTy example

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

SetTimer

The procedure shown in Figure 168 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 168. *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 169 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 169. *TcpAbort* example

Parameter	Description
Connection	The connection number, as returned by <code>TcpOpen</code> or <code>TcpWaitOpen</code> in the <code>Connection</code> field of the <code>StatusInfoType</code> record.
ReturnCode	Indicates success or failure of call. Possible return values are: <ul style="list-style-type: none">• 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 25 on page 723.

TcpClose

Use the procedure shown in Figure 170 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 170. *TcpClose* example

Parameter	Description
Connection	The connection number, as returned by <code>TcpOpen</code> or <code>TcpWaitOpen</code> in the <code>Connection</code> field of the <code>StatusInfoType</code> record.
ReturnCode	Indicates success or failure of call. Possible return values are: <ul style="list-style-type: none">• <code>OK</code>• <code>ABNORMALcondition</code>• <code>ALREADYclosing</code>• <code>NOsuchCONNECTION</code>• <code>NOTyetBEGUN</code>• <code>TCPIPshutdown</code>• <code>SOFTWAREerror</code>• <code>REMOTEReset</code>

For a description of Pascal return codes, see Table 25 on page 723.

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 section 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. A return code of `OK` means that the request has been accepted. When received data has been placed in your buffer, your program receives a `DATAdelivered` notification.

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 171. *TcpFReceive example*

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

Figure 172. *TcpReceive example*

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

Figure 173. *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 25 on page 723.

Notes (TcpWaitReceive):

1. For BytesRead OK, the function was initiated, but the connection is no longer receiving for an unspecified reason. Your program does not have to issue TcpClose, but the connection is not completely terminated until a NONEXISTENT notification is received for the connection.
2. For BytesRead REMOTEclose, the foreign host has closed the connection. Your program should respond with TcpClose.
3. If you receive any of the codes marked with (+), the function was initiated but the connection has now been terminated (see 2 on page 718). 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 section 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 174. TcpFSend example

```

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

```

Figure 175. TcpSend example

```

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

```

Figure 176. TcpWaitSend example

Parameter	Description
-----------	-------------

Connection	The connection number, as returned by <code>TcpOpen</code> or <code>TcpWaitOpen</code> in the <code>Connection</code> field of the <code>StatusInfoType</code> 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 <i>urgent</i> . 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: <ul style="list-style-type: none"> • OK • ABNORMALcondition • BADlengthARGUMENT • CANNOTsendDATA • FATALerror • NObufferSPACE (<code>TcpSend</code> and <code>TcpFSend</code>) • NOsuchCONNECTION • NOTyetBEGUN • NOTyetOPEN • TCPipSHUTDOWN • INVALIDvirtualADDRESS • SOFTWAREerror • REMOTEreset
-------------------	---

For a description of Pascal return codes, see Table 25 on page 723.

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 177 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. (Refer to the *z/OS Communications Server: IP Configuration Reference*.)

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

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

Figure 177. *TcpNameChange* example

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

TcpOpen and TcpWaitOpen

The examples in this section 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 178. *TcpOpen* example

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

Figure 179. *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	<p>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 to 65 534). You can specify the address, the port, or both if particular values are required by your application. The address must be a valid home address for your node, and the port must be available (not reserved, and not in use by another application).</p> <p>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.</p>
ForeignSocket	<p>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.</p> <p>Passive Open: If your program is offering a service to anyone who wants it, specify an address of</p>

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: <ul style="list-style-type: none"> • 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 25 on page 723.

TcpOption

Use the procedure shown in Figure 180 to set an option for a TCP connection.

```

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

```

Figure 180. *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.				
<table> <tr> <th>Name</th><th>Description</th></tr> <tr> <td>OPTIONtcpKEEPALIVE</td><td> <p>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</p> </td></tr> </table>		Name	Description	OPTIONtcpKEEPALIVE	<p>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</p>
Name	Description				
OPTIONtcpKEEPALIVE	<p>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</p>				

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 25 on page 723.

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 181 shows an example of `TcpStatus`.

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

Figure 181. *TcpStatus* example

Parameter	Description
-----------	-------------

ConnectionInfo	
-----------------------	--

	If <code>ReturnCode</code> 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 <code>WAITforever</code> .
--	--

BytesToRead	
--------------------	--

	The number of bytes of incoming data queued for your program (waiting for <code>TcpReceive</code> , <code>TcpFReceive</code> , or <code>TcpWaitReceive</code>).
--	--

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 25 on page 723.

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 182 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 182. UdpClose example

Parameter	Description
ConnIndex	The ConnIndex value returned from UdpOpen.
ReturnCode	Indicates success or failure of a call. Possible return values are: <ul style="list-style-type: none"> • OK • NOsuchCONNECTION • NOTyetBEGUN • TCPipSHUTDOWN • SOFTWAREerror

For a description of Pascal return codes, see Table 25 on page 723.

UdpNReceive

The procedure shown in Figure 183 on page 751 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 183. 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 65 507 bytes, only the first 65 507 bytes are used.
ReturnCode	Indicates success or failure of a call. Possible return values are: <ul style="list-style-type: none"> • OK • ABNORMALcondition • FATALerror • NOsuchCONNECTION • NOTyetBEGUN • RECEIVEstillPENDING • TCPipSHUTDOWN • SOFTWAREerror • BADlengthARGUMENT • INVALIDvirtualADDRESS

For a description of Pascal return codes, see Table 25 on page 723.

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 184 on page 752 shows an example.

```

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

```

Figure 184. 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: <ul style="list-style-type: none"> • OK • ABNORMALcondition • FATALerror • LOCALportNOTavailable • NObufferSPACE • NOTyetBEGUN • SOFTWAREerror • TCPIPshutdown • UDPlocalADDRESS • TOOManyOPENS • UNAUTHORIZEDuser

For a description of Pascal return codes, see Table 25 on page 723.

UdpReceive

The procedure shown in Figure 185 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 185. 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: <ul style="list-style-type: none"> • OK • ABNORMALcondition • FATALerror • NOsuchCONNECTION • NOTyetBEGUN • SOFTWAREerror • TCPipSHUTDOWN • INVALIDvirtualADDRESS

For a description of Pascal return codes, see Table 25 on page 723.

UdpSend

The procedure shown in Figure 186 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 186. 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: <ul style="list-style-type: none"> • OK • BADlengthARGUMENT • NObufferSPACE • NOsuchCONNECTION • NOTyetBEGUN

- SOFTWAREerror
- TCPipSHUTDOWN
- UDPunspecADDRESS
- UDPunspecPORT
- INVALIDvirtualADDRESS

For a description of Pascal return codes, see Table 25 on page 723.

Unhandle

Use the procedure shown in Figure 187 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 187. 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: <ul style="list-style-type: none"> • OK • NOTyetBEGUN • INVALIDrequest

For a description of Pascal return codes, see Table 25 on page 723.

Sample Pascal program

This section contains an example of a Pascal application program. The source code can be found in the *hlq.SEZAINST* data set.

Building the sample Pascal API module

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

The following steps describe how to run the sample module:

1. Run the Pascal API sample program with the Receive option (shown in Figure 188 on page 755).

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 188. 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 189).

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 189. Sample Pascal API with send option

Sample Pascal application program

The following is an example of 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 }

```

Figure 190. Sample Pascal application program (Part 1 of 6)

```

    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)) );
    end;
end;

```

Figure 190. Sample Pascal application program (Part 2 of 6)

```

        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;

```

Figure 190. Sample Pascal application program (Part 3 of 6)


```

        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;
                {issue next TcpFReceive }
            end
    end
end

```

Figure 190. Sample Pascal application program (Part 4 of 6)

```

    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

```

Figure 190. Sample Pascal application program (Part 5 of 6)

```

    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 190. Sample Pascal application program (Part 6 of 6)

Part 4. Appendixes

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 `hlq.SEZAINST` data set.

```

/**** IBMCOPYR *****/
/*
/* Component Name: MTCSRVR (alias EZAEC047)
/*
/*
/* 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: EZAEC049
/*
/*
/**** IBMCOPYR *****/

/*****/
/* 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:
/*     . server master
/*     . server subtask - separate TCB within server address space
/*     . client
/*
/*****/

static char ibmcopyr[] =
    "MTCSRVR - 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>

```

Figure 191. MTCSRVR C socket server program sample (Part 1 of 7)


```

#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);
}

```

Figure 191. MTCSRVR C socket server program sample (Part 2 of 7)

```

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);

```

Figure 191. MTCSRVR C socket server program sample (Part 3 of 7)

```

        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);
        /****** do this simplified situation first *****/
        exit(0); /******
        /******
    } /* 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 */

```

Figure 191. MTCSRVR C socket server program sample (Part 4 of 7)

```

    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 writmask;
    struct fd_set excpmask;
    int rc;
    struct timeval time;

    temps = *s;
    time.tv_sec = 1000;

```

Figure 191. MTCSRVR C socket server program sample (Part 5 of 7)

```

time.tv_usec = 0;
maxfdpl = temps + 1;

FD_ZERO(&readmask);
FD_ZERO(&writmask);
FD_ZERO(&excpmask);

FD_SET(temps, &readmask);

rc = select(maxfdpl, &readmask, &writmask, &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 writmask;
    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(&writmask);
    FD_ZERO(&excpmask);

    /* FD_SET(temps, &readmask); */
    /* FD_SET(temps, &writmask); */
    /* FD_SET(temps, &excpmask); */

```

Figure 191. MTCSRVR C socket server program sample (Part 6 of 7)

```

    rc = select(maxfdpl, &readmask, &writmask, &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;
}

```

Figure 191. MTCSRVR C socket server program sample (Part 7 of 7)

The subtask sample program in C

The following C socket server program is the MTCCSUB member in the *hlq.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/recv 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

```

Figure 192. MTCCSUB C socket server program sample (Part 1 of 3)

```

*/
csub(int *clsock,      /* address of socket passed */
     char *tskname,    /* address of caller's name */
     char *xtskname)   /* 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;
}

/*-----*/

```

Figure 192. MTCCSUB C socket server program sample (Part 2 of 3)


```

/*    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 rcvbytes;
    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;
    rcvbytes = recv(temps, data, sizeof(data), 0);
    if (rcvbytes > 0)
        printf("Csub: data rcv: %s\n", data);
    else
        printf("Csub: client stopped sending data\n");
    printf("Csub: rcvbytes = %d\n", rcvbytes);
    return rcvbytes;
}

```

Figure 192. MTCCSUB C socket server program sample (Part 3 of 3)

The client sample program in C

The following C socket server program is the MTCCLNT member in the *hlq.SEZAINST* data set.

```

/**** IBMCOPYR *****/
/*
/* Component Name: MTCCLNT
/*
/*
/* 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[] =
    "MTCCLNT - 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);

```

Figure 193. MTCCLNT C socket server program sample (Part 1 of 4)

```

/*
 * 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);
    }
}

```

Figure 193. MTCCLNT C socket server program sample (Part 2 of 4)

```

}

/*-----*/
/*  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()    */
/*-----*/

```

Figure 193. MTCCLNT C socket server program sample (Part 3 of 4)

```

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 193. MTCCLNT C socket server program sample (Part 4 of 4)

Appendix B. Return codes

This appendix contains system error return codes for socket calls. They apply to all socket APIs in this book. 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 appendix, it is a return code that is received from z/OS UNIX. Refer to *z/OS UNIX System Services Messages and Codes* for the z/OS UNIX ERRNOs.

See “User abend U4093” on page 794 for a description of user abend U4093.

System error codes for socket calls

This section 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 26. Sockets ERRNOs

Error number	Message name	Socket type	Error description	Programmer's response
1	EAL_NONAME	GETADDRINFO GETNAMEINFO	NODE or HOST cannot be found.	Ensure the NODE or HOST name can be resolved.
1	EPERM	All	Permission is denied. No owner exists.	Check that TPC/IP is still active; check protocol value of socket () call.
1	EDOM	All	Argument too large.	Check parameter values of the function call.

Table 26. Sockets ERRNOs (continued)

Error number	Message name	Socket type	Error description	Programmer's response
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	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.
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.

Table 26. Sockets ERRNOs (continued)

Error number	Message name	Socket type	Error description	Programmer's response
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.
12	ENOMEM	All	There is not enough storage.	Check validity of function parameters.
13	EACCES	All	Permission denied, caller not authorized.	Check access authority of file.
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 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.
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 validity of function parameters.
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.

Table 26. Sockets ERRNOs (continued)

Error number	Message name	Socket type	Error description	Programmer's response
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.
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	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	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().
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.
37	EALREADY	Connect	The socket is marked nonblocking and the previous connection has not been completed.	Reissue Connect().
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 SETIBMOP (IBMTCP_IMAGE), has already been made.	Only call Setibmopt() 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	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.

Table 26. Sockets ERRNOs (continued)

Error number	Message name	Socket type	Error description	Programmer's response
41	EPROTOTYPE	All	The specified protocol type is incorrect for this socket.	Correct the protocol type parameter.
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	Getibmssockopt Setibmssockopt	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	IOCTL	The specified IOCTL command is not supported by this socket API.	Correct the IOCTL COMMAND.
45	EOPNOTSUPP	RECV, RECVFROM, RECVMMSG, SEND, SENDTO, SENDMSG	The specified flags are not supported on this socket type or protocol.	Correct the FLAG.
45	EOPNOTSUPP	Accept Givesocket	The selected socket is not a stream socket.	Use a valid socket.
45	EOPNOTSUPP	Listen	The socket does not support the Listen call.	Change the type on the Socket() call when the socket was created. Listen() only supports a socket type of SOCK_STREAM.
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().
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	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.
48	EADDRINUSE	Bind	The address is in a timed wait because a LINGER delay from a previous close or another process is using the address.	If you want to reuse the same address, use Setsockopt() with SO_REUSEADDR. See Setsockopt(). Otherwise, use a different address or port in the socket address structure.

Table 26. Sockets ERRNOs (continued)

Error number	Message name	Socket type	Error description	Programmer's response
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.
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	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.
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.
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.

Table 26. Sockets ERRNOs (continued)

Error number	Message name	Socket type	Error description	Programmer's response
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.
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 CS 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 not timeout 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.
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.

Table 26. Sockets ERRNOs (continued)

Error number	Message name	Socket type	Error description	Programmer's response
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.
87	EMULTIHOP	All	A multihop 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.
145	E2BIG	All	The argument list is too long.	Eliminate excessive number of arguments.
156	EMVSINITIAL	All	Process initialization error. 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 may not be defined or may contain errors such as an improper HOME() directory specification.	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.
1002	EIBMSOCKOUTOFRANGE	Socket	A socket number assigned by the client interface code is out of range.	Check the socket descriptor parameter.
1003	EIBMSOCKINUSE	Socket	A socket number assigned by the client interface code is already in use.	Use a different socket descriptor.

Table 26. Sockets ERRNOs (continued)

Error number	Message name	Socket type	Error description	Programmer's response
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.
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	EIBMINVTCPCONNECTION	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	EIBMINVTSRBUSERDATA	All	The request control block contained data that is not valid.	Call your system programmer.
1038	EIBMINVUSERDATA	All	The request control block contained user data that is not valid.	Check your function parameters and call your system programmer.

Table 26. Sockets ERRNOs (continued)

Error number	Message name	Socket type	Error description	Programmer's response
1040	EIBMSELECTEXPOST	SELECTEX	SELECTEX passed an ECB that was already posted.	Check whether the user's ECB was already posted.
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.	Only issue the INITIALIZE call once in your program.
2005	ESUBTASKALNOTACTIVE	REXX	The subtask is not active.	Issue the INITIALIZE call before any other socket call.
2006	ESOCKETNETNOTALLOCATED	REXX	The specified socket 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.
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.

z/OS UNIX return codes

All return codes not listed in either “Sockets return codes (ERRNOs)” on page 781 or “Sockets extended ERRNOs” on page 791 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(BPXVERNO) and are defined in *z/OS UNIX System Services Messages and Codes*.

For more information about z/OS UNIX error codes, refer to *z/OS UNIX System Services Messages and Codes*.

