

IBM z/VSE



z/VSE TCP/IP Support

Version 5

IBM z/VSE



z/VSE TCP/IP Support

Version 5

Note!

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

This edition applies to the following programs

- Version 5 of IBM z/Virtual Storage Extended (z/VSE), Program Number 5609-ZV5
- Version 1 Release 5 of TCP/IP for VSE/ESA, Program Number 5686-A04
- Version 1 Release 1 of IPv6/VSE, Program Number 5686-BS1

and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces SC34-2640-02.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the addresses given below.

A form for readers' comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Deutschland Research & Development GmbH
Department 3282
Schoenaicher Strasse 220
D-71032 Boeblingen
Federal Republic of Germany

You may also send your comments by FAX or via the Internet:

Internet: s390id@de.ibm.com
FAX (Germany): 07031-16-3456
FAX (other countries): (+49)+7031-16-3456

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

© Copyright IBM Corporation 1997, 2014.

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

Contents

Figures ix

Tables xi

Notices xiii

Trademarks xiii

Accessibility xv

Using Assistive Technologies xv

Documentation Format xv

About This Publication xvii

Where to Find More Information. xvii

Understanding Syntax Diagrams. xvii

Summary of Changes xxi

Important Considerations - Read this First! xxiii

Part 1. Using TCP/IP for VSE/ESA . . . 1

Chapter 1. Overview 3

Documentation for the TCP/IP for VSE/ESA

(5686-A04) Program 3

General Considerations on the TCP/IP for VSE/ESA

Program Setup. 3

The Demo Mode for TCP/IP for VSE/ESA 4

Supplying the Product Key 5

Installing Product Keys. 6

Defining Customer Information 6

Migration Considerations 6

Chapter 2. TCP/IP for VSE/ESA

Configuration 9

How TCP/IP for VSE/ESA is Installed 9

TCP/IP for VSE/ESA Partition Startup 9

Configuring CICS 10

 Example for CICS/TS 1.1. 12

HTMLINST.Z. 13

Chapter 3. TCP/IP for VSE/ESA

Configuration Dialogs 15

Configuring TCP/IP Using the Configuration

Dialogs. 15

 Configuring TCP/IP Using the IUI-based

 Configuration Dialog 16

Chapter 4. Security Manager

Exploitation by TCP/IP for VSE/ESA . . . 23

Using BSM Capabilities for TCP/IP Security Checks 23

Activation of The Security Exit 24

Deactivation of the Security Exit 25

Using Pre- and Postprocessing Exits 25

Chapter 5. InfoPrint Manager Support of TCP/IP for VSE/ESA 29

Setting the Parameters for the IPM Support 30

Customizing the InfoPrint Manager 31

Changing the Properties of the Actual Destination 31

Technical Background Information. 32

Software Prerequisites 32

Chapter 6. z/VSE Related Hardware Functions Supported by TCP/IP for VSE/ESA 1.5 33

Chapter 7. Performance Considerations 35

Changing Performance Parameters 35

General Performance Issues 35

Principal Performance Dependencies for TCP/IP for

VSE/ESA 36

Part 2. Using IPv6/VSE 39

Chapter 8. IPv6/VSE Overview. 41

IPv6 TCP/IP Stack 41

Dual Stack Support. 41

IPv6-Enabled Utility Applications 41

Documentation for the IPv6/VSE (5686-BS1)

Program 43

IPv6/VSE Installation Requirements 43

Part 3. Programming Interfaces. 45

Chapter 9. Introducing Socket Programming 47

What is a TCP/IP Socket Connection ? 47

Socket Application Programming Interfaces

Available with z/VSE 48

Portability Aspects 49

Which API to use? 50

Exploiting the LE/VSE Socket API. 52

 C Language 52

 Assembler Language 53

 PL/I. 54

 COBOL. 55

Exploiting the EZASMI/EZASOKET Programming

Interfaces 60

LE/VSE 1.4 C Socket Programming 69

 General C Programming Considerations. 69

 Messages 70

TCP/IP Functions Supported by z/VSE 71

ERRNO Values 74

CICS Considerations 83

CICS Considerations for the EZA Interfaces	83
Executing TCP/IP Application Programs	84

Chapter 10. TCP/IP Support for the LE/VSE C Socket Interface 85

TCP/IP Callable Functions — Function Descriptions	85
accept() — Accept a New Connection on a Socket	85
aio_cancel() — Cancel an Asynchronous I/O Request.	88
aio_error() — Retrieve Error Status for an Asynchronous I/O Operation	89
aio_read() — Asynchronous Read from a Socket	90
aio_return() — Retrieve Status for an Asynchronous I/O Operation	92
aio_suspend() — Wait for an Asynchronous I/O Request.	93
aio_write() — Asynchronous Write to a Socket.	94
bind() — Bind a Name to a Socket.	96
close() — Close a Socket	99
connect() — Connect a Socket	100
endhostent() — Work with a Host Entry	103
endnetent() — Close Network Information Data Sets	103
endprotoent() — Work with a Protocol Entry	103
endservent() — Close Network Services Information Data Sets	104
fcntl() — Control Open Socket Descriptors	104
freeaddrinfo() — Free addrinfo storage.	105
gai_strerror() — Address and name information error description	106
getaddrinfo() — Get address information	106
getclientid() — Get the Identifier for the Calling Application	110
gethostbyaddr() — Get a Host Entry by Address	111
gethostbyname() — Get a Host Entry by Name	113
gethostent() — Get the Next Host Entry	114
gethostid() — Get the Unique Identifier of the Current Host	115
gethostname() — Get the Name of the Host Processor	115
getibmopt() — Get IBM TCP/IP image	116
getnameinfo() — Get name information	117
getnetbyaddr() — Get a Network Entry by Address	119
getnetbyname() — Get a Network Entry by Name	120
getnetent() — Get the Next Network Entry	121
getpeername() — Get the Name of the Peer Connected to a Socket	121
getprotobyname() — Get a Protocol Entry by Name	123
getprotobynumber() — Get a Protocol Entry by Number	123
getprotoent() — Get the Next Protocol Entry	124
getservbyname() — Get a Service Entry by Name	125
getservbyport() — Get a Service Entry by Port	125
getservent() — Get the Next Service Entry.	126
getsockname() — Get the Name of a Socket	127
getsockopt() — Get the Options Associated with a Socket	128

givesocket() — Make the Specified Socket Available	133
gsk_free_memory() — Free memory allocated for SSL	135
gsk_get_cipher_info() — Query Cipher Related Information	135
gsk_get_dn_by_label() — Get Distinguished Name Based on the Label	137
gsk_initialize() — Initialize the SSL Environment	137
gsk_secure_soc_close() — Close a Secure Socket Connection	139
gsk_secure_soc_init() — Initialize Data Areas for a Secure Socket Connection.	140
gsk_secure_soc_read() — Receive Data on a Secure Socket Connection	143
gsk_secure_soc_reset() — Refresh the Security Parameters	144
gsk_secure_soc_write() — Send Data on a Secure Socket Connection	145
gsk_uninitialize() — Remove Current Settings for the SSL Environment	146
gsk_user_set() — Provide Callback Routines	146
htonl() — Translate Address Host to Network Long	147
htons() — Translate an Unsigned Short Integer into Network Byte Order	147
if_freenameindex() — Free the Memory Allocated by if_nameindex()	148
if_indextoname() — Map a Network Interface Index to its Corresponding Name	148
if_nameindex() — Return all Network Interface Names and Indexes	149
if_nametoindex() — Map a Network Interface Name to its Corresponding Index	150
inet_addr() — Translate an Internet Address into Network Byte Order	150
inet_lnaof() — Translate a Local Network Address into Host Byte Order.	151
inet_makeaddr() — Create an Internet Host Address	152
inet_netof() — Get the Network Number from the Internet Host Address	152
inet_network() — Get the Network Number from the Decimal Host Address	153
inet_ntoa() — Get the Decimal Internet Host Address	153
inet_ntop() — Convert Internet Address Format from Binary to Text	154
inet_pton() — Convert Internet Address Format from Text to Binary	155
initapi() — Initialize Socket API for a Subtask	156
ioctl() — Control Socket.	156
listen() — Prepare the Server for Incoming Client Requests.	158
maxdesc() — Get Socket Numbers to Extend Beyond the Default Range	159
ntohl() — Translate a Long Integer into Host Byte Order	160
ntohs() — Translate an Unsigned Short Integer into Host Byte Order	160
poll() — Monitor Activity on Socket Descriptors	161

read()	— Read From a Socket	162
readv()	— Read Data on a Socket and Store in a Set of Buffers	164
recv()	— Receive Data on a Socket	165
recvfrom()	— Receive Messages on a Socket	167
recvmsg()	— Receive Messages on a Socket and Store in an Array of Message Headers	168
select()	— Monitor Activity on Sockets	170
selectex()	— Monitor Activity on Sockets	174
send()	— Send Data on a Socket	176
sendmsg()	— Send Messages on a Socket	177
sendto()	— Send Data on a Socket	179
sethostent()	— Open the Host Information Data Set	181
setibmopt()	— Set IBM TCP/IP Image	181
setnetent()	— Open the Network Information Data Set	182
setprotoent()	— Open the Protocol Information Data Set	182
setservent()	— Open the Network Services Information Data Set	183
setsockopt()	— Set Options Associated with a Socket	183
shutdown()	— Shut Down a Connection	189
socket()	— Create a Socket	190
socketpair()	— Create a Pair of Sockets	193
takesocket()	— Acquire a Socket from Another Program	193
termapi()	— Terminate the Socket API for a Subtask	194
write()	— Write Data on a Socket	195
writex()	— Write Data on a Socket from an Array	196

Chapter 11. Using the CALL Instruction Application Programming Interface (EZASOKET API) 199

Environmental Restrictions and Programming Requirements	199
CALL Instruction Application Programming Interface (API)	199
Understanding COBOL, Assembler, and PL/I Call Formats	200
Converting Parameter Descriptions	201
Error Messages and Return Codes	201
Debugging	201
Code CALL Instructions	201
ACCEPT	201
BIND	204
CLOSE	206
CONNECT	207
FCNTL	209
FREEADDRINFO	211
GETADDRINFO	211
GETCLIENTID	218
GETHOSTBYADDR	219
GETHOSTBYNAME	221
GETHOSTID	222
GETHOSTNAME	223
GETIBMOPT	224

GETNAMEINFO	225
GETPEERNAME	229
GETSOCKNAME	230
GETSOCKOPT	232
GIVESOCKET	234
GSKFREEMEM	236
GSKGETCIPHINF	237
GSKGETDNBYLAB	238
GSKINIT	239
GSKSSOCCLOSE	240
GSKSSOCINIT	241
GSKSSOCREAD	244
GSKSSOCRESET	245
GSKSSOCWRITE	245
GSKUNINIT	246
INITAPI	247
IOCTL	249
LISTEN	250
NTOP	251
PTON	253
READ	254
READV	255
RECV	257
RECVFROM	258
SELECT	261
SELECTEX	265
SEND	267
SENDTO	268
SETSOCKOPT	270
SHUTDOWN	274
SOCKET	275
TAKESOCKET	276
TERMAPI	278
WRITE	278
WRITEV	279
Using Data Translation Programs for Socket Call Interface	280
EZACIC04	281
EZACIC05	281
EZACIC06	282
EZACIC08	284
EZACIC09	286

Chapter 12. Using the Macro Application Programming Interface (EZASMI API) 291

Environmental Restrictions and Programming Requirements	291
EZASMI Macro Application Programming Interface (API)	292
Defining Storage for the API Macro	292
Understanding Common Parameter Descriptions	293
Characteristics of Stream Sockets	294
Task Management and Asynchronous Function Processing	294
Error Messages and Return Codes	295
Debugging	295
Macros for Assembler Programs	295
ACCEPT	296
BIND	298

CANCEL	300
CLOSE	301
CONNECT	302
FCNTL	305
FREEADDRINFO	306
GETADDRINFO	307
GETCLIENTID	314
GETHOSTBYADDR	315
GETHOSTBYNAME	317
GETHOSTID	319
GETHOSTNAME	320
GETIBMOPT	321
GETNAMEINFO	322
GETPEERNAME	326
GETSOCKNAME	328
GETSOCKOPT	330
GIVESOCKET	333
GSKFREEMEM	335
GSKGETCIPHINF	335
GSKGETDNBYLAB	336
GSKINIT	337
GSKSSOC_CLOSE	338
GSKSSOCINIT	339
GSKSSOCREAD	343
GSKSSOCRESET	344
GSKSSOCWRITE	345
GSKUNINIT	345
INITAPI	346
IOCTL	348
LISTEN	350
NTOP	351
PTON	352
READ	354
READV	355
RECV	357
RECVFROM	358
SELECT	360
SELECTEX	365
SEND	367
SENDTO	368
SETSOCKOPT	371
SHUTDOWN	375
SOCKET	376
TAKESOCKET	379
TASK	381
TERMAPI	381
WRITE	382
WRITEV	383

Part 4. Using Fast Path to Linux 385

Chapter 13. Running z/VSE With a Linux Fast Path 387

Overview of Linux Fast Path	387
Prerequisites for Using Linux Fast Path.	388
Restrictions When Using Linux Fast Path	389
Communication Flow When Not Using Linux Fast Path	389
Communication Flow When Using Linux Fast Path in a z/VM Environment.	390

Communication Flow When Using Linux Fast Path in an LPAR Environment	391
Preparing Linux on System z to Use Linux Fast Path	393
Preparing to use Socket APIs with Linux Fast Path on z/VSE	395
Configuring Linux Fast Path	397
z/VM	397
Linux on System z	398
z/VSE.	402
Sample configurations	405
Starting and Stopping Linux Fast Path	407
z/VSE.	407
Linux on System z	408
Administrative Tasks	411
z/VSE.	411
Linux on System z	414

Chapter 14. z/VSE - z/VM IP Assist 417

Communication Flow when Using z/VSE VIA z/VSE VIA z/VM guest configuration	417
z/VSE VIA Linux Fast Path Configuration	419
z/VSE VIA Linux Fast Path Administration	420
LFPD Command	420
LFPD-ADMIN Command	421

Chapter 15. OpenSSL 423

Key store considerations.	424
Creating the Key Store Using Keyman/VSE	424
Programming z/VSE Applications for OpenSSL	427
Include Files.	427
Passed Socket Number	428
Callback routines	429
Socket Calls	429
Switching between gsk and OpenSSL Socket Calls	430
Specifying the Key Ring	430
Using a Password Protected Keyring	431
Supported Cipher Suites.	432
Specifying Cipher Suites.	433
Supported RSA Key Lengths	433
Debugging	434
Hardware Crypto Support	434
Performing the OpenSSL speed test	435
z/OS SSL API	436

Part 5. CICS Listener Support 439

Chapter 16. Setting Up and Configuring CICS Listener Support 441

CICS — Defining CICS Resources	441
Transaction Definitions	441
Program Definitions	442
File Definitions	443
Transient Data Definition	443
CICS Monitoring	444
CICS Program List Table (PLT)	445
Configuring the CICS TCP/IP Environment	445

Building the Configuration data set with the Configuration Macro (EZACICD)	446	4. The Client CICS TCP/IP Application.	499
Customizing the Configuration Dataset.	451	Socket Addresses	499
Chapter 17. Configuring the CICS Domain Name Server Cache	469	GETCLIENTID, GIVESOCKET, and TAKESOCKET	501
Function Components	469	The Listener	502
How the DNS Cache Handles Requests	470	Listener Input Format (Standard Listener only)	502
Using the DNS Cache	471	Listener Output Format	504
Step 1: Create the Initialization Module.	471	Writing Your Own Security Link Module for the Listener	505
Step 2: Define the Cache File to CICS	473	Data Conversion Routines	507
Step 3: Execute EZACIC25	473		
Chapter 18. Starting and Stopping the CICS Listener Support	475	<hr/>	
Starting/Stopping CICS Listener Support		Part 6. Appendixes	509
Automatically	475	Appendix A. Examples to be used with TCP/IP for VSE/ESA	511
Starting/Stopping CICS Manually	475	Autonomous FTP	511
START Function	476	AUTOLPR – Printing with the CICS Report	
STOP Function	478	Controller Feature (RCF)	512
Starting/Stopping CICS Listener Support with Program Link	480	GPS and RCF	513
Chapter 19. Writing Your Own Listener	483	TELNET and Subnetting in a Class-C Network	514
Using IBM's Environmental Support.	483	TELNET daemons and logmode	515
Chapter 20. External Data Structures	487	VSAMCAT Usage	515
Configuration Data Set Record Formats	487	Using the Command preprocessor	516
Global Work Area	488	Sample Programs	517
Parameter List (COMMAREA) for EZACIC20	491	Compiling Your Program	521
Listener Control Area (LCA)	492	Appendix B. Debugging Facility for EZASMI and EZASOKET Interfaces (EZAAPI Trace).	525
Chapter 21. CICS Listener Programming Considerations	493	Appendix C. Advanced OSAX Device Driver Configuration	527
Writing CICS TCP/IP Applications	493	Configurable QDIO Buffers.	527
1. The Client-Listener-Child-Server Application Set	494	VLAN Support	527
2. Writing Your Own Concurrent Server	496	Index	529
3. The Iterative Server CICS TCP/IP Application	498	Glossary	539

Figures

1.	TCP/IP Configuration Panel CON\$SEL	16	25.	z/VSE LFP via IUCV Configuration Example	405
2.	TCP/IP Configuration Panel: Set IPADDR and MASK	17	26.	Linux LFP via IUCV Configuration Example	406
3.	TCP/IP Configuration Panel: Link List	17	27.	z/VSE LFP via HiperSockets Configuration Example	406
4.	TCP/IP Configuration: Link	18	28.	Linux LFP via HiperSockets Configuration Example	407
5.	TCP/IP Configuration Panel: Adapter List	18	29.	Linux System Log Output Example	410
6.	TCP/IP Configuration Panel: Adapter	19	30.	Linux System Log Output Example of a Disconnected z/VSE System	410
7.	TCP/IP Configuration Panel: Route List	19	31.	Active Instance List Output Example	411
8.	TCP/IP Configuration Panel: Define Route	20	32.	LFP Instance on z/VSE via IUCV Connection Output Example	411
9.	TCP/IP Configuration Panel: TELNET LIST	20	33.	LFP Instance on z/VSE via HiperSockets Connection Output Example	413
10.	TCP/IP Configuration Panel: TELNET DAEMON	21	34.	Socket Diagnosis Output Example	414
11.	LPR-Job on TCP/IP for VSE/ESA	30	35.	lfpd-admin via IUCV Connection Output Example	414
12.	Control Flow when using LE/VSE C Sockets with different TCP/IP stacks	52	36.	lfpd-admin via HS CQ Connection Output Example	415
13.	Sample Program Using EZASMI Macro (Synchronously)	60	37.	Communication using z/VSE VIA and an IUCV connection	417
14.	Sample Program Using EZASMI Macro (Asynchronously)	62	38.	z/VSE VIAAdministrative Command Output Example	420
15.	Sample Program Using EZASOKET Call Using COBOL	66	39.	z/VSE VIA startdbg Command Output Example	421
16.	Storage Definition Statement Examples	201	40.	Addition to the DCT Required by CICS TCP/IP	443
17.	HOSTENT Structure Returned by the GETHOSTBYADDR Call	220	41.	The Monitor Control Table (MCT) for Listener	444
18.	HOSTENT Structure Returned by the GETHOSTBYNAME Call	222	42.	The DNS Hostent	474
19.	ECB Input Parameter	295	43.	The Sequence of Sockets Calls	495
20.	HOSTENT Structure Returned by the GETHOSTBYADDR Macro	316	44.	Sequence of Socket Calls with an Iterative Server	498
21.	HOSTENT Structure Returned by the GETHOSTBYNAME Macro	318	45.	Sequence of Socket Calls between a CICS Client and a Remote Iterative Server	499
22.	Communication Between VM Guests via HiperSockets	390	46.	Transfer of CLIENTID Information	501
23.	Communication Between VM Guests via a Linux Fast Path	391			
24.	Communication Between LPARs via HiperSockets	392			

Tables

1. Principal Performance Parameters	36	13. Conditions for Translation of Tranid and User Data	451
2. TCP/IP Performance-Relevant Parameters	37	14. Configuration File Format	487
3. Supported call functions by Interface and TCP/IP Stack	71	15. Global Work Area Format	489
4. ERRNO Values Sorted by Value.	74	16. COMMAREA Format for EZACIC20	491
5. EZASMI/EZASOKET only - ERRNO Values Sorted by Value	79	17. Listener Control Area (LCA)	492
6. ERRNO Values sorted by Name.	79	18. Calls for the Client Application	495
7. COMMAREA parameter list	83	19. Calls for the Server Application	496
8. NUM_IMAGES Field Settings	321	20. Calls for the Concurrent Server Application	497
9. IOCTL Macro Arguments	349	21. Listener Output Format - using the IPv4 protocol	504
10. Specifying Instance IDs	396	22. Output from EZA Listener to the Child Server - Enhanced mode - using the IPv6 protocol .	504
11. OpenSSL Cipher Suites	432	23. Output from EZA Listener to the Security / Transaction Exit.	505
12. Listener action based on RTYTIME and stack state	450		

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM 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 of the intellectual property rights of IBM may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter 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 the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785, U.S.A.

Any pointers in this publication to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement. IBM accepts no responsibility for the content or use of non-IBM websites specifically mentioned in this publication or accessed through an IBM website that is mentioned in this publication.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Deutschland GmbH
Dept. M358
IBM-Allee 1
71139 Ehningen
Germany

Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

IPv6/VSE is a registered trademark of Barnard Software, Inc.

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/VSE 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/VSE. Consult the assistive technology documentation for specific information when using such products to access z/VSE interfaces.

Documentation Format

The publications for this product are in Adobe Portable Document Format (PDF) and should be compliant with accessibility standards. If you experience difficulties when you use the PDF files and want to request a web-based format for a publication, you can either write an email to s390id@de.ibm.com, or use the Reader Comment Form in the back of this publication or direct your mail to the following address:

IBM Deutschland Research & Development GmbH
Department 3282
Schoenaicher Strasse 220
D-71032 Boeblingen
Federal Republic of Germany

In the request, be sure to include the publication number and title.

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.

About This Publication

This publication describes how to communicate with an IBM® z/VSE host using TCP/IP and the programs that enable the connection and interchange.

Before using this publication read the section “Important Considerations - Read this First!” on page xxiii. It gives you a brief overview on the content and structure of this manual.

Where to Find More Information

z/VSE Home Page

z/VSE has a home page on the World Wide Web, which offers up-to-date information about VSE-related products and services, new z/VSE functions, and other items of interest to VSE users.

You can find the z/VSE home page at

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

You can also find VSE User Examples (in zipped format) at

<http://www.ibm.com/systems/z/os/zvse/downloads/samples.html>

Understanding Syntax Diagrams

This section describes how to read the syntax diagrams in this manual.

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

- The ►— symbol indicates the beginning of a syntax diagram.
- The —→ symbol, at the end of a line, indicates that the syntax diagram continues on the next line.
- The ►— symbol, at the beginning of a line, indicates that a syntax diagram continues from the previous line.
- The —► symbol indicates the end of a syntax diagram.

Syntax items (for example, a keyword or variable) may be:

- Directly on the line (required)
- Above the line (default)
- Below the line (optional)

Uppercase Letters

Uppercase letters denote the shortest possible abbreviation. If an item appears entirely in uppercase letters, it can not be abbreviated.

You can type the item in uppercase letters, lowercase letters, or any combination. For example:



In this example, you can enter KEYWO, KEYWOR, or KEYWORD in any combination of uppercase and lowercase letters.

Symbols

You **must** code these symbols exactly as they appear in the syntax diagram

- * Asterisk
- :
- ,
- = Equal Sign
- Hyphen
- // Double slash
- () Parenthesis
- .
- + Add

For example:

* \$\$ LST

Variables

Highlighted lowercase letters denote variable information that you must substitute with specific information. For example:



Here you must code USER= as shown and supply an ID for user_id. You may, of course, enter USER in lowercase, but you must not change it otherwise.

Repetition

An arrow returning to the left means that the item can be repeated.



A character within the arrow means you must separate repeated items with that character.



A footnote (1) by the arrow references a limit that tells how many times the item can be repeated.



Notes:

- 1 Specify *repeat* up to 5 times.

Defaults

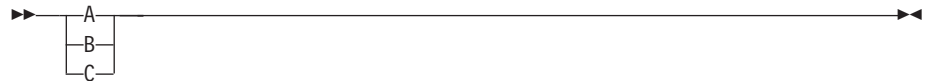
Defaults are above the line. The system uses the default unless you override it. You can override the default by coding an option from the stack below the line. For example:



In this example, A is the default. You can override A by choosing B or C.

Required Choices

When two or more items are in a stack and one of them is on the line, you **must** specify one item. For example:



Here you must enter either A or B or C.

Optional Choice

When an item is below the line, the item is optional. Only one item **may** be chosen. For example:



Here you may enter either A or B or C, or you may omit the field.

Required Blank Space

A required blank space is indicated as such in the notation. For example:

* \$\$ E0J

This indicates that at least one blank is required before and after the characters \$\$.

Summary of Changes

This publication has been updated to reflect enhancements and changes that are implemented with z/VSE 5.2. It also includes terminology, maintenance, and editorial changes.

These are the enhancements that have been made available with z/VSE 5.2

- The TCP/IP for VSE/ESA product provided by Connectivity Systems Incorporated is now installed in library PRD2.TCPIPC and no longer in PRD1.BASE. Therefore, jobs that use TCP/IP for VSE/ESA must now have PRD2.TCPIPC as the first library in their library search chain.
- CICS Listener Support has been enhanced to support IPv6, specification of an TCP/IP stack, and more. For details refer to Part 5, "CICS Listener Support," on page 439.

These are the enhancements that have been made available via the June 2013 Service Upgrade of z/VSE 5.1

- z/VSE now supports OpenSSL. For details refer to Chapter 15, "OpenSSL," on page 423.
- Appendix C, "Advanced OSAX Device Driver Configuration," on page 527 has been added to this publication.

These are the enhancements that have been made available via the June 2012 Service Upgrade of z/VSE 5.1:

- The Linux Fast Path can now connect z/VSE and Linux images running in LPAR environments.
 - The communication flow is described in "Communication Flow When Using Linux Fast Path in an LPAR Environment" on page 391.
 - The new configuration parameters are described in "Configuring Linux Fast Path" on page 397.
- An additional chapter has been added to reflect the new z/VSE - z/VM IP Assist function. This function is intended for customers who want run a z/VM guest image without running a Linux on System z system. For details refer to Chapter 14, "z/VSE - z/VM IP Assist," on page 417.

With z/VSE 5.1

The following EZASMI/EZASOKET interfaces support new options, especially for IPv6 support:

- GETSOCKOPT (refer to "GETSOCKOPT" on page 330 and "GETSOCKOPT" on page 232 for details.)
- SETSOCKOPT (refer to "SETSOCKOPT" on page 371 and "SETSOCKOPT" on page 270 for details.)

Important Considerations - Read this First!

This publication describes how to communicate with an IBM z/VSE host using TCP/IP and the programs that enable the connection and interchange. You can run z/VSE with the following TCP/IP products:

- TCP/IP for VSE/ESA (5686-A04), which supports the IPv4 protocol.
- IPv6/VSE (5686-BS1), which supports the IPv4 and the IPv6 protocol.

IBM provides extensions that can be used with these programs. They are described in detail and include the LE/VSE C socket interface, the EZASMI macro interface, the EZASOKET call interface, the Linux Fast Path to System z[®] function, the CICS[®] Listener Support and more. Be aware that these extensions in its external interfaces might describe more functionality than really provided by the TCP/IP stacks. Check the individual function descriptions for such exceptions.

This publication is divided into 5 parts:

- Part 1, "Using TCP/IP for VSE/ESA," on page 1 describes how to setup and use the TCP/IP for VSE/ESA (5686-A04) product.
- Part 2, "Using IPv6/VSE," on page 39 gives an overview on the IPv6/VSE (5686-BS1) product.
- Part 3, "Programming Interfaces," on page 45 describes the various connection methods to a z/VSE host and how to interchange data with the system.
- Part 4, "Using Fast Path to Linux," on page 385 describes the Fast Path to Linux on System z and z/VSE - z/VM IP Assist functions.
- Part 5, "CICS Listener Support," on page 439 shows how to configure the CICS Listener Support.

Read this First!

Part 1. Using TCP/IP for VSE/ESA

Chapter 1. Overview

TCP/IP is a communication facility that permits bidirectional communication between VSE-based software and software running on other platforms equipped with TCP/IP.

Before using the TCP/IP for VSE/ESA program read the following very carefully.

Documentation for the TCP/IP for VSE/ESA (5686-A04) Program

The product description of the TCP/IP for VSE/ESA 1.5 product (IBM product number 5686-A04) is available in PDF format on the z/VSE Homepage and in the online collection kit that is available in compressed format for download from the IBM publication center (SK3T-8348).

The TCP/IP for VSE/ESA product documentation consists of 6 publications with the original product description from Connectivity Systems Inc., the provider of the TCP/IP for VSE product, plus one publication describing the setup of the TCP/IP for VSE/ESA product IBM provides – this publication.

The 7 publications are as follows:

- *z/VSE TCP/IP Support* (this publication)
- *TCP/IP for VSE 1.5 Installation Guide*
- *TCP/IP for VSE 1.5 User's Guide*
- *TCP/IP for VSE 1.5 Commands*
- *TCP/IP for VSE 1.5 Programmer's Reference*
- *TCP/IP for VSE 1.5 Messages and Codes*
- *TCP/IP for VSE 1.5 Optional Features*

You can use the Adobe Acrobat Reader to view and print these publications. If you do not already have Acrobat Reader installed, or if you need information on installing and using Acrobat Reader, see the Adobe website at <http://www.adobe.com>.