Additional return codes

The following section 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 27. 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.	For a list of valid first calls, refer to the section on special considerations in the chapter on general programming.
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 27. 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 GETSOCKOPT option length is less than the minimum or greater 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.	For a list of valid first calls, refer to the section on special considerations in the chapter on general programming.
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.
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.

Table 27. Sockets extended ERRNOs (continued)

Error code	Problem description	System action	Programmer's response
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.
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, RECVMMSG, SENDMSG, or WRITEV call.	End the call.	Correct the IOVCNT parameter.

Table 27. Sockets extended ERRNOs (continued)

Error code	Problem description	System action	Programmer's response
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 sockets 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.
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 Sockets Interface before executing this call.
20003	The CICS Sockets Interface is not in operation.	End the call.	Contact the CICS Systems programmer. Ensure that the user ID being used is permitted to have at least UPDATE access to the EXITPROGRAM resource.

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 194 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 194. Example of abend U4093

Appendix C. Address family cross reference

This appendix contains AF_INET, AF_INET6, and AF_IUCV address family cross reference information for the major APIs in this book.

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.

Note:

1. Pascal API supports only AF_INET address family.
2. XTI API supports only AF_INET address family.

Note: In the following table, INET6 is not supported.

Table 28. 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
endnetent()	yes	n/a
endprotoent()	yes	n/a
endservent()	yes	n/a
fcntl()	yes	no
getclientid()	yes	no
getdtablesize()	yes	yes

Table 28. C socket address families cross reference (continued)

Application Programming Interface (API)		
Function	C SOCKETS	
	INET	IUCV
gethostbyaddr()	yes	no
gethostbyname()	yes	n/a
gethostent()	yes	n/a
gethostid()	yes	no
gethostname()	yes	no
getibmopt()	yes	no
getibmssockopt()	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_ianaof()	yes	n/a
inet_makeaddr()	yes	n/a
inet_netof()	yes	n/a
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

Table 28. C socket address families cross reference (continued)

Application Programming Interface (API)		
Function	C SOCKETS	
	INET	IUCV
select()	yes	yes
selectex()	yes	yes
send()	yes	yes
sendmsg()	yes	no
sendto()	yes	no
setibmopt()	yes	no
setibmssockopt()	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
writerv()	yes	yes

Note: In the following table, IUCV is not supported.

Table 29. 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
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

Table 29. MACRO, CALL, REXX, socket address families cross reference (continued)

Application Programming Interface (API)						
COMMAND	MACRO		CALL		REXX	
	INET	INET6	INET	INET6	INET	INET6
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
GETSERVBYPORT					yes	yes
GETSOCKNAME	yes	yes	yes	yes	yes	yes
GETSOCKOPT see Table 30 on page 801 for exceptions.	yes	yes	yes	yes	yes	yes
GIVESOCKET	yes	yes	yes	yes	yes	yes
GLOBAL	yes	yes	yes	yes		
INITAPI	yes	yes	yes	yes		
IOCTL see Table 30 on page 801 for exceptions.	yes	yes	yes	yes	yes	yes
LISTEN	yes	yes	yes	yes	yes	yes
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 30 on page 801 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		

Table 29. MACRO, CALL, REXX, socket address families cross reference (continued)

Application Programming Interface (API)						
COMMAND	MACRO		CALL		REXX	
	INET	INET6	INET	INET6	INET	INET6
VERSION					yes	yes
WRITE	yes	yes	yes	yes	yes	yes
WRITEV	yes	yes	yes	yes		

Table 30. 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_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_DROP_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
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	yes	yes	yes	yes		
SIOCGIFNAMEINDEX	yes	yes	yes	yes	yes	yes

Appendix D. GETSOCKOPT/SETSOCKOPT command values

You can use the table below 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 31. GETSOCKOPT/SETSOCKOPT command values for Macro, Assembler, and PL/I

Command name	Decimal value	Hex value
IP_ADD_MEMBERSHIP	1048581	X'00100005'
IP_DROP_MEMBERSHIP	1048582	X'00100006'
IP_MULTICAST_IF	1048583	X'00100007'
IP_MULTICAST_LOOP	1048580	X'00100004'
IP_MULTICAST_TTL	1048579	X'00100003'
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'
SO_BROADCAST	32	X'00000020'
SO_ERROR	4103	X'00001007'
SO_LINGER	128	X'00000080'
SO_KEEPALIVE	8	X'00000008'
SO_OOBINLINE	256	X'00000100'
SO_RCVBUF	4098	X'00001002'
SO_REUSEADDR	4	X'00000004'
SO_SNDBUF	4097	X'00001001'
SO_TYPE	4104	X'00001008'
TCP_NODELAY	2147483649	X'80000001'

Table 32. GETSOCKOPT/SETSOCKOPT optname value for C programs

Option name	Decimal value
IP_ADD_MEMBERSHIP	5
IP_DROP_MEMBERSHIP	6

Table 32. GETSOCKOPT/SETSOCKOPT optname value for C programs (continued)

IP_MULTICAST_IF	7
IP_MULTICAST_LOOP	4
IP_MULTICAST_TTL	3
SO_ACCEPTCONN	2
SO_BROADCAST	32
SO_CLUSTERCONNTYPE	16385
SO_DEBUG	1
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_NODELAY	1

Appendix E. Abbreviations and acronyms

This appendix lists the abbreviations and acronyms used throughout this book.

AIX®	Advanced Interactive Executive
ANSI	American National Standards Institute
API	Application program interface
APPC	Advanced Program-to-Program Communications
APPN	Advanced Peer-to-Peer Networking®
ARP	Address Resolution Protocol
ASCII	American National Standard Code for Information Interchange
ASN.1	Abstract Syntax Notation One
AUI	Attachment Unit Interface
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 National Language 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. Related protocol specifications (RFCs)

This appendix lists the related protocol specifications 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.

You can request RFCs through electronic mail, from the automated Network Information Center (NIC) mail server, by sending a message to `service@nic.ddn.mil` with a subject line of RFC *nnnn* for text versions or a subject line of RFC *nnnn*.PS for PostScript versions. To request a copy of the RFC index, send a message with a subject line of RFC INDEX.

For more information, contact `nic@nic.ddn.mil` or at:

Government Systems, Inc.
Attn: Network Information Center
14200 Park Meadow Drive
Suite 200
Chantilly, VA 22021

Hard copies of all RFCs are available from the NIC, either individually or by subscription. Online copies are available at the following Web address:
<http://www.rfc-editor.org/rfc.html>.

See "Internet Drafts" on page 820 for draft RFCs implemented in this and previous Communications Server releases.

Many features of TCP/IP Services are based on the following RFCs:

RFC	Title and Author
768	<i>User Datagram Protocol</i> J. Postel
791	<i>Internet Protocol</i> J. Postel
792	<i>Internet Control Message Protocol</i> J. Postel
793	<i>Transmission Control Protocol</i> J. Postel
821	<i>Simple Mail Transfer Protocol</i> J. Postel
822	<i>Standard for the Format of ARPA Internet Text Messages</i> D. Crocker
823	<i>DARPA Internet Gateway</i> R. Hinden, A. Sheltzer
826	<i>Ethernet Address Resolution Protocol or Converting Network Protocol Addresses to 48.Bit Ethernet Address for Transmission on Ethernet Hardware</i> D. Plummer
854	<i>Telnet Protocol Specification</i> J. Postel, J. Reynolds
855	<i>Telnet Option Specification</i> J. Postel, J. Reynolds
856	<i>Telnet Binary Transmission</i> J. Postel, J. Reynolds

- 857 *Telnet Echo Option* J. Postel, J. Reynolds
- 858 *Telnet Suppress Go Ahead Option* J. Postel, J. Reynolds
- 859 *Telnet Status Option* J. Postel, J. Reynolds
- 860 *Telnet Timing Mark Option* J. Postel, J. Reynolds
- 861 *Telnet Extended Options—List Option* J. Postel, J. Reynolds
- 862 *Echo Protocol* J. Postel
- 863 *Discard Protocol* J. Postel
- 864 *Character Generator Protocol* J. Postel
- 877 *Standard for the Transmission of IP Datagrams over Public Data Networks* J. Korb
- 885 *Telnet End of Record Option* J. Postel
- 894 *Standard for the transmission of IP datgrams over Ethernet networks* C. Hornig
- 896 *Congestion Control in IP/TCP Internetworks* J. Nagle
- 903 *Reverse Address Resolution Protocol* R. Finlayson, T. Mann, J. Mogul, M. Theimer
- 904 *Exterior Gateway Protocol Formal Specification* D. Mills
- 919 *Broadcasting Internet Datagrams* J. Mogul
- 922 *Broadcasting Internet Datagrams in the Presence of Subnets* J. Mogul
- 950 *Internet Standard Subnetting Procedure* J. Mogul, J. Postel
- 951 *Bootstrap Protocol* W.J. Croft, J. Gilmore
- 952 *DoD Internet Host Table Specification* K. Harrenstien, M. Stahl, E. Feinler
- 959 *File Transfer Protocol* J. Postel, J. Reynolds
- 974 *Mail Routing and the Domain Name System* C. Partridge
- 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
- 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
- 1006 *ISO Transport Service on top of the TCP Version 3* M. Rose, D. Cass
- 1009 *Requirements for Internet Gateways* R. Braden, J. Postel
- 1011 *Official Internet Protocols* J. Reynolds, J. Postel
- 1013 *X Window System Protocol, Version 11: Alpha Update* R. Scheifler
- 1014 *XDR: External Data Representation Standard* Sun Microsystems Incorporated
- 1027 *Using ARP to Implement Transparent Subnet Gateways* S. Carl-Mitchell, J. Quarterman
- 1032 *Domain Administrators Guide* M. Stahl
- 1033 *Domain Administrators Operations Guide* M. Lottor
- 1034 *Domain Names—Concepts and Facilities* P. Mockapetris
- 1035 *Domain Names—Implementation and Specification* P. Mockapetris

- 1042 *Standard for the Transmission of IP Datagrams over IEEE 802 Networks* J. Postel, J. Reynolds
- 1044 *Internet Protocol on Network System's HYPERchannel: Protocol Specification* K. Hardwick, J. Lekashman
- 1055 *Nonstandard for Transmission of IP Datagrams over Serial Lines: SLIP* J. Romkey
- 1057 *RPC: Remote Procedure Call Protocol Version 2 Specification* Sun Microsystems Incorporated
- 1058 *Routing Information Protocol* C. Hedrick
- 1060 *Assigned Numbers* J. Reynolds, J. Postel
- 1073 *Telnet Window Size Option* D. Waitzman
- 1079 *Telnet Terminal Speed Option* C. Hedrick
- 1091 *Telnet Terminal-Type Option* J. VanBokkelen
- 1094 *NFS: Network File System Protocol Specification* Sun Microsystems Incorporated
- 1096 *Telnet X Display Location Option* G. Marcy
- 1101 *DNS encoding of network names and other types* P. Mockapetris
- 1112 *Host Extensions for IP Multicasting* S. Deering
- 1118 *Hitchhikers Guide to the Internet* E. Krol
- 1122 *Requirements for Internet Hosts—Communication Layers* R. Braden
- 1123 *Requirements for Internet Hosts—Application and Support* R. Braden
- 1155 *Structure and Identification of Management Information for TCP/IP-Based Internets* M. Rose, K. McCloghrie
- 1156 *Management Information Base for Network Management of TCP/IP-Based Internets* K. McCloghrie, M. Rose
- 1157 *Simple Network Management Protocol (SNMP)* J. Case, M. Fedor, M. Schoffstall, C. Davin
- 1158 *Management Information Base for Network Management of TCP/IP-based internets: MIB-II* M. Rose
- 1179 *Line Printer Daemon Protocol* The Wollongong Group, L. McLaughlin III
- 1180 *TCP/IP Tutorial* T. Socolofsky, C. Kale
- 1183 *New DNS RR Definitions* C. Everhart, L. Mamakos, R. Ullmann, P. Mockapetris, (Updates RFC 1034, RFC 1035)
- 1184 *Telnet Linemode Option* D. Borman
- 1187 *Bulk Table Retrieval with the SNMP* M. Rose, K. McCloghrie, J. Davin
- 1188 *Proposed Standard for the Transmission of IP Datagrams over FDDI Networks* D. Katz
- 1191 *Path MTU Discovery* J. Mogul, S. Deering
- 1198 *FYI on the X Window System* R. Scheifler
- 1207 *FYI on Questions and Answers: Answers to Commonly Asked "Experienced Internet User" Questions* G. Malkin, A. Marine, J. Reynolds

- 1208 *Glossary of Networking Terms* O. Jacobsen, D.Lynch
- 1213 *Management Information Base for Network Management of TCP/IP-Based Internets: MIB-II* K. McCloghrie, M. Rose
- 1215 *Convention for Defining Traps for Use with the SNMP* M. Rose
- 1228 *SNMP-DPI Simple Network Management Protocol Distributed Program Interface* G.C. Carpenter, B. Wijnen
- 1229 *Extensions to the Generic-Interface MIB* K. McCloghrie
- 1230 *IEEE 802.4 Token Bus MIB* K. McCloghrie, R. Fox
- 1231 *IEEE 802.5 Token Ring MIB* K. McCloghrie, R. Fox, E. Decker
- 1236 *IP to X.121 Address Mapping for DDN* L. Morales, P. Hasse
- 1267 *A Border Gateway Protocol 3 (BGP-3)* K. Lougheed, Y. Rekhter
- 1268 *Application of the Border Gateway Protocol in the Internet* Y. Rekhter, P. Gross
- 1269 *Definitions of Managed Objects for the Border Gateway Protocol (Version 3)* S. Willis, J. Burruss
- 1270 *SNMP Communications Services* F. Kastenholz, ed.
- 1321 *The MD5 Message-Digest Algorithm* R. Rivest
- 1323 *TCP Extensions for High Performance* V. Jacobson, R. Braden, D. Borman
- 1325 *FYI on Questions and Answers: Answers to Commonly Asked "New Internet User" Questions* G. Malkin, A. Marine
- 1340 *Assigned Numbers* J. Reynolds, J. Postel
- 1348 *DNS NSAP RRs* B. Manning
- 1349 *Type of Service in the Internet Protocol Suite* P. Almquist
- 1350 *TFTP Protocol* K.R. Sollins
- 1351 *SNMP Administrative Model* J. Davin, J. Galvin, K. McCloghrie
- 1352 *SNMP Security Protocols* J. Galvin, K. McCloghrie, J. Davin
- 1353 *Definitions of Managed Objects for Administration of SNMP Parties* K. McCloghrie, J. Davin, J. Galvin
- 1354 *IP Forwarding Table MIB* F. Baker
- 1356 *Multiprotocol Interconnect on X.25 and ISDN in the Packet Mode* A. Malis, D. Robinson, R. Ullmann
- 1363 *A Proposed Flow Specification* C. Partridge
- 1372 *Telnet Remote Flow Control Option* D. Borman, C. L. Hedrick
- 1374 *IP and ARP on HIPPI* J. Renwick, A. Nicholson
- 1381 *SNMP MIB Extension for X.25 LAPB* D. Throop, F. Baker
- 1382 *SNMP MIB Extension for the X.25 Packet Layer* D. Throop
- 1387 *RIP Version 2 Protocol Analysis* G. Malkin
- 1388 *RIP Version 2—Carrying Additional Information* G. Malkin
- 1389 *RIP Version 2 MIB Extension* G. Malkin
- 1390 *Transmission of IP and ARP over FDDI Networks* D. Katz