The Secure Sockets Layer (SSL) setup used in z/VSE is described in *z/VSE Administration*.

You can find more information about the support of HiperSockets and OSA Express® in *z/VSE Planning*.

General Considerations on the TCP/IP for VSE/ESA Program Setup

As described above the product documentation for the TCP/IP for VSE/ESA 1.5 product (IBM product number 5686-A04) is available on the z/VSE DVD collection (SK3T-8348) in PDF format only. This DVD contains 6 original publications on the product from Connectivity Systems Inc., the provider of the TCP/IP for VSE/ESA product.

When you read the product description from Connectivity Systems Inc. (CSI) note the following differences when using the TCP/IP for VSE/ESA product from IBM:

- The 'TCP/IP for VSE/ESA' product from IBM (product number 5686-A04) is in general the same as the product 'TCP/IP for VSE' from CSI; the differences and additional functions exploiting TCP/IP for VSE/ESA are listed in this publication.
- The TCP/IP for VSE/ESA product provided by Connectivity Systems Incorporated is now installed in library PRD2.TCPIPC and no longer in PRD1.BASE. Therefore, batch jobs that use TCP/IP for VSE/ESA must now have PRD2.TCPIPC as the first library in their library search chain.
- TCP/IP for VSE/ESA from IBM is preinstalled in the PRD2.TCPIPC library; therefore all references in the documentation from CSI which describe product installation tasks (for example restoring the product) do not apply.
- TCP/IP for VSE/ESA from IBM uses a specific key verification procedure. How to install the IBM product key for TCP/IP for VSE/ESA is described below.
- There are two types of REXX support for TCP/IP for VSE/ESA available:
 - The REXX Socket API support within REXX/VSE. The description of this REXX Socket API is in the online manual *REXX/VSE Reference*.
 - The REXX support within TCP/IP for VSE/ESA (for example REXX Socket API). The documentation of this REXX support can be found in the *TCP/IP for VSE 1.5 Programmer's Reference* manual.
- The CAF (CICS Access Facility) of TCP/IP for VSE is not yet available from IBM; therefore all references to CAF do not apply.
- Connectivity Systems Inc. provides interim service to their TCP/IP for VSE product using 'alpha and beta service packs'. These service packs contain updates to the TCP/IP for VSE product which are not officially available from IBM.

If a customer is using an 'alpha' or 'beta' version of a CSI service pack, the VSE-TCP/IP environment has to be considered in general as 'unsupported' for purposes of interfacing with IBM products that exploit TCP/IP for VSE/ESA. This is true regardless of whether the customer is an IBM TCP/IP customer or a CSI TCP/IP customer. Further information can be found in Information APAR II11836.

When CSI provides such a service pack in production mode IBM provides a PTF for the same service pack. Go to the z/VSE® Home page for the latest TCP/IP for VSE/ESA APARs and PTFs: <http://www.ibm.com/systems/z/os/zvse/support/tcpip.html>.

- In case of problems with TCP/IP for VSE see the Connectivity Systems Inc. web page for TCP/IP support at <http://www.csi-international.com/> for any available fix that might resolve your problem.
- When you have licensed the TCP/IP for VSE/ESA product from IBM you have to use the normal IBM service channel to get support in case of problems. Tapes and problem documentation have to be provided to the appropriate service center. Therefore special Technical Support Considerations in CSI's documentation do not apply.

The Demo Mode for TCP/IP for VSE/ESA

TCP/IP for VSE/ESA as shipped to all customers is configured to run in demonstration mode. Demonstration mode is intended to be used to configure and test TCP/IP for VSE/ESA in customer environments and it is not suitable for production use. TCP/IP for VSE/ESA has the following characteristics while running in demonstration mode:

- TCP/IP for VSE/ESA will shut itself down every hour.

- You are limited to one (1) concurrent FTP session.
- You are limited to one (1) concurrent TELNET session.
- You are limited to one (1) concurrent Line Printer Daemon.
- You can only establish one (1) concurrent session with the TCP/IP for VSE/ESA web server.

NFS, GPS, and SSL are not usable in demonstration mode.

You can enable production use of TCP/IP for VSE/ESA by installing a product key that you can obtain from IBM after licensing the product.

To run the TCP/IP for VSE/ESA in demo mode, a VSE partition of at least 20 MB size is required.

Note that you must run TCP/IP in a VSE partition with high priority. As TCP/IP is like VTAM[®] a timing dependent product, it is recommended to use a partition with a PRTY about equal to VTAM.

Supplying the Product Key

TCP/IP for VSE/ESA, the native TCP/IP solution for VSE, is preinstalled in the z/VSE base, and is available as an optional-priced IBM program. IBM has licensed this program from Connectivity Systems Incorporated.

The Application Pak, which is the basic TCP/IP for VSE/ESA function set, requires a key. The Network File System (NFS) feature and the General Print Server (GPS) feature are optional-priced additional applications on top of the TCP/IP for VSE/ESA Application Pak and require a separate key.

SSL for VSE is part of TCP/IP for VSE/ESA and is also key protected. It cannot be used in demo mode, because it is used in conjunction with the Application Pak for TCP/IP for VSE/ESA.

The different keys for the Application Pak, NFS or GPS, are delivered to the customer when the product is licensed. To license the TCP/IP for VSE/ESA product and its features, you have to use the normal IBM ordering process using CFSW for example.

The Application Pak includes the Socket Application Programming Interface (API), the TCP/IP Protocol stack and handles all layers of the TCP/IP communication from the physical layer up to the application functions. It also includes the following TCP/IP Applications:

- TN3270 server and Telnet/TN3270 client
- FTP server and client
- Web server (HTTP daemon)
- Line Printer Requestor (LPR) and Line Printer Daemon (LPD)

NFS and GPS are not included in the Application Pak.

TCP/IP for VSE/ESA is shipped with a "demonstration mode" product key. This key is installed into the sublibrary PRD2.TCPIPC together with the product's phases. Prior to running TCP/IP for VSE/ESA in production mode, it is necessary to supply a permanent product key. This product key is based upon the license you have signed. It is recommended that you place your production product key in the sublibrary allocated to "configuration" data (for example PRD2.CONFIG)

and that this sublibrary is first in the LIBDEF search order. In this way, application of maintenance or a product reinstallation will not overlay your production key.

The product enabling is driven by two different phases which can be generated using the job streams shown in the examples below.

If you plan to use NFS (Network File System), you have to install a separate product key for NFS in addition to the key for the Application Pak.

If you plan to use GPS (General Print Server), you have to install a separate product key for GPS in addition to the key for the Application Pak.

Installing Product Keys

```
// JOB KEY
// LIBDEF *,SEARCH=PRD2.TCPIPC
// LIBDEF PHASE,CATALOG=PRD2.CONFIG
// OPTION CATAL
// EXEC ASMA90,SIZE=(ASMA90,50K)
        PRODKEY 1234-5678-9012-3456-7890 /* APPLICATION PAK */
        PRODKEY 1234-4567-9123-5678-9012 /* NFS */
        PRODKEY 3456-7890-1234-5678-9012 /* GPS */
        END
/*
// EXEC LNKEDT
/&
```

Defining Customer Information

```
// JOB TCPCUS
// LIBDEF *,SEARCH=(PRD2.TCPIPC)
// LIBDEF PHASE,CATALOG=PRD2.CONFIG
// OPTION CATAL
// EXEC ASSEMBLY
        CUSTDEF DEFINE,                                X
                NAME='IBM z/VSE Development',          X
                NUMBER=C123-456-7890
        END
/*
// EXEC LNKEDT
/&
```

Note:

1. In the preceding example, PRD2.CONFIG is the name of the library into which the configuration data of TCP/IP for VSE/ESA is being installed.
2. Once you have completed a license agreement for the software, you can replace the string shown in the example with a real product key. The keys that appear here in this example are only for illustrative purposes.
3. The customer number used by TCP/IP for VSE/ESA (as shown in the second example above) is not the IBM customer number. The customer number to be used in the CUSTDEF macro is provided on the same memo where the key for the product is specified.

Migration Considerations

TCP/IP for VSE/ESA preinstalled with z/VSE can be used with TCP/IP for z/OS® and TCP/IP for z/VM in a VM/VSE environment. Either product can be used as a gateway to an intranet or the internet in general. Check your TCP/IP documentation for the configuration necessary to couple to those products. For

example, you could use a CTCA connection. TCP/IP for VSE/ESA could also be used to connect to any TCP/IP product on a non-VSE system, as long as this TCP/IP implementation follows the TCP/IP standards.

If you chose to purchase TCP/IP for VSE/ESA from IBM and intend to use it concurrently with a different non-IBM/non-Connectivity Systems TCP/IP-implementation on the z/VSE system, you are running in an environment which has not been tested explicitly. In this case both products must be carefully configured to avoid any problems. For example, the products may use the same file names where it is not predictable how they will behave if the LIBDEF chains are not properly set up (for example duplicate SOCKET.H C language header file).

If you decide to run any other than the preinstalled TCP/IP together with z/VSE , run the IBM supplied delete job (see skeleton DELTCPIP in ICCF library 59) to make sure that this TCP/IP does not interfere with the preinstalled TCP/IP for VSE/ESA.

If you are migrating to TCP/IP for VSE/ESA from any other TCP/IP product than the one from Connectivity Systems, follow the configuration steps as supplied with the product, and use your current TCP/IP specific parameters like the host IP address to ease the product setup.

If you have been using TCP/IP for VSE from Connectivity Systems or one of its distributors before migrating to TCP/IP for VSE/ESA on z/VSE, consider the following:

- z/VSE preinstalls TCP/IP for VSE/ESA in the PRD2.TCPIPC system library. If you have followed the installation recommendation from Connectivity Systems and installed TCP/IP in its private sublibrary, remove this sublibrary from your default LIBDEF chains.
- TCP/IP for VSE/ESA is stored in PRD2.TCPIPC. Jobs referring to any other TCP/IP sublibrary need to be changed. This includes the TCP/IP startup job itself, as well as any job performing for example LPR, FTP, or TELNET sessions from within a batch job. If you perform TCP/IP related development yourself, the respective development procedures may also be affected.
- If you do not have stored your TCP/IP specific configuration files like IPINITxx.L and NETWORK.L in a separate sublibrary (as recommended by Connectivity Systems), you should move these modified files into PRD2.CONFIG to ensure they will not be replaced with the next z/VSE service refresh. This includes any enhancements/modifications you may have done to your IPXLATE translation phase as well as Telnet related terminal definitions in the supplied TCPAPPL source book or specific replacement of it.
- Rename the PRODKEYS phase you had assembled with product keys from Connectivity Systems, and generate a new PRODKEYS phase with the product keys as supplied by IBM.

While product keys from Connectivity Systems only require to generate phase PRODKEYS, IBM supplied keys additionally require the generation of phase CUSTDEF as described under "Defining Customer Information" on page 6. The validation of the IBM supplied keys requires that the LE/VSE C runtime environment must be accessible. Include PRD2.SCEEBASE in the LIBDEF definition of the startup job of TCP/IP for this purpose.

- Your TCP/IP defined virtual file system may have changed by migrating to z/VSE. Update your IPINITxx.L configuration member accordingly.
- TCP/IP for VSE/ESA provided with z/VSE is fully MSHP controlled, i.e. you must not apply any Connectivity Systems Inc. provided service pack to the

system as you may have done with your previous product setup. Instead you should only install IBM supplied PTFs. Otherwise you may be running in an unsupported environment. Applying other kind of fixes than PTFs may downgrade your system and cause unpredictable effects.

- If you have self written TCP/IP for VSE/ESA applications:
 - you may need to re-assemble your assembler application(s) if they were using the TCP/IP for VSE/ESA *SOCKET* macro. This macro contains inline code which may have been refreshed with IBM's TCP/IP for VSE/ESA.
 - you may need to relink your application(s) if they had been using the BSD-C socket interface as provided with the product or when using the product's preprocessor for resolving EXEC TCP source statements in COBOL, PL/I or assembler programs. This may be necessary because the IPNxxxx.OBJ files linked to the application may have been serviced.

Chapter 2. TCP/IP for VSE/ESA Configuration

How TCP/IP for VSE/ESA is Installed

TCP/IP for VSE/ESA is preinstalled with z/VSE in the PRD2.TCPIPC library. Do not keep any personalized information, for example the key and customer definition or the TCP/IP startup member in PRD2.TCPIPC. This is necessary because some modules might be serviced by applying a PTF or by a system refresh, a Fast Service Upgrade (FSU).

TCP/IP for VSE/ESA Partition Startup

z/VSE defines the default partition F7 to TCP/IP for VSE/ESA. A default partition startup member TCPSTART.Z can be found in PRD2.TCPIPC. You can adjust it according to your configuration and put it into the VSE/POWER RDR queue using the DTRIINIT utility (see the following example). You can store the updated member in PRD2.CONFIG.

The default partition for TCP/IP is F7 and is 20 MB per default. It is highly recommended to use TCP/IP for VSE/ESA in a partition with at least 30 MB to benefit from the 31-bit exploitation of the product.

Note that TCP/IP requires a VSE partition with a high priority. As for any other timing dependent product such as VTAM it is therefore highly recommended to use a partition with a PRTY about equal to VTAM. This is especially true if, for example, TCP/IP has to service CICS for the use of Telnet or MQSeries®.

Example

The following job stream can be used to load the TCP/IP for VSE/ESA startup member TCPSTART.Z to the POWER® RDR queue:

```
* $$ JOB JNM=TCPLOAD,CLASS=A,DISP=D
* $$ LST CLASS=A,DISP=D
// JOB TCPLOAD LOAD TCPIP STARTUP INTO POWER
// LIBDEF *,SEARCH=IJSYSRS.SYSLIB
// EXEC DTRIINIT
ACCESS PRD2.TCPIPC
LOAD TCPSTART.Z
/*
/&
* $$ E0J
```

TCPSTART.Z looks as follows:

```
* $$ JOB JNM=TCPSTART,CLASS=7,DISP=K
* $$ LST CLASS=A,DISP=D
// JOB TCPIP
// LIBDEF *,SEARCH=(PRD2.CONFIG,PRD2.TCPIPC,PRD2.SCEEBASE)
// EXEC PROC=DTRICCF
// SETPFIX LIMIT=(400K,2100K)
// EXEC IPNET,SIZE=IPNET,PARM='ID=00,INIT=IPINIT00',DSPACE=2M
/&
* $$ E0J
```

TCP/IP for VSE/ESA Configuration

Note:

1. In the example PRD2.TCPIPC is the library where TCP/IP for VSE/ESA is preinstalled and PRD2.CONFIG is the library where you have placed your installation-dependent values (initialization member and authorization code).
2. The PRD2.SCEEBASE library contains the LE/VSE C runtime environment and is necessary for IBM product key verification.
3. Be sure that the LIBDEF statement specifies “*”. If you specify “phase”, TCP/IP for VSE/ESA is unable to locate the initialization member IPINIT00.L.
4. If you do not use the system supplied TCP/IP startup job, make sure that the LIBDEF definition includes the LE/VSE C runtime contained in PRD2.SCEEBASE. This is essential for proper IBM product key validation.