- 1393 *Traceroute Using an IP Option* G. Malkin
- 1397 *Default Route Advertisement In BGP2 And BGP3 Versions of the Border Gateway Protocol* D. Haskin
- 1398 *Definitions of Managed Objects for the Ethernet-Like Interface Types* F. Kastenholz
- 1408 *Telnet Environment Option* D.Borman, Ed.
- 1416 *Telnet Authentication Option* D. Borman, ed.
- 1464 *Using the Domain Name System to Store Arbitrary String Attributes* R. Rosenbaum
- 1469 *IP Multicast over Token-Ring Local Area Networks* T. Pusateri
- 1497 *BOOTP Vendor Information Extensions* J.Reynolds
- 1533 *DHCP Options and BOOTP Vendor Extensions* S.Alexander, R.Droms
- 1534 *Interoperation Between DHCP and BOOTP* R.Droms
- 1535 *A Security Problem and Proposed Correction With Widely Deployed DNS Software* E. Gavron
- 1536 *Common DNS Implementation Errors and Suggested Fixes* A. Kumar, J. Postel, C. Neuman, P. Danzig, S.Miller
- 1537 *Common DNS Data File Configuration Errors* P. Beertema
- 1540 *IAB Official Protocol Standards* J. Postel
- 1541 *Dynamic Host Configuration Protocol* R.Droms
- 1542 *Clarifications and Extensions for the Bootstrap Protocol* W.Wimer
- 1571 *Telnet Environment Option Interoperability Issues* D. Borman
- 1572 *Telnet Environment Option* S. Alexander
- 1577 *Classical IP and ARP over ATM* M. Laubach
- 1583 *OSPF Version 2* J. Moy
- 1591 *Domain Name System Structure and Delegation* J. Postel
- 1592 *Simple Network Management Protocol Distributed Protocol Interface Version 2.0* B. Wijnen, G. Carpenter, K. Curran, A. Sehgal, G. Waters
- 1594 *FYI on Questions and Answers: Answers to Commonly Asked "New Internet User" Questions* A. Marine, J. Reynolds, G. Malkin
- 1646 *TN3270 Extensions for LUname and Printer Selection* C.Graves, T.Butts, M.Angel
- 1647 *TN3270 Enhancement* B.Kelly
- 1695 *Definitions of Managed Objects for ATM Management Version 8.0 Using SMIPv2* M. Ahmed, K. Tesink
- 1706 *DNS NSAP Resource Records* B. Manning, R. Colella
- 1713 *Tools for DNS debugging* A. Romao
- 1723 *RIP Version 2—Carrying Additional Information* G. Malkin
- 1766 *Tags for the Identification of Languages* H. Alvestrand
- 1794 *DNS Support for Load Balancing* T. Brisco

- 1832 *XDR: External Data Representation Standard* R. Srinivasan
- 1840 *Schema Publishing in X500 Directory* G.Mansfield, P.Rajeev, S.Raghavan, T.Howes
- 1850 *OSPF Version 2 Management Information Base* F. Baker, R. Coltun
- 1876 *A Means for Expressing Location Information in the Domain Name System* C. Davis, P. Vixie, T. Goodwin, I. Dickinson
- 1886 *DNS Extensions to support IP version 6* S. Thomson, C. Huitema
- 1901 *Introduction to Community-Based SNMPv2* J. Case, K. McCloghrie, M. Rose, S. Waldbusser
- 1902 *Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMPv2)* J. Case, K. McCloghrie, M. Rose, S. Waldbusser
- 1903 *Textual Conventions for Version 2 of the Simple Network Management Protocol (SNMPv2)* J. Case, K. McCloghrie, M. Rose, S. Waldbusser
- 1904 *Conformance Statements for Version 2 of the Simple Network Management Protocol (SNMPv2)* J. Case, K. McCloghrie, M. Rose, S. Waldbusser
- 1905 *Protocols Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)* J. Case, K. McCloghrie, M. Rose, S. Waldbusser
- 1906 *Transport Mappings for Version 2 of the Simple Network Management Protocol (SNMPv2)* J. Case, K. McCloghrie, M. Rose, S. Waldbusser
- 1907 *Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)* J. Case, K. McCloghrie, M. Rose, S. Waldbusser
- 1908 *Coexistence between Version 1 and Version 2 of the Internet-Standard Network Management Framework* J. Case, K. McCloghrie, M. Rose, S. Waldbusser
- 1912 *Common DNS Operational and Configuration Errors* D. Barr
- 1918 *Address Allocation for Private Internets* Y. Rekhter, B. Moskowitz, D. Karrenberg, G.J. de Groot, E. Lear
- 1928 *SOCKS Protocol Version 5* M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, L. Jones
- 1939 *Post Office Protocol-Version 3* J. Myers, M. Rose
- 1981 *Path MTU Discovery for IP version 6* J. McCann, S. Deering, J. Mogul
- 1982 *Serial Number Arithmetic* R. Elz, R. Bush
- 1995 *Incremental Zone Transfer in DNS* M. Ohta
- 1996 *A Mechanism for Prompt Notification of Zone Changes (DNS NOTIFY)* P. Vixie
- 2010 *Operational Criteria for Root Name Servers* B. Manning, P. Vixie
- 2011 *SNMPv2 Management Information Base for the Internet Protocol Using SMIv2* K. McCloghrie
- 2012 *SNMPv2 Management Information Base for the Transmission Control Protocol Using SMIv2* K. McCloghrie
- 2013 *SNMPv2 Management Information Base for the User Datagram Protocol Using SMIv2* K. McCloghrie
- 2030 *Simple Network Time Protocol* D. Mills

- 2052 *A DNS RR for specifying the location of services (DNS SRV)* A. Gulbrandsen, P. Vixie
- 2065 *Domain Name System Security Extensions* D. Eastlake, C. Kaufman
- 2080 *RIPng for IPv6* G. Malkin, R. Minnear
- 2096 *IP Forwarding Table MIB* F. Baker
- 2104 *HMAC: Keyed-Hashing for Message Authentication* H. Krawczyk, M. Bellare, R. Canetti
- 2132 *DHCP Options and BOOTP Vendor Extensions* S. Alexander, R. Droms
- 2133 *Basic Socket Interface Extensions for IPv6* R. Gilligan, S. Thomson, J. Bound, W. Stevens
- 2136 *Dynamic Updates in the Domain Name System (DNS UPDATE)* P.Vixie, Ed.,S.Thomson, Y.Rekhter,J.Bound
- 2137 *Secure Domain Name System Dynamic Update* D. Eastlake
- 2163 *Using the Internet DNS to Distribute MIXER Conformant Global Address Mapping (MCGAM)* C. Allocchio
- 2168 *Resolution of Uniform Resource Identifiers using the Domain Name System* R. Daniel, M. Mealling
- 2178 *OSPF Version 2* J. Moy
- 2181 *Clarifications to the DNS Specification* R. Elz, R. Bush
- 2205 *Resource ReSerVation Protocol (RSVP) Version 1* R. Braden, L. Zhang, S. Berson, S. Herzog, S. Jamin
- 2210 *The Use of RSVP with IETF Integrated Services* J. Wroclawski
- 2211 *Specification of the Controlled-Load Network Element Service* J. Wroclawski
- 2212 *Specification of Guaranteed Quality of Service* S. Shenker, C. Partridge, R. Guerin
- 2215 *General Characterization Parameters for Integrated Service Network Elements* S. Shenker, J. Wroclawski
- 2219 *Use of DNS Aliases for Network Services* M. Hamilton, R. Wright
- 2228 *FTP Security Extensions* M. Horowitz, S. Lunt
- 2230 *Key Exchange Delegation Record for the DNS* R. Atkinson
- 2233 *The Interfaces Group MIB Using SMIV2* K. McCloghrie, F. Kastenholz
- 2240 *A Legal Basis for Domain Name Allocation* O. Vaughn
- 2246 *The TLS Protocol Version 1.0* T. Dierks, C. Allen
- 2251 *Lightweight Directory Access Protocol (v3)* M.Wahl,T.Howes, S.Kille
- 2308 *Negative Caching of DNS Queries (DNS NCACHE)* M. Andrews
- 2317 *Classless IN-ADDR.ARPA delegation* H. Eidnes, G. de Groot, P. Vixie
- 2320 *Definitions of Managed Objects for Classical IP and ARP over ATM Using SMIV2* M. Greene, J. Luciani, K. White, T. Kuo
- 2328 *OSPF Version 2* J. Moy
- 2345 *Domain Names and Company Name Retrieval* J. Klensin, T. Wolf, G. Oglesby
- 2352 *A Convention for Using Legal Names as Domain Names* O. Vaughn

- 2355 *TN3270 Enhancements* B. Kelly
- 2373 *IP Version 6 Addressing Architecture* R. Hinden, M. O'Dell, S. Deering
- 2374 *An IPv6 Aggregatable Global Unicast Address Format* R. Hinden, M. O'Dell, S. Deering
- 2375 *IPv6 Multicast Address Assignments* R. Hinden, S. Deering
- 2389 *Feature negotiation mechanism for the File Transfer Protocol* P. Hethmon, R. Elz
- 2401 *Security Architecture for Internet Protocol* S.Kent, R. Atkinson
- 2402 *IP Authentication Header* S.Kent, R. Atkinson
- 2403 *HMAC-MD5-96 within ESP and AH* Madson, R. Glenn
- 2404 *The Use of HMAC-MD5-96 within ESP and AH* C. Madson, R. Glenn
- 2405 *The ESP DES-CBC Cipher Algorithm With Explicit IVC* Madson, N. Doraswamy
- 2406 *IP Encapsulating Security Payload (ESP)* S.Kent, R. Atkinson
- 2409 *The Internet Key Exchange (IKE)* D. Harkins, D. Carrel
- 2410 *The NULL Encryption Algorithm and Its Use With IPsec* R. Glenn, S. Kent,
- 2428 *FTP Extensions for IPv6 and NATs* M. Allman, S. Ostermann, C. Metz
- 2460 *Internet Protocol, Version 6 (IPv6) Specification* S. Deering, R. Hinden
- 2461 *Neighbor Discovery for IP Version 6 (IPv6)* T. Narten, E. Nordmark, W. Simpson
- 2462 *IPv6 Stateless Address Autoconfiguration* S. Thomson, T. Narten
- 2464 *Transmission of IPv6 Packets over Ethernet Networks* M. Crawford
- 2474 *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers* K. Nichols, S. Blake, F. Baker, D. Black
- 2487 *SMTP Service Extension for Secure SMTP over TLS* P. Hoffman
- 2505 *Anti-Spam Recommendations for SMTP MTAs* G. Lindberg
- 2535 *Domain Name System Security Extensions* D. Eastlake
- 2539 *Storage of Diffie-Hellman Keys in the Domain Name System (DNS)* D. Eastlake
- 2570 *Introduction to Version 3 of the Internet-standard Network Management Framework* J. Case, R. Mundy, D. Partain, B. Stewart
- 2571 *An Architecture for Describing SNMP Management Frameworks* D. Harrington, R. Presuhn, B. Wijnen
- 2572 *Message Processing and Dispatching for the Simple Network Management Protocol (SNMP)* J. Case, D. Harrington, R. Presuhn, B. Wijnen
- 2573 *SNMP Applications* D. Levi, P. Meyer, B. Stewart
- 2574 *User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3)* U. Blumenthal, B. Wijnen
- 2575 *View-based Access Control Model (VACM) for the Simple Network Management Protocol (SNMP)* B. Wijnen, R. Presuhn, K. McCloghrie
- 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

- 2578 *Structure of Management Information Version 2 (SMIv2)* K. McCloghrie, D. Perkins, J. Schoenwaelder
- 2635 *Don't SPEW A Set of Guidelines for Mass Unsolicited Mailings and Postings (spam*)* S.Hambridge, A.Lunde
- 2640 *Internationalization of the File Transfer Protocol* B. Curtin
- 2665 *Definitions of Managed Objects for the Ethernet-like Interface Types* J. Flick, J. Johnson
- 2672 *Non-Terminal DNS Name Redirection* M. Crawford
- 2710 *Multicast Listener Discovery (MLD) for IPv6* S. Deering, W. Fenner, B. Haberman
- 2711 *IPv6 Router Alert Option* C. Partridge, A. Jackson
- 2740 *OSPF for IPv6* R. Coltun, D. Ferguson, J. Moy
- 2758 *Definitions of Managed Objects for Service Level Agreements Performance Monitoring* K. White
- 2845 *Secret Key Transaction Authentication for DNS (TSIG)* P. Vixie, O. Gudmundsson, D. Eastlake, B. Wellington
- 2874 *DNS Extensions to Support IPv6 Address Aggregation and Renummering* M. Crawford, C. Huitema
- 2941 *Telnet Authentication Option* T. Ts'o, ed., J. Altman
- 2942 *Telnet Authentication: Kerberos Version 5* T. Ts'o
- 2946 *Telnet Data Encryption Option* T. Ts'o
- 2952 *Telnet Encryption: DES 64 bit Cipher Feedback* T. Ts'o
- 2953 *Telnet Encryption: DES 64 bit Output Feedback* T. Ts'o, ed.
- 3060 *Policy Core Information Model—Version 1 Specification* B. Moore, E. Ellessen, J. Strassner, A. Westerinen
- 3363 *Representing Internet Protocol version 6 (IPv6) Addresses in the Domain Name System* R.Bush, A.Durand, B.Fink, O.Gudmundsson,T.Hain
- 3390 *Increasing TCP's Initial Window* M. Allman, S. Floyd, C. Partridge
- 3411 *An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks* D.Harrington, R.Presuhn, B.Wijnen
- 3412 *Message Processing and Dispatching for the Simple Network Management Protocol (SNMP)* J.Case, D.Harrington, R.Presuhn, B.Wijnen
- 3413 *Simple Network Management Protocol (SNMP) Applications* D.Levi, P. Meyer, B.Stewart
- 3414 *User- Based Security Model (USM) for version 3 of the Simple Network management Protocol (SNMPv3)* U. Blumenthal, B. Wijnen
- 3415 *View-based Access Control Model (VACM) for the Simple Network Management Protocol (SNMP)* B.Wijnen, R.Presuhn, K.McCloghrie
- 3484 *Default Address Selection for Internet Protocol version 6 (IPv6)* R. Draves
- 3493 *Basic Socket Interface Extensions for IPv6* R. Gilligan, S. Thomson, J. Bound, J. McCann, W. Stevens
- 3513 *Internet Protocol Version 6 (IPv6) Addressing Architecture* R. Hinden, S. Deering

Internet Drafts

Internet drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Other groups may also distribute working documents as Internet drafts. You can see Internet drafts at <http://www.ietf.org/ID.html>.

Several areas of IPv6 implementation include elements of the following Internet drafts and are subject to change during the RFC review process.

Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification

A. Conta, S. Deering

Appendix G. Information APARs

This appendix lists information APARs for IP and SNA documents.

Notes:

1. Information APARs contain updates to previous editions of the manuals listed below. Documents updated for V1R6 are complete except for the updates contained in the information APARs that might be issued after V1R6 documents went to press.
2. Information APARs are predefined for z/OS V1R6 Communications Server and might not contain updates.
3. Information APARs for z/OS documents are in the document called *z/OS and z/OS.e DOC APAR and PTF ++HOLD Documentation*, which can be found at http://publibz.boulder.ibm.com:80/cgi-bin/bookmgr_OS390/BOOKS/ZIDOCMST/CCONTENTS.

Information APARs for IP documents

Table 33 lists information APARs for IP documents.

Table 33. IP information APARs for z/OS Communications Server

Title	V1R6	V1R5	V1R4	V1R2
New Function Summary (both IP and SNA)	II13824			
Quick Reference (both IP and SNA)	II13831		II13246	II12500
IP and SNA Codes	II13842		II13254	II12504
IP API Guide	II13844	II13577	II13255 II13790	II12861 II13655
IP CICS Sockets Guide		II13578	II13257	II12862 II13627
IP Configuration Guide	II13826	II13568	II13244 II13541 II13652 II13646	II12498 II13087 II13364 II13634 II13651
IP Configuration Reference	II13827	II13569 II13789	II13245 II13521 II13647 II13739	II12499 II13394 II13637
IP Diagnosis	II13836	II13571	II13249 II13493	II12503 II13473
IP Messages Volume 1	II13838	II13572	II13624 II13250	II12857 II13229 II13405
IP Messages Volume 2	II13839	II13573	II13251	II12858
IP Messages Volume 3	II13840	II13574	II13252	II12859
IP Messages Volume 4	II13841	II13575	II13253 II13628	II12860

Table 33. IP information APARs for z/OS Communications Server (continued)

Title	V1R6	V1R5	V1R4	V1R2
IP Migration		II13566	II13242 II13738	II12497 II13636
IP Network and Application Design Guide	II13825	II13567	II13243	
IP Network Print Facility				II12864
IP Programmer's Reference	II13843	II13581	II13256	II12505
IP User's Guide and Commands	II13832	II13570	II13247	II12501 II13404
IP System Admin Commands	II13833	II13580	II13248 II13792	II12502 II13793

Information APARs for SNA documents

Table 34 lists information APARs for SNA documents.