Configuring CICS

TCP/IP for VSE/ESA includes several CICS-based clients. These clients provide CICS users with the ability to use TCP/IP for example to:

- Logon (from CICS) to other platforms and applications via Telnet. For example, a user could logon to a UNIX system from CICS.
- Initiate a file transfer between the TCP/IP for VSE/ESA FTP server and a remote FTP server.
- Printing files using LPR.
- Check the network connection using the Ping client.

Setup CICS

- Ensure that TCP/IP for VSE/ESA is set in your CICS partition's search chain. This may be accomplished by modifying your CICS startup JCL as follows:

```
// LIBDEF *,SEARCH=(lib,lib,PRD2.TCPIPC)
```
- Define the Programs and Transactions to your CICS which should be used with TCP/IP.

Note:

1. The use of group “TCPIP” and list “VSELIST” is arbitrary. You can make any adjustments that your site requires.
2. The DFHPPTIP.A shipped with the TCP/IP for VSE/ESA product looks as follows:

DFHPPTIP — CICS Processing Program Table

```
PPTIP    TITLE    'DFHPPTIP - Cics Processing Program Table'
DFHPPT   TYPE=INITIAL,                                *
          SUFFIX=IP
DFHPPT   TYPE=ENTRY,                                  Entry                               *
          PROGRAM=TELNET01,                             Program Identification              *
          RSL=PUBLIC,                                   Public Program                      *
          PGMLANG=ASSEMBLER                             Assembler
DFHPPT   TYPE=ENTRY,                                  Entry                               *
          PROGRAM=FTP01,                                 Program Identification              *
          RSL=PUBLIC,                                   Public Program                      *
          PGMLANG=ASSEMBLER                             Assembler
DFHPPT   TYPE=ENTRY,                                  Entry                               *
          PROGRAM=CLIENT01,                             Program Identification              *
          RSL=PUBLIC,                                   Public Program                      *
          PGMLANG=ASSEMBLER                             Assembler
DFHPPT   TYPE=FINAL
END
```

3. The DFHPCTIP.A shipped with the TCP/IP for VSE/ESA product looks as follows:

DFHPCTIP — CICS Transaction Table

PCTIP	TITLE		
	'DFHPCTIP - Cics Transaction Table'		
DFHPCT	TYPE=INITIAL,		*
	SUFFIX=IP		
DFHPCT	TYPE=ENTRY,	Entry	*
	TRANSID=TELN,	Transaction Name	*
	PROGRAM=TELNET01,	Program Identification	*
	RSL=PUBLIC	Public	
DFHPCT	TYPE=ENTRY,	Entry	*
	TRANSID=teln,	Transaction Name	*
	PROGRAM=TELNET01,	Program Identification	*
	RSL=PUBLIC	Public	
DFHPCT	TYPE=ENTRY,	Entry	*
	TRANSID=TELC,	Transaction Name	*
	PROGRAM=TELNET01,	Program Identification	*
	RSL=PUBLIC	Public	
DFHPCT	TYPE=ENTRY,	Entry	*
	TRANSID=TELW,	Transaction Name	*
	PROGRAM=TELNET01,	Program Identification	*
	RSL=PUBLIC	Public	
DFHPCT	TYPE=ENTRY,	Entry	*
	TRANSID=TELR,	Transaction Name	*
	PROGRAM=TELNET01,	Program Identification	*
	RSL=PUBLIC	Public	
DFHPCT	TYPE=ENTRY,	Entry	*
	TRANSID=FTP,	Transaction Name	*
	PROGRAM=FTP01,	Program Identification	*
	RSL=PUBLIC	Public	
DFHPCT	TYPE=ENTRY,	Entry	*
	TRANSID=ftp,	Transaction Name	*
	PROGRAM=FTP01,	Program Identification	*
	RSL=PUBLIC	Public	
DFHPCT	TYPE=ENTRY,	Entry	*
	TRANSID=FTPC,	Transaction Name	*
	PROGRAM=FTP01,	Program Identification	*
	RSL=PUBLIC	Public	
DFHPCT	TYPE=ENTRY,	Entry	*
	TRANSID=FTPW,	Transaction Name	*
	PROGRAM=FTP01,	Program Identification	*
	RSL=PUBLIC	Public	
DFHPCT	TYPE=ENTRY,	Entry	*
	TRANSID=FTPR,	Transaction Name	*
	PROGRAM=FTP01,	Program Identification	*
	RSL=PUBLIC	Public	
DFHPCT	TYPE=ENTRY,	Entry	*
	TRANSID=LPR,	Transaction Name	*
	PROGRAM=CLIENT01,	Program Identification	*
	RSL=PUBLIC	Public	
DFHPCT	TYPE=ENTRY,	Entry	*
	TRANSID=lpr,	Transaction Name	*
	PROGRAM=CLIENT01,	Program Identification	*
	RSL=PUBLIC	Public	
DFHPCT	TYPE=ENTRY,	Entry	*
	TRANSID=PING,	Transaction Name	*
	PROGRAM=CLIENT01,	Program Identification	*
	RSL=PUBLIC	Public	
DFHPCT	TYPE=ENTRY,	Entry	*
	TRANSID=ping,	Transaction Name	*
	PROGRAM=CLIENT01,	Program Identification	*
	RSL=PUBLIC	Public	
DFHPCT	TYPE=ENTRY,	Entry	*
	TRANSID=TCPC,	Transaction Name	*
	PROGRAM=CLIENT01,	Program Identification	*
	RSL=PUBLIC	Public	
DFHPCT	TYPE=ENTRY,	Entry	*
	TRANSID=TCPW,	Transaction Name	*
	PROGRAM=CLIENT01,	Program Identification	*

TCP/IP for VSE/ESA Configuration

```

RSL=PUBLIC                               Public
DFHPCT  TYPE=ENTRY,                       Entry          *
        TRANSID=TCPR,                     Transaction Name *
        PROGRAM=CLIENT01,                 Program Identification *
        RSL=PUBLIC                         Public
DFHPCT  TYPE=FINAL
END
```

Example for CICS/TS 1.1

Use member IPNCSD.Z to define the programs and transactions to your CICS. Additionally a member IPNCSDUP.Z is available to use IPNCSD.Z. Following is an example of IPNCSDUP.Z:

```

* $$ JOB JNM=IPNCSDUP,CLASS=0,DISP=D
// JOB IPNCSDUP
* SHUT DOWN CICS FIRST
// PAUSE CLOSE DFHCSD FILE IF CICS IS UP : CEMT SE FI(DFHCSD) CLOSE
/*
// LIBDEF *,SEARCH=(PRD2.CONFIG,PRD1.BASE,PRD2.SCEEBASE)
// EXEC DFHCSDUP,SIZE=600K          INIT AND LOAD CICS
DELETE GROUP(TCPIP)
* $$ SLI MEM=IPNCSD.Z,S=(PRD1.BASE)
ADD GROUP(TCPIP) LIST(VSELIST)
LIST ALL
/*
/&
* $$ E0J
```

Note:

1. The use of group "TCPIP" and list "VSELIST" is arbitrary. You can make any adjustments that your site requires.
2. The IPNCSD.Z shipped with the TCP/IP for VSE/ESA product looks as follows:

IPNCSD.Z shipped with TCP/IP for VSE/ESA

```

*-----*
*   FOLLOWING ARE THE PPT ENTRIES REQUIRED FOR TCP/IP for VSE/ESA   *
*-----*
DEFINE PROGRAM(TELNET01) GROUP(TCPIP)
        LANGUAGE(ASSEMBLER)
DEFINE PROGRAM(FTP01)   GROUP(TCPIP)
        LANGUAGE(ASSEMBLER)
DEFINE PROGRAM(CLIENT01) GROUP(TCPIP)
        LANGUAGE(ASSEMBLER)
*-----*
*   FOLLOWING ARE THE PCT ENTRIES REQUIRED FOR TCP/IP for VSE/ESA   *
*-----*
DEFINE TRANSACTION(TRAC)  GROUP(TCPIP)
        PROGRAM(CLIENT01)
DEFINE TRANSACTION(trac)  GROUP(TCPIP)
        PROGRAM(CLIENT01)
DEFINE TRANSACTION(REXE)  GROUP(TCPIP)
        PROGRAM(CLIENT01)
DEFINE TRANSACTION(rexe)  GROUP(TCPIP)
        PROGRAM(CLIENT01)
DEFINE TRANSACTION(DISC)  GROUP(TCPIP)
        PROGRAM(CLIENT01)
DEFINE TRANSACTION(disc)  GROUP(TCPIP)
        PROGRAM(CLIENT01)
DEFINE TRANSACTION(EMAI)  GROUP(TCPIP)
        PROGRAM(CLIENT01)
DEFINE TRANSACTION(ema)   GROUP(TCPIP)
        PROGRAM(CLIENT01)
DEFINE TRANSACTION(PING)  GROUP(TCPIP)
```

```

        PROGRAM(CLIENT01)
DEFINE TRANSACTION(ping)  GROUP(TCPIP)
        PROGRAM(CLIENT01)
DEFINE TRANSACTION(TELN)  GROUP(TCPIP)
        PROGRAM(TELNET01)
DEFINE TRANSACTION(te1n)  GROUP(TCPIP)
        PROGRAM(TELNET01)
DEFINE TRANSACTION(TELC)  GROUP(TCPIP)
        PROGRAM(TELNET01)
DEFINE TRANSACTION(TELW)  GROUP(TCPIP)
        PROGRAM(TELNET01)
DEFINE TRANSACTION(TELR)  GROUP(TCPIP)
        PROGRAM(TELNET01)
DEFINE TRANSACTION(FTP)   GROUP(TCPIP)
        PROGRAM(FTP01)
DEFINE TRANSACTION(ftp)   GROUP(TCPIP)
        PROGRAM(FTP01)
DEFINE TRANSACTION(FTPC)  GROUP(TCPIP)
        PROGRAM(FTP01)
DEFINE TRANSACTION(FTPW)  GROUP(TCPIP)
        PROGRAM(FTP01)
DEFINE TRANSACTION(FTPR)  GROUP(TCPIP)
        PROGRAM(FTP01)
DEFINE TRANSACTION(TCPC)  GROUP(TCPIP)
        PROGRAM(CLIENT01)
DEFINE TRANSACTION(TCPW)  GROUP(TCPIP)
        PROGRAM(CLIENT01)
DEFINE TRANSACTION(TCPR)  GROUP(TCPIP)
        PROGRAM(CLIENT01)
DEFINE TRANSACTION(LPR)   GROUP(TCPIP)
        PROGRAM(CLIENT01)
DEFINE TRANSACTION(lpr)   GROUP(TCPIP)
        PROGRAM(CLIENT01)
*-----*
*                               END OF TCP/IP MEMBER                               *
*-----*

```

HTMLINST.Z

To interchange Hyper Text Markup Language (HTML) documents, HTTP is used. An HTML document is a file that contains printable text, interspersed with HTML “tags” that describe the document to be displayed. Additional elements of HTML allow you to include links to other documents, embedded graphics, and special effects.

TCP/IP for VSE/ESA provides special HTML files for security reasons:

- PASSWORD.HTML
- VIOLATED.HTML
- BLANKING.HTML

The member HTMLINST.Z in PRD2.TCPIPC contains a job stream which generates default members of these special HTML files. The member HTMLINST.Z can be loaded into the VSE/POWER RDR queue using the DTRIINIT utility. An example is shown in the following.

Example

```

* $$ JOB JNM=HTMLLOAD,CLASS=A,DISP=D
* $$ LST CLASS=A,DISP=D
// JOB HTMLLOAD LOAD HTMLINST.Z INTO POWER
// LIBDEF *,SEARCH=IJSYSRS.SYSLIB
// EXEC DTRIINIT
ACCESS PRD2.TCPIPC

```

TCP/IP for VSE/ESA Configuration

```
LOAD HTMLINST.Z  
/*  
/&  
* $$ E0J
```

Details on how to use the single HTML members can be found in the section 'Security' of chapter 'Configuring the HTTP Daemon' in the *TCP/IP for VSE 1.5 Installation Guide*.

Chapter 3. TCP/IP for VSE/ESA Configuration Dialogs

It is important that you have read Chapter 1, “Overview,” on page 3 before you start configuring your system for TCP/IP for VSE/ESA!

Before you can use *TCP/IP for VSE/ESA*, some configuration work should be done.

This can be done either manually by providing the necessary definitions in the related TCP/IP for VSE/ESA library members and definition jobs. To assist you some configuration members are provided, for example TCPSTART.Z (see “TCP/IP for VSE/ESA Partition Startup” on page 9 for details), TCPAPP00.B (sample VTAM definitions for Telnet daemons), and IPINIT00.L (sample TCP/IP for VSE/ESA initialization member). Or you can use the “Configuring TCP/IP Using the IUI-based Configuration Dialog” on page 16.

Prior to running TCP/IP for VSE/ESA in production mode, you must have installed your product key as described under “Supplying the Product Key” on page 5.

Configuring TCP/IP Using the Configuration Dialogs

Before starting TCP/IP for VSE/ESA, you must provide information about your configuration. The following can be specified:

General information

Some general configuration is necessary. For example, you must specify the HOST IP address of TCP/IP for VSE/ESA.

Links You need to identify each device, controller, and connection mechanism that TCP/IP for VSE/ESA will use for external communication.

Daemons

A definition of the service Daemons must be provided. Daemons are the routines that provide services to the end user. For example, FTP is a service daemon that provides access to the VSE file system.

Routing information

Depending on your configuration, it may be necessary to define routing information to the TCP/IP for VSE/ESA product. This information is used to control connections with other TCP/IP platforms.

How To Do It

All configuration information is specified by a series of console operator commands. For this reason, you may simply start the TCP/IP for VSE/ESA product and then provide all configuration data by command or more conveniently — you may place your configuration commands in an initialization library member IPINITxx.L. This is described in detail in the *TCP/IP for VSE 1.5 Installation Guide*.

Most conveniently, you can use the IUI-based Configuration Dialog described in “Configuring TCP/IP Using the IUI-based Configuration Dialog” on page 16.

Performing Configuration Work

A default member IPINIT00.L is shipped with z/VSE in VSE library PRD2.TCPIPC. It contains many configuration parameters set to their default values. This a good starting point in developing your own configuration.

Your initialization member should be placed in a sublibrary that you have reserved for configuration data, for example PRD2.CONFIG. Thus, your member is not accidentally replaced during application of maintenance to the TCP/IP product or to the z/VSE system.

Configuring TCP/IP Using the UI-based Configuration Dialog

The Interactive Interface has been enhanced with the **TCP/IP Configuration** panel (Fast Path245) to help you configure your TCP/IP environment. After completing your definitions for the TCP/IP parameters shown in the panel below, you must press PF5 (PROCESS) to create a job stream which updates the related configuration members such as IPINIT00.L in system library PRD2.CONFIG. If no member exists in PRD2.CONFIG, the system default member IPINIT00 from PRD2.TCPIPC is used. The following description shows a list of panels where you can define the values for the parameters shown in the *TCP/IP Configuration* panel (Figure 1) below. Press PF5 (PROCESS) in this panel after you have entered the required values.