Table 34. SNA information APARs for z/OS Communications Server

Title	V1R6	V1R5	V1R4	V1R2
New Function Summary (both IP and SNA)	II13824			
Quick Reference (both IP and SNA)	II13831		II13246	II12500
IP and SNA Codes	II13842		II13254	II12504
SNA Customization	II13857	II13560	II13240	II12872
SNA Diagnosis		II13558	II13236 II13735	II12490 II13034
SNA Diagnosis, Vol. 1: Techniques and Procedures	II13852			
SNA Diagnosis, Vol. 2: FFST Dumps and the VIT	II13853			
SNA Messages	II13854	II13559	II13238 II13736	II12491
SNA Network Implementation Guide	II13849	II13555	II13234 II13733	II13635 II12487
SNA Operation	II13851	II13557	II13237	II12489
SNA Migration		II13554	II13233 II13732	II12486
SNA Programming	II13858		II13241	II13033
SNA Resource Definition Reference	II13850	II13556	II13235 II13734	II12488
SNA Data Areas, Vol. 1 and 2			II13239	II12492
SNA Data Areas, 1	II13855			
SNA Data Areas, 2	II13856			

Other information APARs

Table 35 lists information APARs not related to documents.

Table 35. Non-document information APARs

Content	Number
index of recommended maintenance for VTAM	II11220
index of Communication Server IP information APARs	II12028
AHHC, MPC, and CTC	II01501
Collecting TCPIP CTRACEs	II12014
CSM for VTAM	II12657
CSM for TCP/IP	II12658
DLUR/DLUS for z/OS V1R2	II12986
DOCUMENTATION REQUIRED FOR OSA/2, OSA EXPRESS AND OSA QDIO	II13016
DYNAMIC VIPA (BIND)	II13215
DNS — common problems and solutions	II13453
Enterprise Extender	II12223
FTPing doc to z/OS Support	II12030
FTP problems	II12079
Generic resources	II10986
HPR	II10953
iQDIO	II11220
LPR problems	II12022
MNPS	II10370
NCPROUTE problems	II12025
OMPROUTE	II12026
OROUTED problems	II12024
PASCAL API	II11814
Performance	II11710 II11711 II11712
Resolver	II13398 II13399 II13452
Socket API	II11996 II12020
SMTP problems	II12023
SNMP	II13477 II13478
SYSLOGD howto	II12021
TCPIP connection states	II12449
Telnet	II11574 II13135
TN3270 TELNET SSL common problems	II13369

Appendix H. Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Volume I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

z/OS information

z/OS information is accessible using screen readers with the BookServer/Library Server versions of z/OS books in the Internet library at:

www.ibm.com/servers/eserver/zseries/zos/bkserv/

One exception is command syntax that is published in railroad track format; screen-readable copies of z/OS books with that syntax information are separately available in HTML zipped file form upon request to usib2hpd@vnet.ibm.com.

Notices

IBM may not offer all of the products, services, or features discussed in this document. 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 give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, 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.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites 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:

Site Counsel
IBM Corporation
P.O. Box 12195
3039 Cornwallis Road
Research Triangle Park, North Carolina 27709-2195
U.S.A

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 information 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 measurement 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.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

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 illustrates 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. 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 IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

IBM is required to include the following statements in order to distribute portions of this document and the software described herein to which contributions have been made by The University of California. Portions herein © Copyright 1979, 1980, 1983, 1986, Regents of the University of California. Reproduced by permission. Portions herein were developed at the Electrical Engineering and Computer Sciences Department at the Berkeley campus of the University of California under the auspices of the Regents of the University of California.

Portions of this publication relating to RPC are Copyright © Sun Microsystems, Inc., 1988, 1989.

Some portions of this publication relating to X Window System** are Copyright © 1987, 1988 by Digital Equipment Corporation, Maynard, Massachusetts, and the Massachusetts Institute Of Technology, Cambridge, Massachusetts. All Rights Reserved.

Some portions of this publication relating to X Window System are Copyright © 1986, 1987, 1988 by Hewlett-Packard Corporation.

Permission to use, copy, modify, and distribute the M.I.T., Digital Equipment Corporation, and Hewlett-Packard Corporation portions of this software and its documentation for any purpose without fee is hereby granted, provided that the above copyright notice appears in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the names of M.I.T., Digital, and Hewlett-Packard not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. M.I.T., Digital, and Hewlett-Packard make no representation about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

Copyright © 1983, 1995-1997 Eric P. Allman

Copyright © 1988, 1993 The Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement:
This product includes software developed by the University of California, Berkeley and its contributors.
4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This software program contains code, and/or derivatives or modifications of code originating from the software program "Popper." Popper is Copyright ©1989-1991 The Regents of the University of California, All Rights Reserved. Popper was created by Austin Shelton, Information Systems and Technology, University of California, Berkeley.

Permission from the Regents of the University of California to use, copy, modify, and distribute the "Popper" software contained herein for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies. HOWEVER, ADDITIONAL PERMISSIONS MAY BE NECESSARY FROM OTHER PERSONS OR ENTITIES, TO USE DERIVATIVES OR MODIFICATIONS OF POPPER.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THE POPPER SOFTWARE, OR ITS DERIVATIVES OR MODIFICATIONS, AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE POPPER SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Copyright © 1983 The Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms are permitted provided that the above copyright notice and this paragraph are duplicated in all such forms and that any documentation, advertising materials, and other materials related to such distribution and use acknowledge that the software was developed by the University of California, Berkeley. The name of the University may not be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Copyright © 1991, 1993 The Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement:
This product includes software developed by the University of California, Berkeley and its contributors.
4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright © 1990 by the Massachusetts Institute of Technology

Export of this software from the United States of America may require a specific license from the United States Government. It is the responsibility of any person or organization contemplating export to obtain such a license before exporting.

WITHIN THAT CONSTRAINT, permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of M.I.T. not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. Furthermore

if you modify this software you must label your software as modified software and not distribute it in such a fashion that it might be confused with the original M.I.T. software. M.I.T. makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

Copyright © 1998 by the FundsXpress, INC. All rights reserved.

Export of this software from the United States of America may require a specific license from the United States Government. It is the responsibility of any person or organization contemplating export to obtain such a license before exporting.

WITHIN THAT CONSTRAINT, permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of FundsXpress not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. FundsXpress makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Copyright © 1999, 2000 Internet Software Consortium.

Permission to use, copy, modify, and distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND INTERNET SOFTWARE CONSORTIUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL INTERNET SOFTWARE CONSORTIUM BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Copyright © 1995-1998 Eric Young (eay@cryptsoft.com) All rights reserved.

This package is an SSL implementation written by Eric Young (eay@cryptsoft.com). The implementation was written so as to conform with Netscape's SSL.

This library is free for commercial and non-commercial use as long as the following conditions are adhered to. The following conditions apply to all code found in this distribution, be it the RC4, RSA, lhash, DES, etc., code; not just the SSL code. The SSL documentation included with this distribution is covered by the same copyright terms except that the holder is Tim Hudson (tjh@cryptsoft.com).

Copyright remains Eric Young's, and as such any Copyright notices in the code are not to be removed. If this package is used in a product, Eric Young should be

given attribution as the author of the parts of the library used. This can be in the form of a textual message at program startup or in documentation (online or textual) provided with the package.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement: "This product includes cryptographic software written by Eric Young (eay@cryptsoft.com)". The word 'cryptographic' can be left out if the routines from the library being used are not cryptographic related.
4. If you include any Windows specific code (or a derivative thereof) from the apps directory (application code) you must include acknowledgement:
"This product includes software written by Tim Hudson (tjh@cryptsoft.com)"

THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The license and distribution terms for any publicly available version or derivative of this code cannot be changed. i.e. this code cannot simply be copied and put under another distribution license [including the GNU Public License.]

This product includes cryptographic software written by Eric Young.

Copyright © 1999, 2000 Internet Software Consortium.

Permission to use, copy, modify, and distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND INTERNET SOFTWARE CONSORTIUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL INTERNET SOFTWARE CONSORTIUM BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Copyright © 2004 IBM Corporation and its licensors, including Sendmail, Inc., and the Regents of the University of California. All rights reserved.

Copyright © 1999,2000,2001 Compaq Computer Corporation

Copyright © 1999,2000,2001 Hewlett-Packard Company

Copyright © 1999,2000,2001 IBM Corporation

Copyright © 1999,2000,2001 Hummingbird Communications Ltd.

Copyright © 1999,2000,2001 Silicon Graphics, Inc.

Copyright © 1999,2000,2001 Sun Microsystems, Inc.

Copyright © 1999,2000,2001 The Open Group

All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

X Window System is a trademark of The Open Group.

If you are viewing this information softcopy, photographs and color illustrations may not appear.

You can obtain softcopy from the z/OS Collection (SK3T-4269), which contains BookManager and PDF formats of unlicensed books and the z/OS Licensed Product Library (LK3T-4307), which contains BookManager and PDF formats of licensed books.

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.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

Advanced Peer-to-Peer Networking	MVS/SP
AFP	MVS/XA
AD/Cycle	NetView
AIX	Network Station
AIX/ESA	Nways
AnyNet	Notes
APL2	OfficeVision/MVS
AS/400	OfficeVision/VM
AT	Open Class
BookManager	OpenEdition
BookMaster	OS/2
C/370	OS/390
CICS	OS/400
CICS/ESA	Parallel Sysplex
C/MVS	PR/SM
Common User Access	PROFS
C Set ++	PS/2
CT	RACF
CUA	Redbooks
DB2	Resource Link
DFSMSdfp	RETAIN
DFSMShsm	RISC System/6000
DFSMS/MVS	RMF
DPI	RS/6000
Domino	S/370
DRDA	S/390
Enterprise Systems Architecture/370	S/390 Parallel Enterprise Server
ESCON	SAA
eServer	SecureWay
ES/3090	SP
ES/9000	SP2
ES/9370	SQL/DS
EtherStreamer	System/360
Extended Services	System/370
FFST	System/390
FFST/2	SystemView
First Failure Support Technology	Tivoli
GDDM	TURBOWAYS
IBM	VM/ESA
IBMLink	VSE/ESA
IMS	VTAM
IMS/ESA	WebSphere
Java	XT
HiperSockets	z/Architecture
Language Environment	z/OS
LANStreamer	zSeries
Library Reader	z/VM
LPDA	400
Micro Channel	3090
Multiprise	3890
MVS	
MVS/DFP	
MVS/ESA	

DB2 and NetView are registered trademarks of International Business Machines Corporation or Tivoli Systems Inc. in the U.S., other countries, or both.

The following terms are trademarks of other companies:

ATM is a trademark of Adobe Systems, Incorporated.

BSC is a trademark of BusiSoft Corporation.

CSA is a trademark of Canadian Standards Association.

DCE is a trademark of The Open Software Foundation.

HYPERchannel is a trademark of Network Systems Corporation.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

ActionMedia, LANDesk, MMX, Pentium, and ProShare are trademarks of Intel Corporation in the United States, other countries, or both. For a complete list of Intel trademarks, see <http://www.intel.com/sites/corporate/tradmarx.htm> .

Other company, product, and service names may be trademarks or service marks of others.

Bibliography

z/OS Communications Server information

This section 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/servers/eserver/zseries/zos/bkserv>
- In softcopy on CD-ROM collections. See “Softcopy information” on page xxv.

z/OS Communications Server library

z/OS Communications Server documents are available on the CD-ROM accompanying z/OS (SK3T-4269 or SK3T-4307). Unlicensed documents can be viewed at the z/OS Internet library site.

Updates to documents are available on RETAIN[®] and in information APARs (info APARs). See Appendix G, “Information APARs,” on page 821 for a list of the documents and the info APARs associated with them.

Info APARs for z/OS documents are in the document called *z/OS and z/OS.e DOC APAR and PTF ++HOLD Documentation* which can be found at http://publibz.boulder.ibm.com:80/cgi-bin/bookmgr_OS390/BOOKS/ZIDOCMST/CCONTENTS.

Planning

Title	Number	Description
<i>z/OS Communications Server: New Function Summary</i>	GC31-8771	This document is intended to help you plan for new IP for SNA function, 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.
<i>z/OS Communications Server: IPv6 Network and Application Design Guide</i>	SC31-8885	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
<i>z/OS Communications Server: IP Configuration Guide</i>	SC31-8775	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 in conjunction with the <i>z/OS Communications Server: IP Configuration Reference</i> .

Title	Number	Description
<i>z/OS Communications Server: IP Configuration Reference</i>	SC31-8776	This document presents information for people who want to administer and maintain IP. Use this document in conjunction with the <i>z/OS Communications Server: IP Configuration Guide</i> . The information in this document includes: <ul style="list-style-type: none"> • TCP/IP configuration data sets • Configuration statements • Translation tables • SMF records • Protocol number and port assignments
<i>z/OS Communications Server: SNA Network Implementation Guide</i>	SC31-8777	This document presents the major concepts involved in implementing an SNA network. Use this document in conjunction with the <i>z/OS Communications Server: SNA Resource Definition Reference</i> .
<i>z/OS Communications Server: SNA Resource Definition Reference</i>	SC31-8778	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 in conjunction with the <i>z/OS Communications Server: SNA Network Implementation Guide</i> .
<i>z/OS Communications Server: SNA Resource Definition Samples</i>	SC31-8836	This document contains sample definitions to help you implement SNA functions in your networks, and includes sample major node definitions.
<i>z/OS Communications Server: AnyNet SNA over TCP/IP</i>	SC31-8832	This guide provides information to help you install, configure, use, and diagnose SNA over TCP/IP.
<i>z/OS Communications Server: AnyNet Sockets over SNA</i>	SC31-8831	This guide provides information to help you install, configure, use, and diagnose sockets over SNA. It also provides information to help you prepare application programs to use sockets over SNA.
<i>z/OS Communications Server: IP Network Print Facility</i>	SC31-8833	This document is for system 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
<i>z/OS Communications Server: IP User's Guide and Commands</i>	SC31-8780	This document describes how to use TCP/IP applications. It contains requests that allow a user to log on to a remote host using Telnet, transfer data sets using FTP, send and receive electronic mail, print on remote printers, and authenticate network users.
<i>z/OS Communications Server: IP System Administrator's Commands</i>	SC31-8781	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.
<i>z/OS Communications Server: SNA Operation</i>	SC31-8779	This document serves as a reference for programmers and operators requiring detailed information about specific operator commands.
<i>z/OS Communications Server: Quick Reference</i>	SX75-0124	This document contains essential information about SNA and IP commands.

Customization

Title	Number	Description
<i>z/OS Communications Server: SNA Customization</i>	LY43-0092	<p>This document enables you to customize SNA, and includes the following:</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
<i>z/OS Communications Server: IP Application Programming Interface Guide</i>	SC31-8788	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.
<i>z/OS Communications Server: IP CICS Sockets Guide</i>	SC31-8807	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.
<i>z/OS Communications Server: IP IMS Sockets Guide</i>	SC31-8830	This document is for programmers who want application programs that use the IMS TCP/IP application development services provided by IBM's TCP/IP Services.
<i>z/OS Communications Server: IP Programmer's Reference</i>	SC31-8787	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.
<i>z/OS Communications Server: SNA Programming</i>	SC31-8829	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.
<i>z/OS Communications Server: SNA Programmer's LU 6.2 Guide</i>	SC31-8811	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.)
<i>z/OS Communications Server: SNA Programmer's LU 6.2 Reference</i>	SC31-8810	This document provides reference material for the SNA LU 6.2 programming interface for host application programs.
<i>z/OS Communications Server: CSM Guide</i>	SC31-8808	This document describes how applications use the communications storage manager.

Title	Number	Description
<i>z/OS Communications Server: CMIP Services and Topology Agent Guide</i>	SC31-8828	This document describes the Common Management Information Protocol (CMIP) programming interface for application programmers to use in coding CMIP application programs. The document provides guide and reference information about CMIP services and the SNA topology agent.

Diagnosis

Title	Number	Description
<i>z/OS Communications Server: IP Diagnosis Guide</i>	GC31-8782	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.
<i>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</i>	LY43-0088 LY43-0089	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.
<i>z/OS Communications Server: SNA Data Areas Volume 1 and z/OS Communications Server: SNA Data Areas Volume 2</i>	LY43-0090 LY43-0091	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
<i>z/OS Communications Server: SNA Messages</i>	SC31-8790	This document describes the ELM, IKT, IST, ISU, 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
<i>z/OS Communications Server: IP Messages Volume 1 (EZA)</i>	SC31-8783	This volume contains TCP/IP messages beginning with EZA.
<i>z/OS Communications Server: IP Messages Volume 2 (EZB)</i>	SC31-8784	This volume contains TCP/IP messages beginning with EZB.
<i>z/OS Communications Server: IP Messages Volume 3 (EZY)</i>	SC31-8785	This volume contains TCP/IP messages beginning with EZY.
<i>z/OS Communications Server: IP Messages Volume 4 (EZZ-SNM)</i>	SC31-8786	This volume contains TCP/IP messages beginning with EZZ and SNM.
<i>z/OS Communications Server: IP and SNA Codes</i>	SC31-8791	This document describes codes and other information that appear in z/OS Communications Server messages.

APPC Application Suite

Title	Number	Description
<i>z/OS Communications Server: APPC Application Suite User's Guide</i>	SC31-8809	This documents the end-user interface (concepts, commands, and messages) for the AFTP, ANAME, and APING facilities of the APPC application suite. Although its primary audience is the end user, administrators and application programmers may also find it useful.

Title	Number	Description
<i>z/OS Communications Server: APPC Application Suite Administration</i>	SC31-8835	This document contains the information that administrators need to configure the APPC application suite and to manage the APING, ANAME, AFTP, and A3270 servers.
<i>z/OS Communications Server: APPC Application Suite Programming</i>	SC31-8834	This document provides the information application programmers need to add the functions of the AFTP and ANAME APIs to their application programs.

Index

A

- abbreviations and acronyms 805
- abend U4093, user 794
- abends
 - C program 103
 - errno value dependency 103
 - return values 104
 - RTL functions 104
 - SCEERUN 104
 - uninitialized storage 104
- ACCEPT (call) 433
- ACCEPT (macro) 259
- Accept (REXX) 626
- accept() 108
- accessibility 825
- addr parameter on C socket calls
 - on accept() 108
 - on gethostbyaddr() 126
- address families 9
- address parameter on TCP/UDP/IP (pascal), on
 - gethoststring 728
- address, loopback 22
- addressing sockets in internet domain 9
- addressing within sockets, network byte order 11
- addrlen parameter on C socket calls
 - on accept() 108
 - on gethostbyaddr() 126
- AddUserNote 725
- AF parameter on call interface, on SOCKET 542
- AF parameter on macro interface, on socket 374
- AF_INET 27
- AF_INET address family 9, 12, 797
- AF_IUCV 27
- AF_IUCV address family 797
- ALET parameter on macro socket interface
 - on RECV 336
 - on RECVFROM 340
 - on SEND 353
 - on SENDTO 359
 - on WRITE 380
- allocate, socket call 27
- AmountOfTime parameter on TCP/UDP/IP (Pascal) , on
 - SetTimer 740
- APITYPE parameter on macro interface, INITAPI call 318
- application program, organizing 15
- applications program interface (API)
 - C language API 95
 - Pascal language API 711
 - REXX sockets API 615
- arg parameter on C socket calls
 - on fcntl() 122
 - on ioctl() 161
- arg parameter, on socket built-in function, REXX sockets 618
- Assembler Callable Services, z/OS UNIX, general
 - description 8
- assembler calls 722
- assembler programs, macro instructions 259
- asynchronous communication, (Pascal API) 711
- asynchronous ECB routine 255
- asynchronous exit routine 255
- asynchronous macro, coding example 257

- asynchronous select 50
- AtoETable parameter on TCP/UDP/IP (Pascal), on
 - ReadXlateTable 737
- ATTACH supervisor call instruction 39, 52

B

- backlog parameter on C socket call, listen() 163
- BACKLOG parameter on call interface, LISTEN call 494
- BACKLOG parameter on macro interface, LISTEN call 327
- backlog parameter on REXX interface, Listen call (input) 632
- BeginTcpIp (Pascal) 725
- Berkeley socket implementation 104
- bind () 110
- BIND (call) 436
- BIND (macro) 262
- Bind (REXX) 627
- bit set macros on C socket calls
 - FD_CLR 178
 - FD_ISSET 178
 - FD_SET 178
 - FD_ZERO 178
- bit-mask on call interface, on EZACIC06 call 552
- bit-mask-length on call interface, on EZACIC06 call 553
- buf parameter on C socket calls
 - on read() 168
 - on recv() 171
 - on recvfrom() 173
 - on write() 209
- BUF parameter on call socket interface
 - on GETIBMOPT 463
 - on READ 500
 - on RECV 504
 - on RECVFROM 507
 - on SEND 521
 - on SENDTO 529
 - on WRITE 546
- BUF parameter on macro socket interface
 - on GETIBMOPT 293
 - on RECV 336
 - on RECVFROM 338
 - on SEND 353
 - on SENDTO 359
 - on WRITE 380
- Buffer parameter on TCP/UDP/IP (Pascal) procedure
 - on MonQuery 732
 - on RawIpReceive 735
 - on RawIpSend 736
 - on TcpFReceive, TcpReceive, TcpWaitReceive 742
 - on TcpFSend, TcpSend, TcpWaitSend 745
- bufferaddress parameter on TCP/UDP/IP (Pascal) procedure
 - on UdpNReceive 751
 - on UdpSend 753
- BufferLength parameter on TCP/UDP/IP (Pascal) interface
 - on RawIpReceive 735
 - on TcpFSend, TcpSend, TcpWaitSend 745
 - on UdpNReceive 751
- BUFFERspaceAVAILABLE (Pascal) 718
- bufSize parameter on TCP/UDP/IP (Pascal), on
 - MonQuery 732
- built-in functions, REXX sockets 616

- byte order parameter on C socket calls
 - on htonl() 153
 - on htons() 154
 - on ntohl() 166
 - on ntohs() 167
- byte ordering convention
 - big endian 11
 - little endian 11
- BytesRead parameter on TCP/UDP/IP (Pascal) procedure, on
 - TcpFReceive, TcpReceive, TcpWaitReceive 742
- BytesToRead 715
- BytesToRead parameter on TCP/UDP/IP (Pascal) procedure,
 - on TcpFReceive, TcpReceive, TcpWaitReceive 742

C

- C applications
 - compiling and link-editing non-reentrant modules 96
 - compiling and link-editing reentrant modules 99
- C socket application programming interface 95
- C socket call syntax 106
- C socket calls
 - accept()
 - description 108
 - example 109
 - return values 109
 - use example 108
 - bind()
 - AF_INET domain example 112
 - AF_IUCV domain example 113
 - created in the AF_INET domain 110
 - created in the AF_IUCV domain 111
 - description 110
 - return values 111
 - use example 30
 - close()
 - description 114
 - example 114
 - return values 114
 - connect()
 - description 115
 - Examples 117
 - return values 116
 - Servers, AF_INET domain 115
 - Servers, AF_IUCV domain 116
 - endhostent() 118
 - endnetent() 119
 - endprotoent() 120
 - endservent() 121
 - fcntl()
 - call example 122
 - description 122
 - return values 122
 - getclientid()
 - call example 124
 - description 124
 - return values 124
 - getdtablesize() 125
 - gethostbyaddr()
 - call example 126
 - description 126
 - return values 126
 - gethostbyname()
 - call example 127
 - description 127
 - return values 127

- C socket calls (*continued*)
 - gethostent()
 - call example 128
 - description 129
 - return values 128
 - gethostid()
 - description 129
 - return values 129
 - gethostname()
 - description 130
 - return values 130
 - getibmsockopt()
 - call example 131
 - description 131
 - return values 132
 - getibmssockopt()
 - call example 133
 - description 133
 - return values 133
 - getnetbyaddr()
 - description 134
 - return values 134
 - getnetbyname()
 - description 135
 - return values 135
 - getnetent()
 - description 136
 - return values 136
 - getpeername()
 - description 137
 - return values 137
 - getprotobyname()
 - description 138
 - return values 138
 - getprotobynumber()
 - description 139
 - return values 139
 - getprotoent()
 - description 140
 - return values 140
 - getservbyname()
 - description 141
 - return values 141
 - getservbyport()
 - description 142
 - return values 142
 - getservent()
 - description 143
 - return values 143
 - getsockname()
 - description 144
 - return values 144
 - getsockopt()
 - call example 149
 - description 145
 - options 146, 148, 198
 - return values 149
 - givesocket()
 - description 151
 - return values 152
 - htonl()
 - description 153
 - return values 153
 - htons()
 - description 154
 - return values 154

C socket calls (*continued*)

- inet_addr()
 - description 155
 - return values 155
- inet_lnaof()
 - description 156
 - return values 156
- inet_makeaddr()
 - description 157
 - return values 157
- inet_netof()
 - description 158
 - return values 158
- inet_network()
 - description 159
 - return values 159
- inet_ntoa()
 - description 160
 - return values 160
- ioctl()
 - call example 161, 162
 - command 161
 - description 161
 - return values 162
- listen()
 - call example 163
 - description 108, 163
 - return values 163
- maxdesc()
 - description 164
 - examples 165
 - return values 164
- ntohl()
 - description 166
 - return values 166
- ntohs()
 - description 167
 - return values 167
- read()
 - description 168
 - return values 168
- readv()
 - description 169
 - return values 169
 - use example 69
- recv()
 - call example 171
 - description 171
 - return values 171
 - use example 69
- recvfrom()
 - call example 173
 - description 173
 - return values 174
 - use example 72
- recvmsg()
 - description 175
 - return values 176
- select()
 - bit set macros 178
 - call example 177, 179
 - description 177
 - return values 179
 - use example 50, 108
- selectex()
 - description 181
 - return values 181

C socket calls (*continued*)

- send()
 - call example 183
 - description 69, 183
 - return values 183
 - use example 69
- sendmsg()
 - description 185
 - return values 186
- sendto()
 - call example 187
 - description 187
 - return values 187
 - use example 72
- sethostent()
 - description 189
 - return values 189
- setibmopt()
 - description 190
 - return values 190
 - structure elements 190
- setibmsockopt()
 - call example 192, 193
 - return values 192
- setnetent()
 - description 194
 - return values 194
- setprotoent()
 - description 195
 - return values 195
- setservent()
 - description 196
 - return values 196
- setsockopt()
 - call example 200
 - description 197
 - options 198, 199
 - return values 200
- shutdown()
 - description 201
 - return values 201
- sock_debug() 202
- sock_do_teststor() 203
- socket()
 - call examples 206
 - description 204
 - limitations 206
 - return values 206
- takesocket()
 - description 207
 - return values 207
- tcperror()
 - call example 208
 - description 208
- write()
 - description 209
 - return values 209
 - use example 69
- writenv()
 - description 210
 - return values 210
 - use examples 69

C socket header files

- bsdtypes.h 105, 164, 178
- fcntl.h 105
- if.h 105
- in.h 105

- C socket header files *(continued)*
 - inet.h 105, 110
 - ioctl.h 105
 - manifest.h 105
 - netdb.h 105
 - rtroute.h 105
 - saiucv.h 105, 111
 - socket.h 175, 185
 - uio.h 105, 169, 210
- C socket programming concepts 5
- C sockets, general description 8
- C structures 106
- C/C++ Sockets, general description 8
- CALAREA parameter on CANCEL 266
- CALL Instruction Interface for Assembler, PL/1, and COBOL 429
- Call Instructions for Assembler, PL/1, and COBOL Programs
 - ACCEPT 433
 - BIND 436
 - CLOSE 438
 - CONNECT 440
 - EZACIC04 550
 - EZACIC05 551
 - EZACIC06 552
 - EZACIC08 554
 - FCNTL 443
 - GETCLIENTID 454
 - GETHOSTBYADDR 455
 - GETHOSTBYNAME 458
 - GETHOSTID 460
 - GETHOSTNAME 461
 - GETIBMOPT 462
 - GETPEERNAME 469
 - GETSOCKNAME 471
 - GETSOCKOPT 473
 - GIVESOCKET 482
 - INITAPI 485
 - IOCTL 487
 - LISTEN 493
 - READ 499
 - READV 501
 - RECV 503
 - RECVFROM 505
 - RECVMSG 508
 - SELECT 512
 - SELECTEX 516
 - SENDMSG 522
 - SENDTO 526
 - SETSOCKOPT 530
 - SHUTDOWN 539
 - SOCKET 541
 - TAKESOCKET 543
 - TERMAPI 544
 - WRITE 545
 - WRITEV 547
- Call Instructions for Assembler, PL/I, and COBOL Programs
 - EZACIC14 562
 - EZACIC15 564
- call interface sample PL/I programs 566
- call sequence 17
- call syntax, C sockets 106
- CallReturn parameter on TCP/UDP/IP (Pascal), on SayCalRe 737
- CANCEL (macro) 265
- CBC3022 103
- CBC3050 103
- CBC3221 103
- CBC3282 103
- CBC3296 103
- CBC3343 103
- CBC5034 103
- CHAR-MASK parameter on call interface, on EZACIC06 552
- Character Generator 21
- CICS (customer information control system) sockets
 - general description 8
- CICS, not using tcperror() 95
- Class parameter on TCP/UDP/IP (pascal), on IsLocalHost 731
- ClearTimer 726
- client and server socket programs 15
- CLIENT parameter on call socket interface
 - on GETCLIENTID 454
 - on GIVESOCKET 484
 - on TAKESOCKET 544
- CLIENT parameter on macro socket interface
 - on GETCLIENTID 283
 - on GIVESOCKET 314
 - on TAKESOCKET 377
- client program, designing 57
- client, socket calls, general
 - givesocket() and takesocket() 39
 - send() and recv() 68
- clientid parameter on C socket call
 - on getclientid() 124
 - on givesocket() 151
 - on takesocket() 207
- clientid parameter on REXX socket interface
 - on Getclientid (output) 651
 - on Givesocket (input) 631
 - on Takesocket (input) 636
- CLOSE (macro) 267
- Close (REXX) 629
- cmd parameter on C socket calls
 - on fcntl() 122
 - on inet_addr() 155
 - on inet_network() 159
 - on ioctl() 161
- COMMAND parameter on call interface, IOCTL call 488
- COMMAND parameter on call socket interface
 - on EZACIC06 553
 - on FCNTL 444
 - on GETIBMOPT 463
- COMMAND parameter on macro interface
 - on FCNTL 272
 - on IOCTL 320
- COMMAND parameter on macro socket interface
 - on GETIBMOPT 293
- Communications Server for z/OS, online information xxvi
- compiling and linking, C sockets
 - compiling and link-editing non-reentrant modules 96
 - compiling and link-editing reentrant modules 99
- concepts, TCP/IP 3
- concurrent server program, designing 39
- concurrent server socket programs 17
- CONNECT (macro) 268
- Connect (REXX) 630
- connect() 115
- connectinfo parameter on REXX interface, on Socketsetstatus (output) 623
- Connection (Pascal) 714
- connection information record (Pascal) 714
- Connection parameter on TCP/UDP/IP (Pascal) procedure
 - on TcpAbort 740
 - on TcpClose 741

- Connection parameter on TCP/UDP/IP (Pascal) procedure (continued)
 - on TcpFReceive, TcpReceive, TcpWaitReceive 742
 - on TcpFSend, TcpSend, TcpWaitSend 745
 - on TcpOption 748
- Connection States (Pascal) 713, 715
- CONNECTIONclosing (Pascal) 713
- ConnectionInfo parameter on TCP/UDP/IP (Pascal) procedure
 - on TcpOpen, TcpWaitOpen 746
 - on TcpStatus 749
- CONNECTIONstateCHANGED (Pascal) 718
- ConnIndex parameter on TCP/UDP/IP (Pascal) procedure
 - on UdpClose 750
 - on UdpNReceive 751
 - on UdpOpen 752
 - on UdpReceive 753
 - on UdpSend 753
- count parameter on REXX interface, on Select (output) 679
- CreateTimer 726
- customer information control system sockets, general description, see also CICS 8