If TELNET daemons are added, press PF9=VTAM to update member TCPAPP00.B (see page 21 for more information).

```
CON$SEL                                TCP/IP CONFIGURATION

Enter the required data and press ENTER.

To modify one or more of the following TCP/IP parameters,
place a 1 next to it.

-   SET          Modify SET IPADDR or SET MASK command
-   LINK         Modify DEFINE LINK command
-   ADAPTER      Modify DEFINE ADAPTER command
-   ROUTE        Modify DEFINE ROUTE command
-   TELNETD     Modify DEFINE TELNETD command

PF1=HELP      2=REDISPLAY  3=END          5=PROCESS
                9=VTAM
```

Figure 1. TCP/IP Configuration Panel CON\$SEL

If SET is selected, the *TCP/IP Configuration: Set IPADDR and MASK* panel CON\$SIP is displayed. The panel shows the already defined values for the SET IPADDR and SET MASK statement and for the DEFINE NAME statement for this z/VSE client. These values can be changed.

```

CONP$SIP          TCP/IP CONFIGURATION: SET IPADDR AND MASK

Enter the required data and press ENTER.

Specify the parameters for the SET IPADDR and the SET MASK command.

IPADDR.....      ___ ___ ___ ___  default network address
MASK.....        ___ ___ ___ ___  value of the mask
NAME.....        _____  this z/VSE client

PF1=HELP          2=REDISPLAY  3=END
    
```

Figure 2. TCP/IP Configuration Panel: Set IPADDR and MASK

If LINK is selected in the *TCP/IP Configuration* panel (Figure 1 on page 16), a list of all defined links is displayed in the *TCP/IP Configuration: Link List* panel.

```

CON$LNKS          TCP/IP CONFIGURATION: LINK LIST

Enter the required data and press ENTER.

OPTIONS: 1 = ADD LINK    2 = ALTER LINK    3 = ADD ROUTE
          4 = ADD ADAPTER 5 = DELETE LINK

OPT      LINKID      DEVICE  TYPE   DATAPATH  IPADDR      PORT
*        OSAFE       500    OSAX   502       F14FEL1
-
-
-
-
-
-
-
-
-
-
-

PF1=HELP          2=REDISPLAY  3=END                5=PROCESS
    
```

Figure 3. TCP/IP Configuration Panel: Link List

If option 1 ADD LINK is entered, the following panel is displayed.

Performing Configuration Work

```
CON$LNK          TCP/IP CONFIGURATION: LINK
Enter the required data and press ENTER.

LINK ID..... _____ Enter the unique name of the link.
TYPE..... _____ Specify the link type.
DEVICE..... _____ Enter the unit address at which the
network connection device resides.
DATAPATH..... _____ Enter the unit address of the data-
path.
IPADDR.....   _ _ _ _ Associated IP address.
OSAPORT..... _____ Specify the port name or OSAPORT.

PF1=HELP      2=REDISPLAY  3=END
```

Figure 4. TCP/IP Configuration: Link

If the input was correct, the dialog goes back to panel *TCP/IP Configuration: LINK LIST* (Figure 3 on page 17).

Select ADAPTER on panel *TCP/IP Configuration* (Figure 1 on page 16) to get the following panel (note that this panel is only possible for links of type OSA or 3172):

```
CON$APTS          TCP/IP CONFIGURATION: ADAPTER LIST
Enter the required data and press ENTER.

OPTIONS: 1 = ADD ADAPTER 2 = ALTER ADAPTER
         5 = DELETE ADAPTER

OPT      LINKID          TYPE      NUMBER
-        ELKEL           FDDI     01
-        F234567890123456 FODI     02
-
-

PF1=HELP      2=REDISPLAY  3=END          5=PROCESS
```

Figure 5. TCP/IP Configuration Panel: Adapter List

If option 1 ADD ADAPTER is entered, the following panel is displayed.

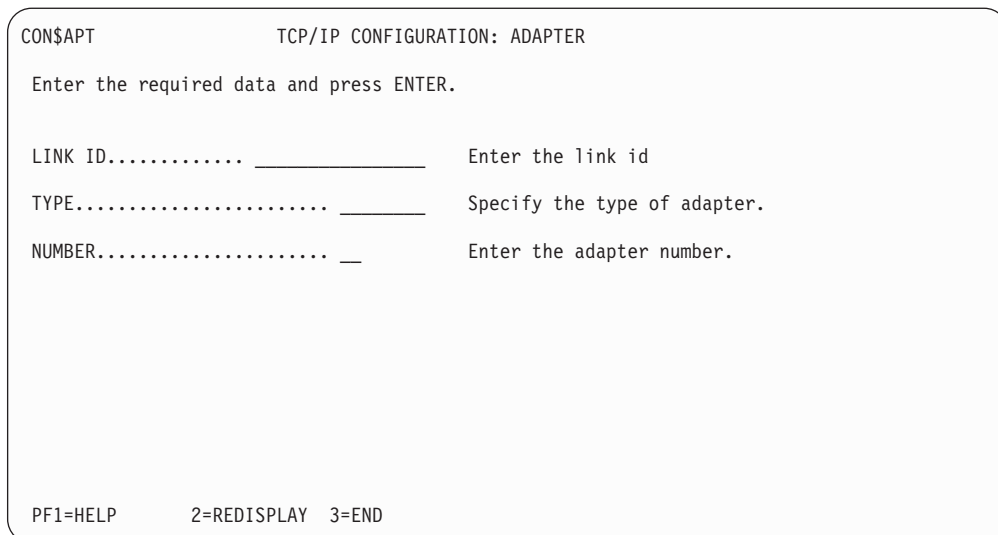


Figure 6. TCP/IP Configuration Panel: Adapter

If the input was correct, the dialog goes back to panel *TCP/IP Configuration: CON\$SEL* (Figure 1 on page 16).

You can now select ROUTE on panel *TCP/IP Configuration* (Figure 1 on page 16) and get the following panel:

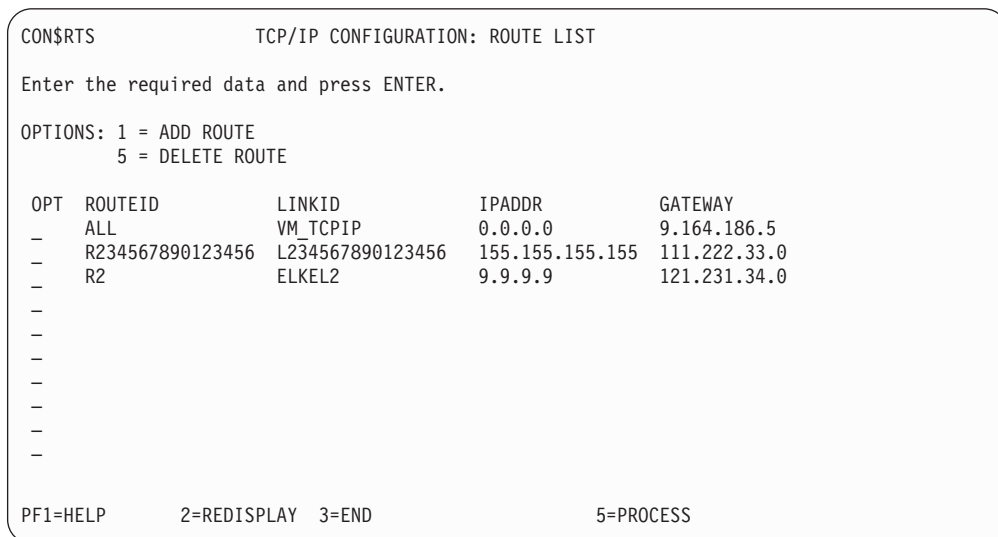


Figure 7. TCP/IP Configuration Panel: Route List

Enter Option 1=ADD ROUTE to get the DEFINE ROUTE panel. You can get the same panel by entering 3=ADD ROUTE in panel *TCP/IP Configuration: LINK LIST* (Figure 3 on page 17). In this case the LINKID has already been specified.

Performing Configuration Work

```
CON$ROUT          TCP/IP CONFIGURATION: DEFINE ROUTE
Enter the required data and press ENTER.

ROUTE ID.....   _____   Unique name of the route.
LINK ID.....    _____   Name of the associated link.
IPADDR.....     ___  ___  ___  ___   Associated IP address.
GATEWAY....     ___  ___  ___  ___   Full network address of a gateway

PF1=HELP          2=REDISPLAY  3=END
```

Figure 8. TCP/IP Configuration Panel: Define Route

If TELNET DAEMON is selected on the *TCP/IP Configuration* panel (Figure 1 on page 16), the TELNET LIST panel is displayed.

```
CON$TELS          TCP/IP CONFIGURATION: TELNET LIST
Enter the required data and press ENTER.

OPTIONS: 1 = ADD TELNET DAEMON  2 = ALTER TELNET DAEMON
        5 = DELETE DAEMON

OPT      DAEMONID      TARGET  TERMNAME  COUNT  LOGMODE
-        MYTEL        DBDCCICS T1000    20     U
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-

PF1=HELP          2=REDISPLAY  3=END          5=PROCESS
```

Figure 9. TCP/IP Configuration Panel: TELNET LIST

Enter option 1=ADD TELNET DAEMON to get the following panel:

```

CON$TELD          TCP/IP CONFIGURATION: TELNET DAEMON

Enter the required data and press ENTER.

DAEMON ID..... _____ Enter the unique name of the daemon

TARGET..... _____ Enter the name of the VTAM applica-
                        tion id you are connecting to

TERMNAME..... _____ Enter the VTAM LU name assigned to
                        the remote terminal.

COUNT..... _ Count for multiple telnet daemons.

LOGMODE..... _ VTAM LOGMODEs for the LU session.

PF1=HELP      2=REDISPLAY  3=END
    
```

Figure 10. TCP/IP Configuration Panel: TELNET DAEMON

When a TELNET daemon is added, the TERMname must be defined as a VTAM application. Therefore the book TCPAPP00 is included in the VTAM configuration member ATCCON00.B. The dialog offers the possibility to automatically add the application definition to TCPAPP00.B. The creation is triggered by pressing PF9=VTAM. You have to press PF5=PROCESS on the *TCP/IP Configuration* panel (Figure 1 on page 16) to activate the updates. Then a job is created which catalogs the updated TCP/IP startup member IPINIT00.L in PRD2.CONFIG and, if requested, the VTAM book TCPAPP00.B.

Chapter 4. Security Manager Exploitation by TCP/IP for VSE/ESA

This chapter shows how the Basic Security Manager's (BSM) functionality is exploited by TCP/IP for VSE/ESA. This implementation applies to z/VSE.

Using BSM Capabilities for TCP/IP Security Checks

TCP/IP allows various platforms to communicate with VSE. With this new openness for VSE, new security requirements arise. This is the reason why TCP/IP for VSE/ESA provides a number of functions to protect VSE resources (see *TCP/IP for VSE 1.5 Commands*).

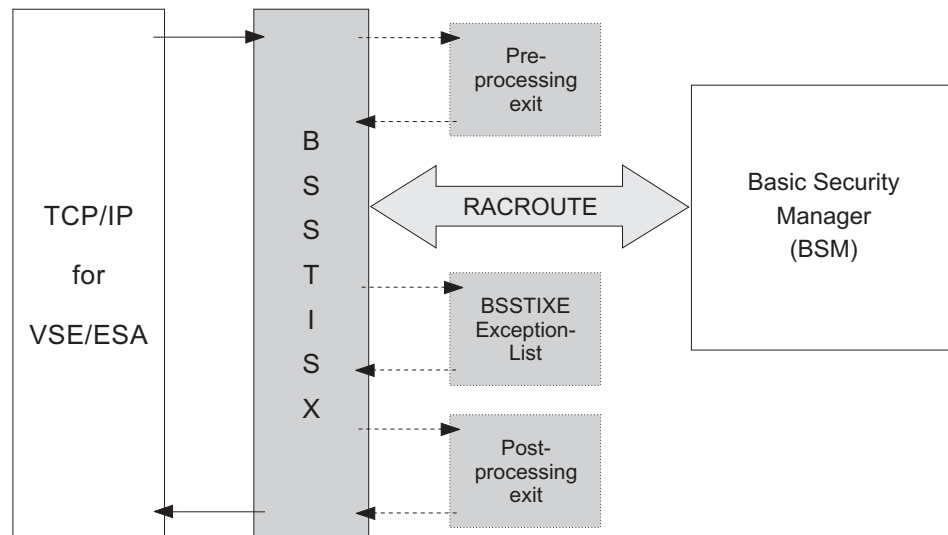
The security concept of TCP/IP for VSE/ESA is described in the *TCP/IP for VSE 1.5 Installation Guide*.

Details of VSE security can be found in *z/VSE Planning* and *z/VSE Administration*.

One of the security functions is the TCP/IP security exit point. It can be used via the TCP/IP provided code sample SECEXIT. But neither the TCP/IP internal security functions nor the sample exit code exploits the security functions of the VSE operating system, i.e. the Basic Security Manager (BSM). As a result, the customer has to define and administrate the user IDs and VSE resources twice, once in TCP/IP and once in the security system of the VSE operating system.

To improve this situation phase BSSTISX was introduced to replace the TCP/IP provided code sample SECEXIT.

The following figure shows the integration of BSSTISX as a link between TCP/IP and BSM.



The phase BSSTISX exploits the BSM capabilities. It issues RACROUTE requests to process user identification and user authentication, and resource access control for VSE files, libraries, and members. It also allows limited access control to POWER spool files and the SITE command.

Basic Security Manager

Access to POWER spool files will be allowed for administrators and users, where

- the user ID matches the FROM or TO user ID of the requested spool file, or
- TO=ANY was specified.

Note that the user ID assigned to ANONYMOUS does not have access to these files.

The SITE command can only be used by an administrator.

There are various other checks possible via the TCP/IP exit point, which are not covered by BSSTISX. Therefore BSSTISX provides a pre- and post-processing exit interface. Customers who need additional checks, can write their own pre-/post-processing routines for BSSTISX.

Exception List BSSTIXE

The exit BSSTISX rejects in general ALL access requests which could not be evaluated by this exit. But it might be necessary to not reject certain requests. These requests can be specified in this exception list by the customer. The exception list has to be assembled and linked as phase BSSTIXE (see SKEXCLST in library 59).

The IBM distributed phase and the related source member BSSTIXE.A is located in IJSYSRS.SYSLIB.

A request is defined by the SXBLOK fields SXTYPE and SXFTYPE. The SXBLOK describes the interface between TCP/IP and BSSTISX. The layout of the SXBLOK is distributed together with TCP/IP for VSE/ESA.

Warning: Define only requests in the exception list which could not be evaluated by BSSTISX. The requests defined in the exception list will NOT be security checked. Be sure to add ONLY these requests to the exception list which will not affect your security requirements of your installation. Instead of the exception list you could use the BSSTISX PRE and POST-PROCESSING EXITS to add your installation specific security checks.

Activation of The Security Exit

To activate the security exit, you have to enter the following TCP/IP commands:

DEFINE SECURITY,DRIVER=BSSTISX[,DATA='data']

The DEFINE SECURITY command loads the security exit BSSTISX.PHASE into the TCP/IP partition.

SET SECURITY =ON

The SET SECURITY=ON command activates the security processing and gives control to BSSTISX for initialization. BSSTISX loads additional parts into storage and initializes its control blocks according to the parameters specified in **data**. From now on TCP/IP passes information to the exit routine BSSTISX for verification.

SET SECURITY =ONX

Specify ONX if the security exit BSSTISX is to be used for FTPBATCH as well.

The parameter **DATA=** of the DEFINE SECURITY command contains the initialization parameter for BSSTISX. The syntax is described below.

DATA=[anonym_uid],[anonym_pwd],[preproc],[postproc]]]'

anonym_uid

Here you can specify a user ID, which is defined to BSM. Each time a client logs on with user ID ANONYMOUS your specified user ID and its access rights will be used.

anonym_pwd

With this parameter you can specify the password of the BSM defined user ID for user ANONYMOUS.

preproc

If you like to use a self-written preprocessing exit, specify here the name of your preprocessing exit phase.

postproc

For a self-written post-processing exit you have to specify here the name of your post-processing exit phase.

Deactivation of the Security Exit

To deactivate the security exit, you have to enter the following TCP/IP commands:

SET SECURITY=OFF

The SET SECURITY=OFF command stops the security processing and gives control to BSSTISX for cleanup and termination. BSSTISX clears its control blocks and frees the storage of its additional parts.

DELETE SECURITY

The DELETE SECURITY frees the security exit BSSTISX.PHASE.

Note: If you want to use a new version of the security exit, you should shut down TCP/IP and restart it again before you enter DEFINE SECURITY.

Using Pre- and Postprocessing Exits

The preprocessing exit gets control after the BSSTISX initialization and later on at the beginning of each request. The postprocessing exit gets control at the end of each request except the termination request. Both exits get the required information from the TCP/IP created SXBLOK.

The SXBLOK describes the interface between TCP/IP's exit point and the security exit. The mapping of the SXBLOK is shipped with TCP/IP for VSE/ESA. Be sure that you use the actual level of the SXBLOK of the TCP/IP you are using for the BSSTISX pre- and postprocessing exits.

Both, preprocessing exit and postprocessing exit have to be:

- reentrant
- AMODE(31)
- RMODE(24)

The general register usage is described below.

Register Settings for Preprocessing Exit

On entry:

R1 Address of SXBLOK

Basic Security Manager

- R13 Standard save area
- R14 Return address
- R15 Entry point of preprocessing exit phase

The preprocessing exit must restore registers prior to return. Register 15 shows the result:

On return:

- R15 = 0
BSSTISX should continue normal processing
- R15 = 'E0'x
BSSTISX should skip all checks and terminate with R15=0 (no violation)
- R15 = 4
BSSTISX should skip all checks and terminate with R15=4 (security violation)

Register Settings for Postprocessing Exit

On entry:

- R0 Current return code value of BSSTISX
- R1 Address of SXBLOK
- R2 Reason code from BSSTISX.
- R13 Standard save area
- R14 Return address
- R15 Entry point of postprocessing exit phase

The postprocessing exit must restore registers prior to return. Register 15 shows the result:

On return:

- R15 = 0
BSSTISX should terminate with R15=0 (no violation)
- R15 = n
BSSTISX should terminate with R15=n. n=4 indicates a security violation
- R15 = 4
BSSTISX should skip all checks and terminate with R15=4 (security violation)

Reason codes from BSSTISX for the post-processing exit:

- X'00' No specific reason code provided
- X'10' Access allowed - user is an administrator
- X'11' Access allowed by exception list entry
- X'12' Access allowed by RACROUTE AUTH request
- X'13' Access allowed - ICCF option specified
- X'14' Access allowed by pre-processing exit
- X'15' Access allowed - to be checked by OPEN

- X'16' Access always allowed by BSSTISX
- X'17' Access allowed by POWER. It is a from/to or ANY user.
- X'18' Access allowed by the right to open a master console.
- X'19' Access allowed because it is a \$NULL file request.
- X'20' Access denied - user is not an administrator
- X'21' Access denied - unsupported request
- X'22' Access denied by RACROUTE AUTH request
- X'23' Access denied due to option code 4
- X'24' Access denied due to option code 8
- X'25' Access denied due to internal error
- X'26' Access denied by pre-processing exit
- X'27' Access denied. It is not a read request to POWER.
- X'28' Access denied by POWER. It is not a from/to or ANY user.

Performance Hints

Depending on the TCP/IP usage, BSSTISX may have to issue a high number of user verifications with the same user IDs. For this condition it is useful to activate the BSM cache via:

```
MSG xx,DATA=DBSTARTCACHE
```

where xx stands for the partition ID of the security server partition (default is FB).

External Security Managers

The TCP/IP security exit BSSTISX can also be used together with External Security Managers (ESMs), if these ESMs support the RACROUTE requests issued by BSSTISX. CA-Top Secret (for example, distributed by CA Inc.) supports these RACROUTE calls.

Chapter 5. InfoPrint Manager Support of TCP/IP for VSE/ESA

The InfoPrint Manager (IPM) support allows to transfer print files in EBCDIC mode to the AIX® or Windows workstation where the IPM is running. Due to the EBCDIC transfer this support maintains the control characters in the document to be printed. The prerequisites for this support are described in “Software Prerequisites” on page 32.

The following table provides an overview of the support provided for the InfoPrint Manager (IPM) on AIX, Windows NT, Windows 2000 or Windows XP.

Support	LPR Script	\$\$ LST specification*
EBCDIC	SET INFOPRINT=YES/NO (or =ON/OFF)	n/a
Pagedef	SET PAGEDEF=pdef ¹	PAGEDEF=pdef ^{1, 2}
Formdef	SET FORMDEF=fdef ³	FORMDEF=fdef ^{3, 4}
Forms	SET FNO=fno ⁵	FNO=fno ⁵

¹ pdef= the maximum number of alphanumeric characters is 6

² The following definition is required in the POWER generation:
DEFINE L,PAGEDEF,1F,1,6,C

³ fdef= the maximum number of alphanumeric characters is 6

⁴ The following definition is required in the POWER generation:
DEFINE L,FORMDEF,1D,1,6,C

⁵ fno= the maximum number of alphanumeric characters is 4

* only usable with AUTOLPR

For ¹ and ³, the actual values can be checked in z/VSE with the POWER command D AUSTMT .

Note:

1. EBCDIC support is only available by using LPR/LPD.
2. If values are defined in \$\$ LST and SET, the values from the SET commands are used.
3. Supported alphanumeric characters are: A-Z, 0-9, #, @, \$

The following example shows how this support can be used:

```
* $$ JOB JNM=TRLTSTPR,CLASS=0,PRI=5,USER=TEST
* $$ LST CLASS=L,DEST=(*,ANY)
// JOB TRLTSTLPR *** LPR to AIX queue ***
// LIBDEF *,SEARCH=(PRD2.CONFIG,PRD2.TCPIPC,PRD2.SCEEBAE)
// EXEC CLIENT,PARM='APPL=LPR,ID=00'
SET HOST=9.66.110.67
SET PRINTER=ipheft
SET PAGEDEF=b111
SET FORMDEF=a222
SET INFOPRINT=YES
SET CC=YES
SET CRLF=UNIX
SET NOEJECT=ON
SET DISP=KEEP
PRINT POWER.LST.L.TRLUH003
QUIT
/*
/&
* $$ E0J
```

Figure 11. LPR-Job on TCP/IP for VSE/ESA

Setting the Parameters for the IPM Support

Description of the SET Parameters

SET HOST=9.66.110.67

With this setting the system (AIX or Windows) containing the InfoPrint Manager with the IP address 9.66.110.67 is addressed. Note that the system is addressed, not the IP address of a specific printer.

SET PRINTER=ipheft

With this setting a Logical Destination (LD) with the name ipheft is addressed. The Logical Destination of the InfoPrint Manager can be compared in general to a printer queue. Note that in AIX no real AIX printer queue with this ipheft name should exist, otherwise this queue would be addressed instead of the LD of the IPM.

With LPR, the LPD of the AIX is addressed. Because the LPD does not know a printer queue with that specific name (ipheft), the LPD is routing the print job automatically to the IPM. IPM then acknowledges the name (ipheft) as a known LD back to the LPD and assigns the print job to this LD.

SET PAGEDEF=b111 and SET FORMDEF=a222

With these settings the PAGEDEF and FORMDEF names can be specified which are required for the AFP print formatting. These definitions are automatically extended with a preceding P1, or F1 and overwrite the default specifications of PAGEDEF and FORMDEF in the addressed LD. These PAGEDEF and FORMDEF resources should be available at the IPM site. In the example shown above, the PAGEDEF definition named P1b111 should be available at the IPM site.

SET INFOPRINT=YES

With this setting the transmission of the print data and its associated control characters is done in EBCDIC. This allows the transmission of machine or ASA codes as well as AFP structured fields. AFP commands (x'5A' data records) can be imbedded, for example, to directly call COPYGROUPS with the IMM command or to control the individual stitching using BDT/EDT commands.

SET CC=YES

This setting is required to transmit the print control characters or AFP control characters (x'5A').

SET CRLF=UNIX, SET NOEJECT=ON, SET DISP=KEEP

These parameters are required to ensure a proper data transfer and to put the transmitted data set into disposition KEEP.

Using the SET FNO= Parameter with IPM

To use the forms parameter SET FNO=fno, you must define the following:

1. In the file /etc/environment of IPM, include one line with: PD_FORMS=true
2. Prepare an actual destination of IPM to use. This is done with 2 parameter changes to the actual destination as described in the following section.

Customizing the InfoPrint Manager

A Logical Destination (LD) must be defined in the InfoPrint Manager on your AIX or Windows system. In the example shown in Figure 11 on page 30, this LD has the name ipheft.

The document defaults of this LD must be defined as follows:

Document Other: Format = line-data

This definition is required for line data transfer using the LPR or LPD communication. It is also valid for line data with imbedded AFP structured fields records, that is, mixed mode AFP print applications.

Document Processing: Transform Options = INDEXOBJ=BDTLY

This setting should only be used if stitching is required when using, for example, the following IBM printers: IP2000, IP60, and IP70. Note that otherwise indexing by using ACIF will not be possible.

Document Line Data: Location of page definitions = /usr/lpp/psf/user/ppfalib

This directory specifies, for example, where the defined PAGEDEF resource resides.

Type of carriage control characters = machine

This definition describes whether machine or ASA print control codes will be used.

Convert to EBCDIC = No

This definition specifies that the data is being transferred in EBCDIC.

Document AFP Resources: Location of form definitions = /usr/lpp/psf/user/ppfalib

This directory specifies, for example, where the defined FORMDEF resource resides.

Changing the Properties of the Actual Destination

Define the following to change the properties of the actual destination:

Load Balancing: Disable on Job mismatch = No

Load balancing: Job-Batches-Ready ADD = *fno-value*

(The maximum *fno-value* is 4 characters).

Note: Several fno-values can be added for this actual destination.

If a job with matching fno-value=Job-Batches-Ready-value is received by IPM, the print job starts printing immediately. The operator of IPM makes sure that the print job has job-batch-value=fno-value assigned. If a job with no matching fno-value=Job-Batches-Ready-value is received by IPM, the print job goes into hold and a message indicating '*resources not ready*' is displayed. In this case, the operator must load the printer with the requested forms paper, and then change Job-Batches-Ready-value of the actual destination to the requested one, and set the actual destination ready. The hold print job starts automatically.

The operator can use a shortcut by right clicking on the actual destination and choosing Job-Batches-Ready. Select ADD or REMOVE the values as needed. The fno-value received from the VSE system is moved into the JOB-BATCH value of the print job, and displayed accordingly. The JOB-BATCH value of the print job is finally compared to the Job-Batches-Ready value of the actual destination and IPM is acting accordingly. In addition, the parameter fno-value is passed to the -opassthru=forms value and is printed on the job separator sheet as 'FORMS: fno-value'.

Technical Background Information

If SET INFOPRINT=YES is used, the LPR of TCP/IP for VSE/ESA also transfers the parameter -ofileformat=record. With this setting, every record is preceded by a field with a length of 2 bytes. This field is detected automatically by IPM and is removed before printing.

In the LPD/LPR control file of the file to be transferred, the following settings can be found, for example, when investigating a TCP/IP trace.

Support	LPR Script	translated to -o option(s)
EBCDIC	SET INFOPRINT=YES/NO (or =ON/OFF)	-ofileformat=record and -odatatype=line
Pagedef	SET PAGEDEF=pdef	-opagedef=P1pdef
Formdef	SET FORMDEF=fdef	-oformdef=F1fdef
Forms	SET FNO=fno	-opassthru=forms=fno and -ofrms=fno

Software Prerequisites

The InfoPrint Manager support requires the following minimum software level:

- IPM for AIX 3.2, APAR IY17446 / PTF U475406, or
IPM for Windows NT or Windows 2000 1.1, CSD level 1.1.0.10

Chapter 6. z/VSE Related Hardware Functions Supported by TCP/IP for VSE/ESA 1.5

The following hardware-related functions are supported by TCP/IP for VSE/ESA 1.5:

- Hardware Crypto
- HiperSockets
- OSA Express2 and OSA Express

Hardware Crypto Support

The z/VSE hardware encryption assist support (referred to as hardware Crypto support) requires a crypto card, such as Crypto Express2 or Crypto Express3 or equivalent. The cards are available in IBM System z environments. It provides encryption assist support and can help to increase the throughput in a TCP/IP network using SSL (Secure Sockets Layer).

If z/VSE runs under z/VM, z/VM 4.2 or higher is required.

Refer to *z/VSE Planning* for further details.

HiperSockets

z/VSE supports high-speed TCP/IP communication among logical partitions (LPAR) and virtual machines using HiperSockets. The HiperSockets support is available in IBM System z environments.

Refer to *z/VSE Planning* for further details.

OSA Express Support

The OSA Express support is provided as integrated hardware feature (OSA Express3, OSA Express2 and OSA Express adapters) in IBM System z environments. The support provides direct connectivity between z/VSE applications and other platforms on the attached network. It is based on the Queued Direct Input/Output (QDIO) architecture which allows a highly efficient data transfer and results in an accelerated TCP/IP data packet transmission.

Refer to *z/VSE Planning* for further details.

For advanced OSAX device driver configuration refer to Appendix C, “Advanced OSAX Device Driver Configuration,” on page 527.