D

- data parameter on REXX socket interface
 - on Read (output) 638
 - on Recv (output) 639
 - on Recvfrom (output) 640
 - on Send (input) 642
 - on Sendto (input) 643
 - on Write (input) 645
- Data parameter on TCP/UDP/IP (Pascal), on SetTimer 740
- data sets
 - hlq.AEZAMAC4 722
 - hlq.ETC.PROTO 120, 138, 139, 195
 - hlq.ETC.SERVICES 9, 121, 141, 142, 196
 - hlq.HOSTS.ADDRINFO 118
 - hlq.HOSTS.SITEINFO 118
 - hlq.SEZACMAC 225
 - hlq.SEZACMTX 225
 - hlq.SEZAINST 53, 225
 - hlq.SEZALOAD 225
 - MANIFEST.H 105
 - NETDB.H 126, 127, 128, 134, 142
 - TCPIP.DATA 203
 - user_id.TCPIP.DATA 728, 746
- data sets for TCP/IP programming libraries 22
- data structures, Pascal 713
- data transfer between sockets 63
- data transfer, sockets 6
- data translation, socket interface 548
 - ASCII to EBCDIC 551
 - bit-mask to character 552
 - character to bit-mask 552
 - EBCDIC to ASCII 550, 562
- DATAdelivered (Pascal) 718
- datagram sockets 6
- datagram sockets, program design 61
- DatagramAddress parameter on TCP/UDP/IP (Pascal), on UdpReceive 753
- DataLength parameter on TCP/UDP/IP (Pascal), on RawIpSend 736
- date parameter on REXX socket interface, on version (output) 689
- debug and measurement tools
 - Character Generator 21
 - Discard 21

- debug and measurement tools (continued)
 - Echo 21
- designing an iterative server program 27
- DestroyTimer 726
- disability 825
- Discard 21
- DNS, online information xxvii
- documents, licensed xxvii
- domain name system (DNS) 746
- domain parameter on C socket calls
 - on getclientid() 124
 - on gethostbyaddr() 126
 - on socket() 204
- domain parameter on REXX socket call
 - on Getclientid (input) 651, 653
 - on Socket (input) 634
- domainname parameter on REXX socket call, on Getdomainname (output) 652
- DomainName parameter on TCP/UDP/IP (Pascal) , on GetIdentity 729
- dotted decimal notation 159

E

- ECB parameter on EZASMI 50
- ECB parameter on macro interface
 - on ACCEPT 261, 265, 266, 268, 271, 273, 283, 289, 291, 300, 303, 305, 315, 325, 327, 332, 334, 337, 340, 344, 348, 354, 357, 361, 363, 372, 375, 378, 381, 383
 - on GETHOSTBYNAME 286
- ECBPTR parameter on C socket call, shutdown () 181
- Echo 21
- endhostent() 118
- endnetent() 119
- endprotoent() 120
- endservent() 121
- EndTcpIp (Pascal) 726
- ERETMSK parameter on call interface, on SELECT 516
- ERETMSK parameter on macro interface, on SELECT 348
- ERRNO parameter on call socket interface
 - on ACCEPT 435
 - on BIND 438
 - on CLOSE 440
 - on CONNECT 443
 - on FCNTL 444
 - on GETCLIENTID 455
 - on GETHOSTNMAE 462
 - on GETIBMOPT 464
 - on GETPEERNAME 471
 - on GETSOCKNAME 473
 - on GETSOCKOPT 474
 - on GIVESOCKET 485
 - on INITAPI 487
 - on IOCTL 493
 - on LISTEN 494
 - on READ 500
 - on READV 502
 - on RECV 504
 - on RECVFROM 508
 - on RECVMSG 512
 - on SELECT 516
 - on SELECTEX 519
 - on SEND 522
 - on SENDMSG 526
 - on SENDTO 529
 - on SETSOCKOPT 531
 - on SHUTDOWN 540

ERRNO parameter on call socket interface (*continued*)

- on SOCKET 542
- on TAKESOCKET 544
- on WRITE 546
- on WRITEV 548

ERRNO parameter on macro socket interface

- on ACCEPT 261
- on BIND 264
- on CANCEL 266
- on CLOSE 268
- on CONNECT 271
- on FCNTL 273, 275, 281, 445, 453
- on GETIBMOPT 294
- on GETSOCKOPT 304, 363
- on GETSPCKNAME 302
- on GIVESOCKET 315
- on GRTCLIENTID 283
- on GRTHOSTNAME 291
- on GRTPEERNAME 300
- on INITAPI 318
- on IOCTL 325
- on LISTEN 327
- on READV 334
- on RECV 336
- on RECVFROM 340
- on RECVMSG 343
- on SELECT 347
- on SELECTEX 350
- on SEND 353
- on SENDMSG 357
- on SENDTO 360
- on SHUTDOWN 372
- on SOCKET 374
- on TAKESOCKET 377
- on WRITE 381
- on WRITEV 383

errno values, code dependency 103

errno values, printing 104

ERRNO.H message file 104

ERROR parameter on macro interface

- on ACCEPT 262
- on BIND 265
- on CLOSE 268
- on CONNECT 271
- on FCNTL 274, 275, 282, 298
- on GETCLIENTID 284
- on GETHOSTBYADDR 285
- on GETHOSTBYNAME 288
- on GETHOSTID 290
- on GETHOSTNAME 292
- on GETPEERNAME 300
- on GETSOCKNAME 303
- on GETSOCKOPT 305, 363
- on GIVESOCKET 315
- on INITAPI 319
- on IOCTL 325
- on LISTEN 327
- on RECV 337
- on RECVFROM 340
- on SELECT 349
- on SEND 354
- on SENDTO 361
- on SHUTDOWN 373
- on SOCKET 375
- on TAKESOCKET 378
- on WRITE 381

ERROR parameter on macro socket interface

- on CANCEL 266
- on GETIBMOPT 294
- on RECVMSG 344
- on SELECTEX 351
- on SENDMSG 358
- on WRITEV 383

ESDNMASK parameter on call interface, on SELECT 515

ESNDMSK parameter on macro interface, on SELECT 348

EtoATable parameter on TCP/UDP/IP (Pascal), on

ReadXlateTable 737

EWouldBLOCK error return, call interface calls

- RECV 503
- RECVFROM 505

EWouldBLOCK error return, macro interface calls 273, 337, 381

EWouldBLOCK error return, REXX interface calls 615, 626, 679

exceptfds parameter on C socket calls

- on select() 177
- on selectex() 181

EZACIC04, call interface, EBCDIC to ASCII translation 550

EZACIC05, call interface, ASCII to EBCDIC translation 551

EZACIC06 47

EZACIC06, call interface, bit-mask translation 552

EZACIC08, HOSTENT structure interpreter utility 554

EZACIC09, RES structure interpreter utility 557

EZACIC14, call interface, EBCDIC to ASCII translation 562

EZACIC15, call interface, ASCII to EBCDIC translation 564

F

fcmd parameter on REXX socket call, on FCNTL

F_GETFL (input) 667

F_SETFL (input) 667

FCNTL (call) 443

Fcntl (REXX) 667

fcntl() 122

FD_SETSIZE on send() 178

file specification record (Pascal) 720

FLAGS parameter on call socket interface

- on RECV 504
- on RECVFROM 506
- on RECVMSG 511
- on SEND 521
- on SENDMSG 526
- on SENDTO 528

FLAGS parameter on macro socket interface

- on RECV 336
- on RECVFROM 340
- on RECVMSG 343
- on SEND 353
- on SENDMSG 357
- on SENDTO 361

flags, parameter on C socket calls

- on recv() 171
- on recvfrom() 173
- on recvmsg() 175
- on send() 183
- on sendmsg() 185
- on sendto() 187

FNDELAY flag on call interface, on FCNTL 444

ForeignAddress parameter on TCP/UDP/IP (Pascal), on

PingRequest 733

ForeignSocket 715

ForeignSocket parameter on TCP/UDP/IP (Pascal), on

UdpSend 753

FSENDresponse (Pascal) 719
 fullhostname parameter on REXX socket call
 on Gethostbyaddr (output) 653
 on Gethostbyname (input) 654
 on Resolve (input) 665
 on Resolve (output) 665
 fvalue parameter on REXX socket call
 on Fcntl (input) 667
 on Fcntl (output) 667

G

GETCLIENTID (call) 454
 GETCLIENTID (macro) 282
 Getclientid (REXX) 651
 getclientid() 124
 Getdomainname (REXX) 652
 getdtablesize() 125
 GETHOSTBYADDR (call) 455
 Gethostbyaddr (REXX) 653
 gethostbyaddr() 126
 GETHOSTBYNAME (call) 458
 GETHOSTBYNAME (macro) 127, 286
 Gethostbyname (REXX) 654
 gethostent() 128
 GETHOSTID (call) 460
 GETHOSTID (macro) 289
 Gethostid (REXX) 655
 gethostid() 129
 GETHOSTNAME (call) 461
 GETHOSTNAME (IUCV) 656
 GETHOSTNAME (macro) 290
 gethostname() 130
 GetHostNumber 727
 GetHostResol 727
 GetHostString 728
 GetIBMOpt 131
 GETIBMOPT (call) 462
 GETIBMOPT (macro) 292
 GetIBMSockopt 133
 GetIdentity 728
 getnetbyaddr() 134
 getnetbyname() 135
 getnetent() 136
 GetNextNote 729
 GETPEERNAME (call) 469
 GETPEERNAME (macro) 298
 getpeername() 137
 Getprotobyname (REXX) 660
 getprotobyname() 138
 Getprotobynumber (REXX) 661
 getprotobynumber() 139
 getprotoent() 140
 getservbyname() 141
 Getservbyname (REXX) 662
 getservbyport() 142
 Getservbyport (REXX) 663
 getservent() 143
 GetSmsg 730
 GETSOCKNAME (call) 471
 GETSOCKNAME (macro) 300
 Getsockname (REXX) 664
 getsockname() 144
 GETSOCKOPT (call) 473
 GETSOCKOPT (macro) 303
 Getsockopt (REXX) 668
 getsockopt() 145

GIVESOCKET (call) 482
 GIVESOCKET (macro) 313
 Givesocket (REXX) 631, 659
 givesocket() 151
 GLOBAL (macro) 315
 guidelines for using socket types 6

H

Handle (Pascal) 730
 header files
 C sockets
 in.h 12, 108, 110
 saiucv.h 111, 116
 general, tcperrno.h 208
 Pascal 712, 713
 hints and tips, programming, REXX sockets 615
 HiperSockets Accelerator 146
 hisdesc parameter on C socket call, takesocket() 207
 host lookup routines 722
 HOSTADDR parameter on call interface, on
 GETHOSTBYADDR 456
 HostAddress parameter on TCP/UDP/IP (Pascal), on
 Handle 731
 HOSTADR parameter on macro socket interface, on
 GETHOSTBYADDR 285
 HOSTENT parameter on call socket interface
 on GETHOSTBYADDR 456
 on GETHOSTBYNAME 459
 HOSTENT parameter on macro socket interface
 on GETHOSTBYADDR 285
 on GETHOSTBYNAME 287
 HOSTENT structure interpreter parameters, on
 EZACIC08 555
 hostname parameter on REXX socket
 on Gethostbyname (input) 654, 665
 on Gethostname (output) 656
 hostname parameter on TCP/UDP/IP (Pascal), on
 GetIdentity 729
 hostnumber parameter on TCP/UDP/IP (Pascal)
 on GetHostNumber 727
 on GetHostResol 728, 729
 how parameter on C socket call, on shutdown() 201
 HOW parameter on call interface, on SHUTDOWN 540
 HOW parameter on macro interface, on SHUTDOWN 372
 how parameter on REXX interface, on Shutdown (input) 633
 htonl() 153
 htons() 154

I

IBM Software Support Center, contacting xx
 icmd parameter on REXX interface, ioctl call (input)
 FIONBIO (input) 677
 FIONREAD (input) 677
 SIOCATMARK (input) 677
 SIOCGIFADDR (input) 677
 SIOCGIFCONF (input) 677
 SIOCGIFDSTADDR (input) 677
 SIOCGIFNETMASK (input) 677
 SIOCGIFBRDADDR (input) 677
 icmd parameter on REXX interface, on ioctl (input) 677
 IDENT parameter on call interface, INITAPI call 486
 IDENT parameter on macro interface, INITAPI call 318
 IMS (information management system) sockets
 general description 7

- in parameter on C socket calls
 - on `inet_lnaof()` 156
 - on `inet_netof()` 158
 - on `inet_ntoa()` 160
- IN-BUFFER parameter on call interface, EZACIC05 call 551
- `inet_addr()` 155
- `inet_lnaof()` 156
- `inet_makeaddr()` 157
- `inet_netof()` 158
- `inet_network()` 159
- `inet_ntoa()` 160
- `inetdesc` parameter on `maxdesc()`, C socket call 164
- information APARs for IP-related documents 821
- information APARs for non- document information 823
- information APARs for SNA-related documents 822
- information management system socket interface, general
 - description, see also IMS 7
- `INITAPI(call)` 485
- `INITAPI(macro)` 316
- initialization procedures, TCP/UDP/IP (Pascal) 721
- initialization, REXX sockets 615
- Initialize, REXX sockets 620
- interface, C socket 5
- internet control message protocol (ICMP) 6
- internet domain, addressing sockets 9
- Internet, finding z/OS information online xxvi
- `InternetAddress` parameter on TCP/UDP/IP (Pascal)
 - procedure
 - on `SayInAd` 738
 - on `SayIntNum` 738
- internetwork, protocol layer 5
- `IOCTL (call)` 487
- `IOCTL (macro)` 319
- `Ioctl (REXX)` 677
- `ioctl()` 161
- `iov` parameter on C socket calls
 - on `readv()` 169
 - on `writenv()` 210
- `IOV` parameter on call socket interface
 - on `READV` 502
 - on `WRITEV` 548
- `IOV` parameter on macro socket interface
 - on `RECVMSG` 343
 - on `SENDMSG` 356
 - on `WRITEV` 382
- `iovcnt` parameter on C socket calls
 - on `readv()` 169
 - on `writenv()` 210
- `IOVCNT` parameter on call socket interface
 - on `READV` 502
 - on `RECVMSG` 511
 - on `SENDMSG` 526
 - on `WRITEV` 548
- `IOVCNT` parameter on macro socket interface
 - on `READV` 334
 - on `RECVMSG` 343
 - on `SENDMSG` 357
 - on `WRITEV` 383
- `ipaddress` parameter on REXX socket calls
 - on `Gethostbyaddr (input)` 653
 - on `Gethostid (output)` 655
 - on `Resolve (input)` 665
 - on `Resolve (output)` 665
- `ipaddresslist` parameter on REXX socket call, on
 - `Gethostbyname (output)` 654
- IPv6 programs 73
- `iQDIO` 146

- `IsLocalAddress` 730
- `IsLocalHost` 731
- iterative server socket programs 16
- `ivalue` parameter on REXX socket calls
 - on `Ioctl (input)` 678
 - on `Ioctl (output)` 678

J

JCL

- non-reentrant modules
 - compiling 97
 - linking 98
 - running 99
- reentrant modules
 - compiling 100
 - prelinking and linking 101
 - running 103