Chapter 7. Performance Considerations

Changing Performance Parameters

It is highly important to have an optimal selection of performance-relevant setup or operational parameters. There are

product defaults

Product defaults apply whenever the value has not been explicitly assigned.

shipped defaults

Shipped defaults apply whenever the customer has not changed or overwritten the startup values in the shipped startup job for TCP/IP for VSE/ESA. The shipped specific startup values for a parameter usually represent a good starting point. However, based on specific loads or configurations, there may be good reasons for a change.

Both values often do not coincide. Be aware before you change a parameter which does not influence the workload(s), you will not see any change.

TCP/IP for VSE/ESA performance is influenced by many different parameters that can be tailored for the specific operating environment.

In general these tuning parameters can be grouped into

- operating system tuning
- TCP/IP tuning
- communication tuning (mainframe end and workstation end)
- TCP/IP application tuning

As operating system tuning is familiar to most z/VSE customers, it need not be addressed in more detail here.

To better understand potential effects of TCP/IP tuning, it is very helpful to understand some basic TCP/IP concepts. These concepts include

- frames, datagrams and segments
- fragmentation and reassembly
- send and receive buffer management via window sizes and acknowledgements

Communication tuning is closely related to TCP/IP for VSE/ESA tuning. It refers to the configuration (including links etc) of the network and also the parameter selection on the other side, which also is TCP/IP.

As it is true for any type of tuning, make only one change at a time. Changing a parameter in your environment may not produce any improvement as another value may dominate performance. Having changed this value, the same change may improve performance considerably.

General Performance Issues

The following types of performance data exist:

Performance Considerations

Resource consumption of an activity

How much CPU-time, I/Os are required to perform a certain TCP/IP activity. For example to use TELNET for CICS transactions, or to transfer 1M of data.

Achievable Throughput/Performance Values

How many terminals can be concurrently supported with TN3270, or, what data rate can be achieved for 1 concurrent FTP activity in a certain environment.

Principal Performance Dependencies for TCP/IP for VSE/ESA

The performance you get with TCP/IP applications is very dependent on all the hardware and software products involved. The following is a list of principal parameters which tries to globally categorize performance/tuning impacts. Overall performance is determined by the components shown:

Table 1. Principal Performance Parameters

Parameter (type)	Host CPU time	Host storage	Transfer time	DASD time
Host CPU speed	X	-	-	-
S/390® System Control Program and setup	X	X	-	x
MTU/MSS used	X	x	X	-
Window size	-	x	X	-
Transfer buffers	-	X	x	-
Type of Comm. Adapter	-	-	X	-
Network/Line speed	-	-	X	-
Network reliability	X	x	X	-
#Appl. bytes in/out	X	X	X	X (application dependent)
TCP/IP implementation	X	X	X	X
TCP/IP application	X	X	X	X
Other TCP/IP parms	X	X	X	X
DASD I/O subsystem	-	-	-	X
DASD I/O blocking	x	-	-	X
<p>Note:</p> <p>X means major impact.</p> <p>x means smaller or secondary impact.</p> <p>- means no or negligible impact.</p> <p>Transfer time includes wait for transfer.</p> <p>DASD time only applicable if DASD involved (for example FTP).</p>				

Overall Capacity is also of interest and of specific importance for multiple concurrent sessions (for example Telnet3270).

The following is a list of principal parameters showing performance-relevant settings in TCP/IP for VSE/ESA. It also shows which TCP/IP activities a parameter can influence.

Table 2. TCP/IP Performance-Relevant Parameters

Scope of TCP/IP Activity					
TCP/IP Parameter / Setting	Any	Outbound	TCP Inbound	TN3270 Out + In	FTP Out+In
DEFINE ADAPTER LINK MTU TELNETD POOL		X		X	
SET ALL_BOUND REDISPATCH ARP_TIME REUSE_SIZE FULL_SCAN GATEWAY CHECKSUM	X X X x X X x				
Set MAX_SEGMENT WINDOW_DEPTH CLOSE_DEPTH WINDOW_RESTART			X1 X1 X1 X1		
SET RETRANSMIT FIXED_RETRANS WINDOW ADDITIONAL_WINDOW		X1 x1 X1 x1			
SET TELNETD_BUFFERS TRANSFER_BUFFERS MAX_BUFFERS				X2	X X
X means major impact x means smaller or secondary impact X1 only for TCP loads (includes FTP, but not NFS) X2 only for POOL=YES TELNET daemons/sessions					

More specific TCP/IP for VSE/ESA performance information and performance results are available on the z/VSE Internet home page at <http://www.ibm.com/systems/z/os/zvsel>.

Refer also to the *IBM TCP/IP Performance Tuning Guide* SC31-7188, which addresses concepts, tuning and benchmark data for TCP/IP for MVS™, VM, AIX, OS/2, DOS, and OS/400®.

See *TCP/IP for VSE 1.5 Commands* for a description of operation and default values of the individual commands.

Part 2. Using IPv6/VSE

Chapter 8. IPv6/VSE Overview

Part 2, “Using IPv6/VSE,” on page 39 of this publication gives an overview of the IPv6/VSE (5686-BS1) program. IPv6/VSE is a native implementation of Transmission Control Protocol/Internet Protocol (TCP/IP) providing an IPv6 solution for z/VSE.

IPv6/VSE provides an IPv6 TCP/IP stack, IPv6 application programming interfaces (APIs), and IPv6-enabled applications. IPv6/VSE V1R1 supports the IPv4 and the IPv6 protocol.

The IPv6 TCP/IP stack of IPv6/VSE can be run concurrently with an IPv4 TCP/IP stack within one z/VSE system.

IBM provides extensions that can be used with this program. They are described in detail and include the EZASMI macro interface and the EZASOKET call interface. Be aware that these extensions in its external interfaces might describe more functionality than really provided by the IPv6/VSE program. Check the IPv6/VSE documentation for such exceptions.

For advanced OSAX device driver configuration refer to Appendix C, “Advanced OSAX Device Driver Configuration,” on page 527.

Before using the IPv6/VSE program read the following very carefully.

IPv6 TCP/IP Stack

The IPv6 TCP/IP stack of IPv6/VSE runs in a separate partition using its own stack ID. This allows both an IPv4 and IPv6 stack to run concurrently within one z/VSE system. Running separate IPv4 and IPv6 stacks concurrently within one z/VSE system addresses both performance and reliability aspects. Existing IPv4 applications continue to run unchanged using the IPv4 TCP/IP stack, thus protecting and leveraging existing client investments. New IPv6-enabled applications can gradually be introduced using the IPv6 stack of IPv6/VSE.

Dual Stack Support

Dual stack support allows an application to connect to both the IPv4 and IPv6 network simultaneously. With the implementation of dual stack support, a single IPv6-enabled CICS transaction or batch application can communicate with partners via either the IPv4 or IPv6 network. Enhanced socket APIs are provided that can be used to introduce IPv6-enabled applications.

IPv6-Enabled Utility Applications

IPv6/VSE provides utility applications that run outside the IPv6/VSE stack partition. Running these applications external to the IPv6/VSE stack partition provides greater stability and better performance.

FTP server

The IPv6/VSE FTP server supports access to z/VSE resources, like POWER queues, VSAM catalogs, SAM file and z/VSE libraries, by remote host FTP clients.

IPv6/VSE Overview

Batch FTP client

The IPv6/VSE batch FTP client runs as a z/VSE batch job providing access to remote host FTP servers. Data can be interchanged with these remote FTP servers.

TN3270E server

The IPv6/VSE TN3270E server supports TN3270/TN3270E terminal sessions and TN3270E printer sessions. In addition, DIRECT, LPR, and FTP printer sessions are supported.

NTP server

The IPv6/VSE NTP server is a Network Time Protocol server that allows remote hosts to query the time of day (TOD) clock of z/VSE to synchronize their clocks with the z/VSE clock.

NTP client

The IPv6/VSE NTP client allows z/VSE to set its TOD clock to an external source.

System Logger client

This application is used to log selected z/VSE console messages to a remote Linux syslog-ng daemon. Once a message is sent to the syslog-ng daemon, Linux automation processing can be used to trigger events.

Batch email client

The IPv6/VSE batch email client is used to send an email to an SMTP server. In turn, the SMTP server will send the email to a destination user. Any number of recipients are permitted and files can be attached to an outgoing email message.

Batch LPR

The IPv6/VSE Batch LPR application extracts data from POWER queues and transfers it to a remote host LPD. The LPD can be in a printer or running as a server on a remote host.

Batch Remote Execution Client

The IPv6/VSE Remote EXEC Client allows a job running in a z/VSE partition to trigger a script to run on a remote host. Any output from the script is returned to the client and scanned for completion information.

Batch PING

The IPv6/VSE Batch PING application is used to ping a remote host.

GZIP data compression

IPv6/VSE provides a simple gzip data compression application. Data can be read, compressed, and written to a SAM file or library member. The compressed data can then be transferred to a remote host for processing. The reverse of this process can also be performed.

REXX automation

IPv6/VSE uses z/VSE REXX EXECs for automation. Automatic FTP of data is handled with a sample REXX EXEC that is provided. Automatic LPR or automatic email of data is handled in the same way. Invoking IPv6/VSE applications with a REXX EXEC allows dynamic creation of commands and parameters, for example file names and dates.

BSTTPRXY / BSTTATLS

IPv6/VSE provides two servers providing SSL for z/VSE server and client applications. Both servers provide SSL/TLS transparently to the application. They support both batch and CICS applications written in any

supported API including applications using the ASM SOCKET macro, EZASMI, EZASOKET and LE/C APIs.

- The SSL proxy server (BSTTPRXY) is a simple proxy server. It allows only a single PROXY command. To proxy multiple connections you must run multiple BSTTPRXY partitions. BSTTPRXY performs IPv4 to IPv6 or IPv6 to IPv4 translation.
- The AT-TLS server (BSTTATLS) Automatic Transport Layer Security is a facility that is similar to the z/OS AT-TLS (Application Transparent - Transport Layer Security) facility. BSTTATLS allows many AT-TLS definitions and monitors incoming and outgoing connections, intercepting and converting sockets from clear text to SSL or vice versa as necessary. However, BSTTATLS do not perform IPv4 to IPv6 or IPv6 to IPv4 translation.

The IPv6/VSE utility applications FTP server, FTP client, LPR, Batch email client, and GZIP support Double Byte Character Set (DBCS).

Documentation for the IPv6/VSE (5686-BS1) Program

The product description of the IPv6/VSE product (IBM product number: 5686-BS1) is only available in PDF format on the z/VSE DVD collection (SK3T-8348) and on the Internet at <http://www.ibm.com/systems/z/os/zvse/documentation>.

The IPv6/VSE product documentation consists of 8 publications with the original product description from Barnard Software Inc., the provider of the IPv6/VSE product, plus one publication provided by IBM giving an overview of the IPv6/VSE product – this publication.

The publications are as follows:

- *z/VSE TCP/IP Support* (this publication)
- *IPv6/VSE IPv6 Installation Guide*
- *IPv6/VSE IPv4 Installation Guide*
- *IPv6/VSE IPv6 User's Guide*
- *IPv6/VSE IPv4 User's Guide*
- *IPv6/VSE Programming Guide*
- *IPv6/VSE Migration Guide*
- *IPv6/VSE Messages and Codes*
- *IPv6/VSE SSL Installation, Programming and User's Guide*

You can use the Adobe Acrobat Reader to view and print these publications. If you do not already have Acrobat Reader installed, or if you need information on installing and using Acrobat Reader, see the Adobe website at <http://www.adobe.com>.

IPv6/VSE Installation Requirements

Operating System Requirements

IPv6/VSE requires z/VSE 4.2 and later. The minimum required service level is APAR DY47077 or z/VSE 4.2.2.

Processor Requirements

IPv6/VSE runs on any hardware configuration supported by z/VSE 4.2 and later.

Preventive Service Planning

Before installing IPv6/VSE check with your IBM Support Center or refer to the VSE Homepage if additional service is available.

User Access Key

IPv6/VSE the native IPv6 solution for z/VSE is available as an optional-priced IBM program (5686-BSI). IBM has licensed this program from Barnard Software, Inc.

IPv6/VSE is also available as a stand-alone product. IPv6/VSE is **not** preinstalled in the z/VSE base.

IPv6/VSE can be used for 30 days after activation without a key. After the 30 day trial period IPv6/VSE requires a unique user access key which depends on the machine's CPUID.

To request a unique user access key, contact the IBM Copenhagen Key Center by email (speckkeys@dk.ibm.com) or phone (+45 48 10 15 30) (non-toll free). When you receive the license key from the Key Center by email, you have to create a BSTTPARM.A phase and include the following lines, completed with your personal data, exactly as shown in the example:

```
COMPANY name
CPUID xxxxxx MODEL xxxx
LICENSE TCP/IP-TOOLS ABCDEFGHL6Z date vcode
*
TCP/IP-TOOLS ENABLE
```

Refer to the IPv6/VSE Installation Guide for detailed installation instructions.

Security Manager Exploitation by IPv6/VSE

IBM provides a security exit routine called BSSTISX. The BSI FTP server security exit routine BSTTFTS1.PHASE can be set up to call the IBM security exit to verify userid and password. For details refer to the IPv6/VSE User's Guide.

z/VSE Related Hardware Functions Supported by IPv6/VSE - Processor Requirements

With the z/VSE 5.1 Service Upgrade the IPv6/VSE BSTT6NET TCP/IP stack requires a processor listed in *z/VSE Planning* under Hardware Support.

Network Interface Requirements

IPv6/VSE supports CTCA, 6in4 tunnels, OSA Express and HiperSockets network interfaces. Only OSA Express adapters running in QDIO mode or HiperSockets network interfaces are supported.

Part 3. Programming Interfaces

Chapter 9. Introducing Socket Programming

This section introduces socket programming as provided by the TCP/IP stacks supported by z/VSE.

You can connect to and from a VSE host and interchange data with the system with different access methods. Check the product information of your TCP/IP stack to verify, which access methods are supported.

Telnet

Telnet can be used from remote hosts to connect to VTAM applications running on the local z/VSE. On the local z/VSE host it can be used to connect to other remote systems running Telnet daemons, connecting to a UNIX workstation

File Transfer Protocol (FTP)

FTP is used to get/put data files from/to a remote host system

Web Server

The web server can be accessed by web browsers retrieving data defined by HTML (Hypertext Markup Language) pages.

- Static page contents : HTML only
- Dynamic page contents : HTML, including JavaScript, Java™ Applets or calling CGI (Common Gateway Interface) programs.

Client/Server applications

Distributed applications communicating over an enterprise intranet or the Internet. The application establishes a peer-to-peer communication exploiting the TCP/IP socket programming interface.

This section focuses on the requirements of TCP/IP socket based Client/Server applications. It intends to show what aspects are to be considered before deciding which programming interface to use and how to use them.

What is a TCP/IP Socket Connection ?

A socket programming interface provides the routines required for interprocess communication between applications, either on the local system or spread in a distributed, TCP/IP based network environment. Once a peer-to-peer connection is established, a socket descriptor is used to uniquely identify the connection. The socket descriptor itself is a task specific numerical value.