K

- keyboard 825

L

- language syntax, C socket call 106
- `len` parameter on C socket calls
 - on `read()` 168
 - on `recv()` 171
 - on `recvfrom()` 173
 - on `send()` 183
 - on `sendto()` 187
 - on `write()` 209
- `LENGTH` parameter on call socket interface
 - on `EZACIC04` 550
 - on `EZACIC05` 551
 - on `EZACIC14` 562
 - on `EZACIC15` 564
- `length` parameter on REXX socket interface
 - on `Read (output)` 638
 - on `Recv (output)` 639
 - on `Recvfrom (output)` 640
 - on `Send (output)` 642
 - on `Sendto (output)` 644
 - on `Write (output)` 645
- `length` parameter on TCP/UDP/IP (Pascal) procedure
 - on `MonQuery` 732
 - on `PingRequest` 733
 - on `UdpSend` 753
- `level` parameter on C socket calls
 - on `getibmssockopt()` 133
 - on `getsockopt()` 145
 - on `setibmssockopt()` 192
 - on `setsockopt()` 197
- `level` parameter on REXX socket calls
 - on `Getsockopt (input)` 668
 - on `Setsockopt (input)` 680
- libraries
 - Data Set 22
 - sockets and pascal API 105
- license, patent, and copyright information 827
- licensed documents xxvii
- `listen ()` 163
- `LISTEN (call)` 493
- `LISTEN (macro)` 325
- `Listen (REXX)` 632

- Listening (Pascal) 713
- lna parameter on inet_makeaddr(), C socket call 157
- LocalSocket 715
- LocalSocket parameter on TCP/UDP/IP (Pascal), on
 UdpOpen 752
- loopback, test address 22

M

- macro instruction interface for assembler programs 249
- macro instructions for assembler programs
 - ACCEPT 259
 - BIND 262
 - CANCEL 265
 - CLOSE 267
 - CONNECT 268
 - GETCLIENTID 282
 - GETDHOSTBYNAME 286
 - GETHOSTID 289
 - GETHOSTNAME 290
 - GETIBMOPT 292
 - GETPEERNAME 298
 - GETSOCKNAME 300
 - GETSOCKOPT 303
 - GIVESOCKET 313
 - GLOBAL 315
 - INITAPI 316
 - IOCTL 319
 - LISTEN 325
 - READV 333
 - RECV 334
 - RECVFROM 337
 - RECVMSG 341
 - SELECT 344
 - SELECTEX 349
 - SEND 352
 - SENDMSG 354
 - SENDTO 358
 - SETSOCKOPT 361
 - SHUTDOWN 371
 - SOCKET 373
 - TAKESOCKET 376
 - TASK 378
 - TERMAPI 379
 - WRITE 379
 - WRITEV 381
- macro interface sample Assembler language programs 383
- MANIFEST.H header file 105
- maxdesc parameter on REXX socket interface
 - on Initialize (input) 620
 - on Initialize (output) 620
- maxdesc() 164
- maximum number of sockets 11
- maxlength parameter on REXX socket interface
 - on Read (input) 638
 - on Recv (input) 639
 - on Recvfrom (input) 640
- MAXSNO parameter on call interface, INITAPI call 486
- MAXSNO parameter on macro interface, INITAPI call 318
- MAXSOC parameter on call socket interface
 - on INITAPI 486
 - on SELECT 515
 - on SELECTEX 519
- MAXSOC parameter on macro socket interface
 - on INITAPI 317
 - on SELECT 347
 - on SELECTEX 350

- MISC SERV (miscellaneous server) 21
- monitor procedures 722
- MonQuery 732
- Motorola-style byte ordering 11
- msg parameter on C socket calls
 - on recvmsg() 175
 - on send() 183
 - on sendmsg() 185
 - on sendto() 187
- MSG parameter on call socket interface
 - on RECVMSG 510
 - on SENDMSG 524
- MSG parameter on macro call interface
 - on RECVMSG 341, 355
- multicast programs 75

N

- name parameter on C socket calls
 - on bind() 110
 - on connect() 115
 - on gethostbyname() 127
 - on gethostname() 130
 - on getnetbyname() 135
 - on getpeername() 137
 - on getprotobyname() 138
 - on getservbyname() 141
 - on getsockname() 144
 - on recvfrom() 174
- NAME parameter on call socket interface
 - on ACCEPT 435
 - on BIND 437
 - on CONNECT 442
 - on GETHOSTBYNAME 458
 - on GETHOSTNAME 461
 - on GETPEERNAME 470
 - on GETSOCKNAME 472
 - on RECVFROM 507
- NAME parameter on macro interface
 - on ACCEPT 260
 - on BIND 263
 - on CONNECT 270
 - on GETHOSTBYNAME 287
 - on GETHOSTNAME 291
 - on GETPEERNAME 299
 - on GETSOCKNAME 301
 - on RECVFROM 338
 - on SENDTO 359
- name parameter on REXX socket calls
 - on Accept (output) 626
 - on Bind (input) 627
 - on Connect (input) 630
 - on Getsockname (output) 664
 - on Recvfrom (output) 640
 - on Sendto (input) 643
- Name parameter on TCP/UDP/IP (Pascal) procedure
 - on GetHostNumber 727
 - on GetHostResol 728
 - on GetHostString 728
 - on IsLocalHost 731
- name resolution, REXX sockets 646
- namelen parameter on C socket calls
 - on bind() 110
 - on connect() 115
 - on gethostname() 130
 - on getpeername() 137
 - on recvfrom() 174

- NAMELEN parameter on call socket interface
 - on GETHOSTBYNAME 458
 - on GETHOSTNAME 461
- NAMELEN parameter on macro socket interface
 - on GETHOSTBYNAME 287
 - on GETHOSTNAME 291
- NBYTE parameter on call socket interface
 - on READ 500
 - on RECV 504
 - on RECVFROM 507
 - on SEND 521
 - on SENDTO 529
 - on WRITE 546
- NBYTE parameter on macro socket interface
 - on RECV 336
 - on RECVFROM 338
 - on SEND 353
 - on SENDTO 359
 - on WRITE 380
- net parameter on C socket call
 - on getnetbyaddr() 134
 - on inet_makeaddr() 157
- NETSTAT command 714
- network concentrator function 146
- NewNameOfTCP parameter on TCP/UDP/IP (Pascal), on
 - TcpNameChange 746
- newsocketid parameter on REXX socket call
 - on Accept (output) 626
 - on Socket (output) 635
 - on Takesocket (output) 636
- nfds parameter on C socket calls
 - on select() 177
 - on selectex() 181
- non-reentrant modules, compiling and link-editing 96
- NONEXISTENT (Pascal) 713
- Note parameter on TCP/UDP/IP (Pascal), on
 - GetNextNote 729
- Notification parameter on TCP/UDP/IP (Pascal), on
 - SayNoeEn 739
- notification record (Pascal) 715
- Notifications parameter on TCP/UDP/IP (Pascal)
 - on Handle 730
 - on Unhandle 754
- NotificationTag (Pascal) 717
- NS parameter on macro interface
 - on ACCEPT 261
 - on SOCKET 375
 - on TAKESOCKET 377
- ntohl() 166
- ntohs() 167
- number of sockets, maximum 11
- NumPackets parameter on TCP/UDP/IP (Pascal), on
 - RawIpSend 736

O

- obey file 6
- onoff parameter on C socket calls
 - on sock_debug() 202
 - on sock_do_teststor() 203
- OPEN (Pascal) 713
- OpenAttemptTimeout 714
- OptionName parameter on TCP/UDP/IP (Pascal), on
 - TcpOption 748
- options, getsockopt(), C socket call 146, 148, 198
- options, REXX sockets calls 666

- OptionValue parameter on TCP/UDP/IP (Pascal), on
 - TcpOption 749
- optlen parameter on C socket calls
 - on getibmssockopt() 133
 - on getsockopt() 145
 - on setibmssockopt() 192
 - on setsockopt() 197
- optname parameter on C socket calls
 - on getibmssockopt() 133
 - on getsockopt() 145
 - on setibmssockopt() 192
 - on setsockopt() 197
- optval parameter on C socket calls
 - on getibmssockopt() 133
 - on getsockopt() 145
 - on setibmssockopt() 192
 - on setsockopt() 197
- optvalue parameter on REXX socket interface
 - on Getsockopt (output) 668
- organizing TCP/IP application program 15
- OS/390 UNIX return codes 790
- OUT-BUFFER parameter on call interface, on EZACIC04 550
- OUT-BUFFER parameter on call interface, on EZACIC14 562
- OUT-BUFFER parameter on call interface, on EZACIC15 564

P

- parameters common, macro interface
 - (reg) 253
 - *indaddr 253
 - 'value' 253
 - address 253
- Pascal
 - assembler calls 722
 - asynchronous communication 711
 - Compiler, IBM VS Pascal and Library 712
 - connection information record 714
 - connection state type 713
 - data structures 713
 - file specification record 720
 - include files
 - tcperrno.h 104
 - notification record 715
 - notifications 721
 - procedure call usage 721
 - return codes 723
 - software requirements 712
 - TCP/UDP/IP 711
- Pascal procedure calls
 - AddUserNote 725
 - BeginTcpIp 725
 - ClearTimer 726
 - CreateTimer 726
 - DestroyTimer 726
 - EndTcpIp 726
 - GetHostNumber 727
 - GetHostResol 727
 - GetHostString 728
 - GetIdentity 728
 - GetNextNote 729
 - GetSmsg 730
 - Handle 730
 - IsLocalAddress 730
 - IsLocalHost 731
 - MonQuery 732
 - PingRequest 733
 - RawIpClose 733

Pascal procedure calls (*continued*)

- RawIpOpen 734
- RawIpReceive 735
- RawIpSend 735
- ReadXlateTable 736
- SayCalRe 737
- SayConSt 737
- SayIntAd 738
- SayIntNum 738
- SayNotEn 739
- SayPorTy 739
- SayProTy 739
- SetTimer 740
- TcpAbort 740
- TcpClose 741
- TcpFReceive, TcpReceive, TcpWaitReceive 741
- TcpFSend, TcpSend, TcpWaitSend 743
- TcpNameChange 746
- TcpOpen, TcpWaitOpen 746
- TcpOption 748
- TcpStatus 749
- UdpClose 750
- UdpNReceive 750
- UdpOpen 751
- UdpReceive 752
- UdpSend 753
- Unhandle 754

Pascal sockets 7

pending activity 46

pending exception 50

pending read 50

performance 6

pererr(), UNIX function 208

PING 722

PingRequest 733

PINGresponse (Pascal) 719

PL/I programs, required statement 432

port parameter on getservbyport – C socket call 142

Port parameter on TCP/UDP/IP (Pascal), on SayPorTy 739

portid parameter on REXX socket call

- on Getservbyname (output) 662
- on Getservbyport (input) 663
- on Getservbyport (output) 663

ports, well known 9

POSIX standard

- using z/OS UNIX C sockets API with 8

program variable definitions, call interface

- assembler definition 432
- COBOL PIC 432
- PL/I declare 432
- VS COBOL II PIC 432

programming with sockets 5

programs

- IPv6 73
- multicast 75

programs, client and server 15

proto parameter on C socket calls

- on getprotobyname() 139
- on getservbyname() 141
- on getservbyport() 142

PROTO parameter on call interface, on SOCKET 542

PROTO parameter on macro interface, on SOCKET 375

Protocol (Pascal) 717

protocol parameter on C socket call, on socket() 204

protocol parameter on REXX socket call, on SOCKET (input) 634

Protocol parameter on TCP/UDP/IP (Pascal), on SayProTy 739

protocolname parameter on REXX socket call

- on Getprotobyname (input) 660
- on Getprotobyname (output) 661
- on Getservbyname (input) 662
- on Getservbyname (output) 662
- on Getservbyport (input) 663
- on Getservbyport (output) 663

ProtocolNo parameter on TCP/UDP/IP (Pascal) procedure

- on RawIpClose 734
- on RawIpOpen 734
- on RawIpReceive 735
- on RawIpSend 736

protocolnumber parameter on REXX socket call

- on Getprotobyname (output) 660
- on Getprotobyname (input) 661

prototyping 105

PushFlag parameter on TCP/UDP/IP (Pascal), on TcpFSend, TcpSend, TcpWaitSend 745

Q

QueryRecord parameter on TCP/UDP/IP (Pascal), on MonQuery 732

R

Raw Ip Interface 722

raw sockets 6

RawIpClose (Pascal) 733

RawIpOpen (Pascal) 734

RAWIPpacketsDELIVERED (Pascal) 719

RawIpReceive (Pascal) 735

RawIpSend (Pascal) 735

RAWIPspaceAVAILABLE (Pascal) 719

READ (call) 499

READ (macro) 330

read() 168

readfds parameter on C socket calls

on select() 177

on selectex() 181

READV (call) 501

READV (macro) 333

readv() 169

ReadXlateTable 736

RECEIVINGonly (Pascal) 713

RECV (call) 503

RECV (macro) 334

recv() 171

recvflags parameter on REXX socket calls

on Recv (input) 639

on Recvfrom (input) 640

RECVFROM (call) 505

RECVFROM (macro) 337

recvfrom() 173

RECVMMSG (call) 508

RECVMMSG (macro) 341

recvmsg() 175

reentrant modules, compiling and link-editing 99

REQARG and RETARG parameter on call socket interface

on FCNTL 444

on IOCTL 492

REQARG parameter on macro socket interface

on FCNTL 273

on IOCTL 324

RESOLVE_VIA_LOOKUP, on C socket call 118, 127, 128
 RESOURCESavailable (Pascal) 719
 Result parameter on TCP/UDP/IP (Pascal), on
 GetIdentity 729
 RETARG parameter on call interface, on IOCTL 493
 RETARG parameter on macro interface, IOCTL call 324
 RETCODE parameter on call socket interface
 on ACCEPT 435
 on BIND 438
 on CLOSE 440
 on CONNECT 443
 on EZACIO6 553
 on FCNTL 444
 on GETCLIENTID 455
 on GETHOSTBYADDR 456
 on GETHOSTBYNAME 459
 on GETHOSTID 460
 on GETHOSTNAME 462
 on GETIBMOPT 464
 on GETPEERNAME 471
 on GETSOCKNAME 473
 on GETSOCKOPT 475
 on GIVESOCKET 485
 on INITAPI 487
 on IOCTL 493
 on LISTEN 495
 on READ 500
 on READV 503
 on RECV 504
 on RECVFROM 508
 on RECVMSG 512
 on SELECT 516
 on SELECTEX 519
 on SEND 522
 on SENDMSG 526
 on SENDTO 529
 on SETSOCKOPT 531
 on SHUTDOWN 541
 on SOCKET 542
 on TAKESOCKET 544
 on WRITE 546
 on WRITEV 548
 RETCODE parameter on macro socket interface
 on ACCEPT 261
 on CANCEL 266
 on CLOSE 268
 on CONNECT 271
 on FCNTL 273, 275, 282, 446, 453
 on GETCLIENTID 283
 on GETHOSTBYADDR 285
 on GETHOSTID 289
 on GETHOSTNAME 291
 on GETIBMOPT 294
 on GETPEERNAME 300
 on GETSOCKNAME 302
 on GETSOCKOPT 305, 363
 on GIVESOCKET 315
 on GRTHOSTBYNAME 287
 on INITAPI 318
 on IOCTL 325
 on LISTEN 327
 on READV 334
 on RECV 336
 on RECVFROM 340
 on RECVMSG 343
 on SELECT 347
 on SELECTEX 350

RETCODE parameter on macro socket interface (*continued*)
 on SEND 353
 on SENDMSG 357
 on SENDTO 360
 on SHUTDOWN 372
 on TAKESOCKET 377
 on WRITE 381
 on WRITEV 383
 return codes
 C sockets 107, 781
 call interface 433
 macro and call interface 791
 macro interface 253
 REXX sockets 619
 socket 781
 return values, code dependency 104
 ReturnCode parameter on TCP/UDP/IP (Pascal) procedure
 on AddUserNote 725
 on BeginTcpip 725
 on GetNextNote 729
 on Handle 730
 on IsLocalAddress 731
 on MonQuery 732
 on Pingrequest 733
 on RawIpClose 734
 on RawIpOpen 734
 on RawIpReceive 735
 on RawIpSend 736
 on ReadXlateTable 737
 on TcpAbort 740
 on TcpClose 741
 on TcpFReceive, TcpReceive, TcpWaitReceive 743
 on TcpFSend, TcpSend, TcpWaitSend 745
 on TcpOpen, TcpWaitOpen 748
 on TcpOption 749
 on TcpStatus 750
 on UdpClose 750
 on UdpNReceive 751
 on UdpOpen 752
 on UdpReceive 753
 on UdpSend 753
 on Unhandle 754
 REXX socket interface 616
 REXX sockets
 general description 8
 initialization 615
 programming hints and tips 615
 return codes 619
 SOCKET, in step library 615
 transtation tables 616
 REXX sockets built-in function, Rxsocket 616, 618
 REXX sockets, calls to process socket sets
 Initialize 620
 Socketset 622
 Socketsetlist 621
 Socketsetstatus 623
 REXX sockets, data exchange
 Read 638
 Recv 639
 Recvfrom 640
 Send 642
 Sendto 643
 Write 645
 REXX sockets, initialize, close, and change
 Accept 626
 Bind 627
 Close 629

REXX sockets, initialize, close, and change (*continued*)

- Connect 630
- Givesocket 631, 659
- Listen 632
- Shutdown 633
- Socket 634
- Takesocket 636

REXX sockets, name resolution

- Getclientid 651
- Getdomainname 652
- Gethostbyaddr 653
- Gethostbyname 654
- Gethostid 655
- Gethostname 656
- Getprotobyname 660
- Getprotobynumber 661
- Getservbyname 662
- Getservbyport 663
- Resolve 665

REXX sockets, options, configuration, mode

- Fcntl 667
- Getsockname 664
- Getsockopt 668
- Ioctl 677
- Select, read, write, exception 679
- Setsockopt 680
- Version 689

REXX sockets, sample programs

- RSCIENT, client sample 690
- RSSERVER, server sample 694

RFC (request for comment)

- list of 811

RFC (request for comments)

- accessing online xxvi

RRETMASK parameter on call interface, on SELECT 516

RRETMASK parameter on macro interface, on SELECT 348

RSNDMSK parameter on call interface, on SELECT 515

RSNDMSK parameter on macro interface, on SELECT 347

RTL functions

- and return codes 104
- built-in 104

S

S, defines socket descriptor on C socket call

- on accept() 108
- on bind() 110
- on close() 114
- on connect() 115
- on fcntl() 122
- on getibmssockopt() 133
- on getpeername() 137
- on getsockname() 141
- on getsockopt() 145
- on ioctl() 161
- on listen() 163
- on read() 168
- on readv() 169
- on recv() 171
- on recvfrom() 173
- on recvmsg() 175
- on send() 183
- on sendmsg() 185
- on sendto() 187
- on setibmssockopt() 192
- on setsockopt() 197
- on shutdown() 201

S, defines socket descriptor on C socket call (*continued*)

- on tcperror() 208
- on write() 209
- on writev() 210

S, defines socket descriptor on macro interface

- on ACCEPT 260
- on BIND 263
- on CLOSE 267
- on CONNECT 270
- on FCNTL 272, 274, 276, 295, 445, 447, 448, 466, 647
- on GETPEERNAME 299
- on GETSOCKNAME 301
- on GETSOCKOPT 304, 363
- on GIVESOCKET 314
- on IOCTL 320
- on LISTEN 327
- on READV 333
- on RECV 336
- on RECVFROM 338
- on RECVMSG 341, 355
- on SEND 353
- on SENDTO 359
- on SHUTDOWN 372
- on WRITE 380
- on WRITEV 382

S, defines socket descriptor on socket call interface

- on ACCEPT 435
- on BIND 437
- on CLOSE 440
- on CONNECT 442
- on FCNTL 444
- on GETPEERNAME 470
- on GETSOCKNAME 472
- on GETSOCKOPT 474
- on GIVESOCKET 484
- on IOCTL 488
- on LISTEN 494
- on READ 500
- on READV 502
- on RECV 504
- on RECVFROM 506
- on RECVMSG 510
- on SEND 521
- on SENDMSG 524
- on SENDTO 528
- on SETSOCKOPT 531
- on SHUTDOWN 540
- on WRITE 546
- on WRITEV 547

sample programs

C socket

- TCP client 212
- TCP server 215
- UDP client 221
- UDP server 218

call interface

- CBLOCK, PL/I 583
- client, PL/I 570
- server, PL/I 566

IUCV sockets

- client, C language 53, 775
- server, C language 53, 765
- subtask, C language 53, 772

macro interface

- client, assembler language 394
- server, assembler language 384

TCP/UDP/IP Pascal 754

- SayCalRe 737
- SayConSt 737
- SayIntAd 738
- SayIntNum 738
- SayNotEn 739
- SayPorTy 739
- SayProTy 739
- SCEERUN 104
- SELECT (call) 512
- SELECT (macro) 344
- select mask 46
- Select, Read, Write, Exception (REXX) 679
- select, server, socket call, general 45
- select() 177
- SELECTEX (call) 516
- SELECTEX (macro) 349
- selectex() 181
- selecting sockets 5
- SEND (call) 520
- SEND (macro) 352
- Send (REXX) 642
- send() 183
- sendflags parameter on REXX socket calls
 - on Send (input) 642
 - on Sendto (input) 643
- SENDINGonly 714
- SENDMSG (call) 522
- SENDMSG (macro) 354
- sendmsg() 185
- SENDTO (call) 526
- SENDTO (macro) 358
- Sendto (REXX) 643
- sendto() 187
- server
 - allocate() 27
 - select() 45
- service parameter on REXX socket calls
 - on Initialize (input) 620
 - on Initialize (output) 620
- servicename parameter on REXX socket calls
 - on Getservbyname (input) 662
 - on Getservbyname (output) 662
 - on Getservbyport (output) 663
- sethostent() 189
- setibmopt() 190
- setibmssockopt() 192
- setnetent() 194
- setprotoent() 195
- setservernt() 196
- SETSOCKOPT (call) 530
- SETSOCKOPT (macro) 361
- Setsockopt (REXX) 680
- setsockopt() 197
- SetTimer 740
- severreason parameter on REXX socket call, on Socketsetstatus
 - (output) 623
- shortcut keys 825
- shouldwait parameter on TCP/UDP/IP (Pascal), on
 - GetNextNote 729
- SHUTDOWN (call) 539
- SHUTDOWN (macro) 371
- Shutdown (REXX) 633
- shutdown() 201
- smmsg parameter on TCP/UDP/IP (Pascal), on GetSmsg 730
- MSGreceived (Pascal) 719
- SO_BULKMODE, on C socket calls. 133
- SO_NONBLOCKLOCAL, on C socket calls. 133
- Socket_debug() 202
- SOCK_DGRAM 6
- Sock_do_teststor() 203
- SOCK_RAW 6
- SOCK_STREAM 108
- SOCKET (call) 541
- SOCKET (macro) 373
- Socket (REXX) 634
- socket call syntax, C 106
- socket definition 5
- socket libraries
 - Native TCP/IP environment 7
 - UNIX environment 8
- socket return codes 781
- socket service types
 - datagram socket 6
 - raw socket 6
 - stream socket 5
- Socket, built-in function 618
- socket() 204
- socketid parameter on REXX socket calls
 - on Accept (input) 626
 - on Bind (input) 627
 - on Close (input) 629
 - on Connect (input) 630
 - on Fcntl (input) 667
 - on Getsockname (input) 664
 - on Getsockopt (input) 668
 - on Givesocket (input) 631, 659
 - on Ioctl (input) 677
 - on Listen (input) 632
 - on Read (input) 638
 - on Recv (input) 639
 - on Recvfrom (input) 640
 - on Send (input) 642
 - on Sendto (input) 643
 - on Setsockopt (input) 680
 - on Shutdown (input) 633
 - on Takesocket (input) 636
 - on Write (input) 645
- socketidlist parameter on REXX socket calls
 - on Select (input) 679
 - on Select (output) 679
- sockets
 - addresses 12, 104
 - connected 18, 69
 - data transfer 6
 - domain parameter 27
 - guidelines for using 6
 - header files
 - MANIFEST.H 105
 - SOCKET.H 163
 - implementation 104
 - interface
 - datagram 6
 - raw 6
 - stream 5
 - transaction 6
 - library 105
 - performance 6
 - protocol parameter 27
 - TCP socket 17
 - type parameter 27
 - typical TCP socket session 17
 - typical UDP socket session 18
 - UDP socket 18
 - unconnected 18

- sockets concepts 4
- Sockets Extended
 - definition of call instruction API 8
 - definition of macro API 8
- sockets programming 5
- sockets, maximum number 11
- socketset, REXX sockets 622
- socketsetlist, REXX sockets 621
- socketsetstatus, REXX sockets 623
- SOCRECV parameter on call interface, TAKESOCKET
 - call 544
- SOCRECV parameter on macro interface, TAKESOCKET
 - call 377
- SOCTYPE parameter on call interface, on SOCKET 542
- SOCTYPE parameter on macro interface, on SOCKET 374
- software requirements, Pascal 712
- SOL_SOCKET, on C socket calls. 133, 145, 197
- state parameter on TCP/UDP/IP (Pascal), on SayConSt 738
- status parameter on REXX interface, on Socketsetstatus
 - (output) 623
- storage definition, macro interface
 - STORAGE=CSECT on EZASMI 251, 252
 - STORAGE=DSECT on EZASMI 251, 252, 316, 378
- STORAGE parameter on macro interface
 - on GLOBAL call 316
 - on TASK call 379
- stream sockets 5
- strerror() 104
- structures, C 106
- subfunction parameter, on built-in function Socket, REXX
 - interface 618
- SUBTASK parameter on call interface, INITAPI call 486
- SUBTASK parameter on macro interface, INITAPI call 318
- subtaskid parameter on REXX socket call interface
 - on Initialize (input) 620
 - on Initialize (output) 620
 - on Socketset (input) 622
 - on Socketset (output) 622
 - on Socketsetstatus (input) 623
 - on Socketsetstatus (output) 623
 - on Terminate (input) 624
 - on Terminate (output) 624
- subtaskidlist parameter on REXX socket call, on Socketsetlist
 - (output) 621
- Success parameter on TCP/UDP/IP (Pascal), on GetSmsg 730
- syntax, C socket call 106
- system errors, printing 95

T

- T parameter on TCP/UDP/IP (Pascal) procedure
 - on ClearTimer 726
 - on CreateTimer 726
 - on DestroyTimer 726
 - on SetTimer 740
- TableName parameter on TCP/UDP/IP (Pascal), on
 - ReadXlateTable 737
- TAKESOCKET (call) 543
- TAKESOCKET (macro) 376
- Takesocket (REXX) 636
- takesocket() 207
- TASK (macro) 378
- task management, macro calls 255
- tasks
 - <gerund phrase>
 - steps for 711

- TCP
 - communication procedures (PASCAL) 722
 - socket session 17
- TCP/IP
 - online information xxvi
 - protocol specifications 811
- TCP/IP concepts 3
- TCP/UDP/IP API (pascal language) 711
- TCP/UDP/IP initialization procedures (Pascal) 721
- TCP/UDP/IP termination procedure (Pascal) 721
- TcpAbort (Pascal) 740
- TcpClose (Pascal) 741
- tcperror() 208
- TcpFReceive (Pascal) 741
- TcpFSend (Pascal) 743
- TcpIpServiceName parameter on TCP/UDP/IP (Pascal), on
 - GetIdentity 729
- TcpNameChange 746
- TcpOpen (Pascal) 746
- TcpOption (Pascal) 748
- TcpReceive (Pascal) 741
- TcpSend (Pascal) 743
- tcperror() 104
- TcpStatus (Pascal) 749
- TcpWaitOpen (Pascal) 746
- TcpWaitReceive 741
- TcpWaitSend (Pascal) 743
- TERMAPI (call) 544
- TERMAPI (macro) 379
- test address, loopback 22
- test tools
 - Character Generator 21
 - Discard 21
 - Echo 21
 - loopback address 22
 - miscellaneous server (MISCSRV) 21
- timeout parameter on C socket calls
 - on select() 177
 - on selectex() 181
- TIMEOUT parameter on call interface, on SELECT 515
- TIMEOUT parameter on call socket interface
 - on SELECTEX 519
- TIMEOUT parameter on macro interface, on SELECT 347
- TIMEOUT parameter on macro socket interface
 - on SELECTEX 350
- timeout parameter on REXX socket call, on Select (input) 679
- Timeout parameter on TCP/UDP/IP (Pascal), on
 - PingRequest 733
- Timer Routines 722
- TIMERexpired (Pascal) 719
- to parameter on C socket calls 187
- TOKEN parameter on call interface, on EZACIC06 552
- tolen parameter on C socket calls 187
- totdesc parameter on C socket calls 164
- trademark information 836
- transaction sockets 6
- transferring data between sockets 63
- TranslateTableSpec parameter on TCP/UDP/IP (Pascal), on
 - ReadXlateTable 737
- translation tables, REXX sockets 616
- transport, protocol layer 5
- TRYINGtoOPEN (Pascal) 714
- type parameter on C socket call
 - on getnetbyaddr() 134
 - on socket() 204
- type parameter on REXX socket call, on Socket (input) 634
- typical TCP socket session 17

typical UDP socket session 18

U

U4093, user abend 794

UDP

- communication procedures 722

- socket session 18

UdpClose (Pascal) 750

UDPdatagramDELIVERED (Pascal) 720

UDPdatagramSPACEavailable (Pascal) 720

UdpNReceive 750

UdpOpen (Pascal) 751

UdpReceive (Pascal) 752

UdpSend (Pascal) 753

Unhandle (Pascal) 754

UnpackedBytes 715

unsolicited event exit 258

urgentflag parameter on TCP/UDP/IP (Pascal) procedure, on

- TcpFsend, TcpSend, TcpWaitSend 745

URGENTpending (Pascal) 720

use of HOSTENT structure interpreter, EZACIC08 554

user abend U4093 794

USERdefinedNOTIFICATION (Pascal) 720

userid parameter on TCP/UDP/IP (Pascal), on

- GetIdentity 729

using socket implementation 104

utility programs 548

- EZACIC04 550

- EZACIC05 551

- EZACIC06 552

- EZACIC08 554

- EZACIC14 562

- EZACIC15 564

V

Versatile Message Transfer Protocol (VMTP) 6

version parameter on REXX socket call, on version

- (output) 689

Version, REXX sockets 689

VTAM, online information xxvi

W

WRETMASK parameter on call interface, on SELECT 516

WRETMASK parameter on macro interface, on SELECT 348

WRITE (call) 545

WRITE (macro) 379

Write (REXX) 645

write() 209

writetds parameter on C socket calls

- on select() 177

- on selectex() 181

WRITEV (call) 547

WRITEV (macro) 381

writev() 210

WSNDMSK parameter on call interface, on SELECT 515

WSNDMSK parameter on macro interface, on SELECT 348

X

X/Open Transport Interface (XTI)

- fcntl() 228

- RFC1006 225

X/Open Transport Interface (XTI) *(continued)*

- select() 228

- selectex() 228

- t_accept() 227

- t_bind() 227

- t_close() 228

- t_connect() 227

- t_error() 228

- t_getinfo() 228

- t_getstate() 228

- t_listen() 227

- t_look() 228

- t_open() 227

- t_rcv() 227

- t_rcvconnect() 227

- t_rcvdis() 227

- t_snd() 227

- t_snddis() 227

- t_unbind() 228

XPG4 standard

- using z/OS UNIX C sockets API with 8

XTI call library 225

XTI management services 225

Z

z/OS, documentation library listing 839

z/OS, listing of documentation available 821

Communicating Your Comments to IBM

If you especially like or dislike anything about this document, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply.

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this document. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Please send your comments to us in either of the following ways:

- If you prefer to send comments by FAX, use this number: 1+919-254-9823
- If you prefer to send comments electronically, use this address:
 - comsvrcf@us.ibm.com.
- If you prefer to send comments by post, use this address:
 - International Business Machines Corporation
 - Attn: z/OS Communications Server Information Development
 - P.O. Box 12195, 3039 Cornwallis Road
 - Department AKCA, Building 501
 - Research Triangle Park, North Carolina 27709-2195

Make sure to include the following in your note:

- Title and publication number of this document
- Page number or topic to which your comment applies.



Program Number: 5694-A01 and 5655-G52

Printed in USA

SC31-8788-04



Spine information:



z/OS Communications Server

z/OS V1R6.0 CS: IP Application Programming Interface Guide

Version 1
Release 6