One end of a peer-to-peer connection of a TCP/IP based distributed network application described by a socket is uniquely defined by

- Internet address
for example 127.0.0.1 (in an IPv4 network) or FF01::101 (in an IPv6 network).
- Communication protocol
 - User Datagram Protocol (UDP)
 - Transmission Control Protocol (TCP)
- Port
A numerical value, identifying an application. We distinguish between
 - "well known" ports, for example port 23 for Telnet

Introducing Socket Programming

- user defined ports

Socket applications were usually C or C++ applications using a variation of the socket API originally defined by the Berkeley Software Distribution (BSD). The JAVA language also provides a socket API. JAVA based Client/Server applications exploit those socket services.

Socket programming interfaces have been standardized for ease of portability by The Open Group for example.

Besides TCP/IP based sockets, UNIX systems provide socket interfaces for interprocess communication (IPC) within the local UNIX host itself. Those UNIX sockets use the local file system for interprocess communication.

z/VSE provides TCP/IP based socket services. They can be used for IPC too, although they are primarily aimed for network communication only.

Socket Application Programming Interfaces Available with z/VSE

z/VSE provides a series of different socket application programming interfaces (APIs) These interfaces are supported by TCP/IP for VSE/ESA, IPv6/VSE, and the Fast Path to Linux function (LFP). For details, please refer to the corresponding product information.

- EZA interfaces
 - z/VSE provides the EZASMI macro interface for HLASM programmers and the EZASOCKET call interface for COBOL, PL/I and HLASM programmers. These interfaces are widely compatible with the corresponding z/OS interfaces and are supported by TCP/IP for VSE/ESA, IPv6/VSE and LFP. Refer to Chapter 11, “Using the CALL Instruction Application Programming Interface (EZASOCKET API),” on page 199 and Chapter 12, “Using the Macro Application Programming Interface (EZASMI API),” on page 291 for a description of these interfaces.

Asynchronous function processing with the EZASMI interface is provided on z/VSE as well. But compared to z/OS, only the ECB method is available, and the ECB area must have a length of 160 bytes (compared to 104 bytes in z/OS).
- TCP/IP APIs using Language Environment for z/VSE
 - LE/VSE 1.4 C socket interface dynamically determines the runtime environment (CICS or Batch). Refer to “PL/I” on page 54 and “COBOL” on page 55 for details.
 - The REXX/VSE Socket API support within REXX/VSE. Refer to *REXX/VSE Reference*, SC33-6642 for details.
- TCP/IP for VSE/ESA 'native' APIs
 - Assembler SOCKET macro interface

This interface supports coding socket applications, but also dynamically connecting to remote systems using TCP/IP built-in Telnet, FTP and LPR application level protocol support. It needs to be specified if used in a batch or CICS environment.
 - COBOL and PL/I preprocessor interface

It needs to be specified if used in a batch or CICS environment.
 - BSD-C socket interface

You can make the application dynamically determine the runtime environment (CICS or Batch). Refer to "CICS Considerations" on page 83 for details.

- REXX socket APIs There are two types of REXX support for TCP/IP for VSE/ESA available:
 - The REXX support within TCP/IP for VSE/ESA (i.e. REXX Socket API). The documentation of this REXX support can be found in the *TCP/IP for VSE 1.5 Programmer's Reference* manual.
 - The REXX/VSE Socket API support within REXX/VSE is described in more detail below.

Refer to *TCP/IP for VSE 1.5 Programmer's Guide* for a detailed description of these interfaces.

Portability Aspects

Assembler

Usage of the TCP/IP for VSE/ESA Assembler SOCKET Macro Interface ties the program to z/VSE. Programs written in Assembler, using the SOCKET macro interface are not portable to non z/VSE operating system environments as there is no API standard for this language.

The EZASMI macro interface and EZASOKET call interface are also available within z/OS, with minor differences. Applications using these interfaces on z/OS can easily be ported to z/VSE and vice versa.

COBOL and PL/I

While COBOL and PL/I are the dominant programming languages in the z/VSE environment, the "native" language for writing TCP/IP based socket applications is C. Interfaces for languages other than C might exist in specific environments or might be provided by product specific programming toolkits, which potentially are available for multiple platforms.

If portability to non z/VSE systems is not essential, you can choose the TCP/IP for VSE/ESA preprocessor API, described in the *TCP/IP for VSE 1.5 Programmer's Reference* manual. If portability to z/OS or z/VM is essential, read section "Language Environment" on page 50 below for further details. If portability to z/OS is essential, consider using the EZASOKET call interface.

C Language

As mentioned before, C is the only programming language besides JAVA where very similar programming interfaces are provided in arbitrary operating system environments.

While the C socket interfaces are standardized by the Berkeley Software Distribution (BSD), there are other standards to assure cross-system and cross-platform portability, for example by The Open Group in their CAE specification : "System Interfaces and Headers, Issue 4, Version 2", in the literature also being referred to as XPG4.2. The Open Group can be found on the Internet at: <http://www.opengroup.org/>.

Language Environment®

The Language Environment (LE) on the IBM System z platform assures portability across z/OS, z/VM and z/VSE. Depending on specific needs and portability issues one of the following languages

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

is appropriate for writing TCP/IP socket interface based Client/Server applications.

LE supports the usage of LE services using any LE enabled High Level Language (C, COBOL, PL/I) or from within an LE conforming Assembler program. This includes support for mixed-language applications.

While LE based programs, using socket services and written in a programming language other than C are not portable to a System z system, LE on System z provides cross system compatibility.

LE Enabled Applications

An application is considered to be "LE enabled" (or "LE Conforming" or "LE Compliant"), if it conforms to the common execution environment (CEE) model and conforms to this runtime linkage, storage and condition handling model. This is true if the application is compiled, or assembled, using LE conforming compilers or prologue/epilogue macros. These are basically all C for VSE, COBOL for VSE, and PL/I for VSE compiled programs and Assembler programs using CEEENTRY/CEETERM macros. C for VSE subroutines including assembler programs using the C prologue/epilogue assembler macros also fulfill this requirement.

Which API to use?

As discussed already, the selection of the appropriate language and API to use depends on:

- Portability
Ease of cross-platform development (single source code).
- Compatibility
The System z platform provides source compatibility between z/OS, z/VM and z/VSE when using LE programming interfaces.
LE/VSE focuses on the interfaces defined by the C feature test macro `_XOPEN_SOURCE_EXTENDED`, where for example z/OS additionally provides slightly different interfaces, enabled by the feature test macro `OE_SOCKETS`.
- Serviceability
By decoupling the socket application from the TCP/IP product allows maintaining (servicing) both parts independently.

Portability, compatibility and serviceability aspects show up differently, depending on the programming language chosen.

Assembler

The SOCKET macro provided by TCP/IP not only supports writing socket based applications, but grants access to the built-in Telnet, FTP and LPR application level protocols as well. If Telnet, FTP and LPR protocol access isn't required, an LE conforming Assembler program can call the LE/VSE C socket interfaces instead of using the SOCKET macro to gain independence from the TCP/IP service level.

TCP/IP service affecting the SOCKET macro might require to reassemble the application.

The EZASMI macro and the EZASOKET call interface are mostly compatible with the corresponding z/OS interfaces. This eases cross-platform development. With both interfaces, socket applications are decoupled from the TCP/IP product, which allows both parts to be serviced independently.

COBOL and PL/I

Using the TCP/IP preprocessor API (EXEC TCP ...) a stub routine linked edited with the user application

- COBOL - IPNETXCO.OBJ
- PL/I - IPNETXPOBJ

TCP/IP service affecting those modules might require to re-link the application.

The following screen shows an example of the usage of the preprocessor interface.

```
*
*   Attempt to open a connection
*
EXEC TCP OPEN FOREIGNPORT(2000)
              FOREIGNIP(IPADDRESS)
              LOCALPORT(0)
              RESULTAREA(RESULTS)
              DESCRIPTOR(MY-DESC)
              ACTIVE
              WAIT(YES)
              ERROR(SECOND-TEST)
END-EXEC.
```

Note that the EZASOKET call interface can be used with COBOL for VSE and PL/I for VSE programs as well.

C Language

Acknowledging the dominance of C in TCP/IP environments, LE/VSE provides C socket interfaces only. However, LE/VSE as well as the Language Environments in z/OS and z/VM, allows to call LE services from Assembler, COBOL and PL/I too. In addition you can also use the EZASOKET interface from COBOL and PL/I programs.

The figure below shows the logical control flow of a LE/VSE C based socket application. The LE/C runtime decouples the application from a specific TCP/IP product. The LE/C TCP/IP Socket API Multiplexer allows to select an appropriate TCP/IP stack at runtime. Per default, the \$EDCTCPV.PHASE is used to work with the TCP/IP stack partition. To use other LE/C TCP/IP Interface routines you can configure the LE/C TCP/IP Socket API Multiplexer. For example, phase IJBLFPLE to use the LFP LE/C TCP/IP interface or phase BSTTTCPV/BSTTTCP6 to use the

Introducing Socket Programming

IPv6/VSE interface. To configure the multiplexer, use skeleton EDCTCPMC in ICCF library 62.

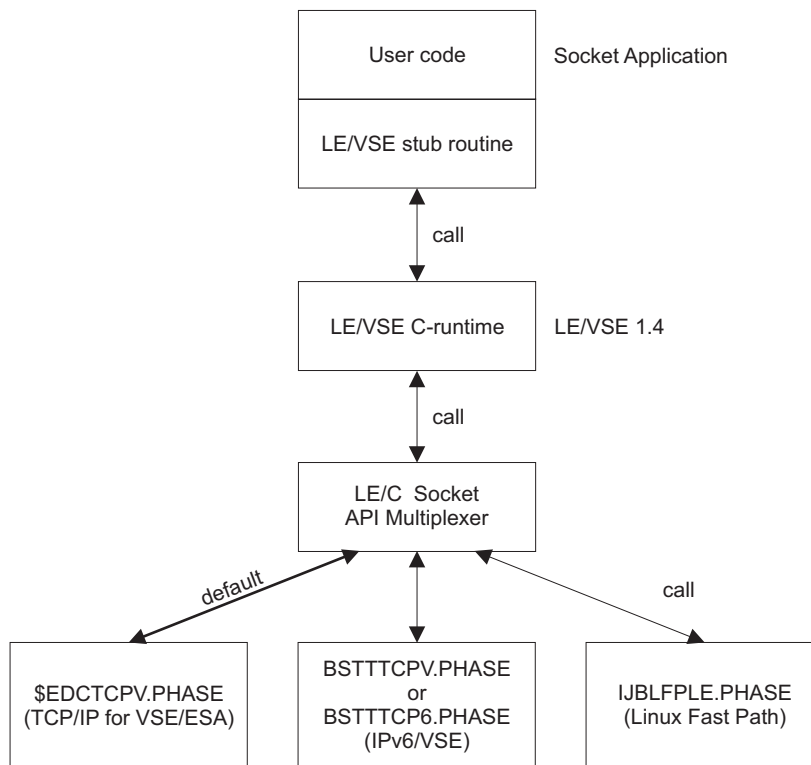


Figure 12. Control Flow when using LE/VSE C Sockets with different TCP/IP stacks.

Note:

1. If you use the **C for VSE** compiler, you should use the socket API provided by the Language Environment 1.4. The C header files required are provided in VSE library PRD2.SCEEBASE.
2. If you use a non-LE enabled C compiler, for example **C/370™**, you are restricted to use the native TCP/IP for VSE/ESA BSD-C interface. This includes the usage of the **socket.h** include file shipped in VSE library PRD2.TCPIPC.

The LE/C Socket Interface can be used with TCP/IP for VSE/ESA, IPv6/VSE, and Linux Fast Path. For details refer to the corresponding product information.

Exploiting the LE/VSE Socket API

Applications using LE runtime services (C, COBOL and PL/I) or LE enabled Assembler programs can use the LE/VSE C socket routines, either directly (C) or using the LE Interlanguage Communication (ILC) support. In addition, you can also use the EZASOKET interface from COBOL and PL/1 programs.

C Language

LE/VSE provides socket programming interfaces for C. These are described in detail in Chapter 10, "TCP/IP Support for the LE/VSE C Socket Interface," on page 85. Per default, these interfaces use the \$EDTCPV.PHASE to work with the TCP/IP stack partition. If TCP/IP for VSE/ESA is installed on your VSE system you will find the \$EDTCPV.PHASE in your TCP/IP for VSE/ESA library. To use other LE/C TCP/IP Interface routines you can configure the LE/C TCP/IP Socket

API Multiplexer. For example, phase IJBLFPLE to use the LFP LE/C TCP/IP interface or phase BSTTTCVPV to use the IPv6/VSE interface. To configure the multiplexer, use skeleton EDCTCPMC in ICCF library 62.

While the TCP/IP HLL interfaces basically provide a OPEN, SEND, RECEIVE, CLOSE interface, the C language calls provide a higher granularity. The calls necessary depend on writing a server or a client program.

Client

The following example shows a simplified code logic for a client application:

```
socket()          - create a socket
  ↓
connect()         - bind and connect to server
  ↓
send() / receive() - data interchange
  ↓
close()           - destroy socket
```

Server

The following example shows a simplified code logic for a server (Daemon) application:

```
socket()          - create a socket using a specific protocol
  ↓
bind()            - bind the socket to a port
  ↓
listen()          - make it a passive socket
  ↓
[ select() ]      - wait for incoming connections
  ↓
accept()          - connect to caller
  ↓
getsockname()     - determine caller
  ↓
send() / receive() - data interchange
  ↓
close()           - destroy socket
```

The select() call in brackets shown above can be used to operate multiple clients concurrently. It can be used to wait for activity on a series of sockets, similar to a WAITM (wait multiple) operating system call. Therefore the server application can wait for new clients to connect (accept() call) and concurrently wait for requests from clients already connected (receive() call).

Assembler Language

LE/VSE supports calling C subroutines from an Assembler program.

Assembler source

The code snippet in the following example uses LE macro CEEENTRY to enable the Language Environment. Then it calls TCP/IP subroutine GETHNAM. At the end of the routine it calls CEETERM to disable the Language Environment as not required any longer.

Note: It is recommended to enable LE at the very beginning and terminate it at the end of your application. Do not call this sequence more than required or there will be high overhead introduced by starting/terminating LE more than necessary.

Introducing Socket Programming

```
*
GETHOSTN  CEEENTRY PPA=MAINPPA,MAIN=YES
*
*
          LA    1,PARMSTR
          CALL  GETHNAM
*
          LTR   15,15
          BZ    RETOK
          WTO   'GETHOSTNAME() FAILED'
          B     RTNEND
RETOK     WTO   'GETHOSTNAME() SUCCESSFUL'
*
RTNEND    CEETERM
*
CBUFLEN   EQU  20
PARMSTR   DC   A(HNAME)
          DC   F(CBUFLEN)
HNAME     DS   CL(CBUFLEN)
```

C subroutine with OS linkage called from Assembler

The following example shows how to write a stub routine with OS linkage convention calling the C routine `gethostname()`.

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

#pragma linkage(GETHOSTNAME, OS)
#pragma map(GETHOSTNAME, GETHNAM)

int GETHOSTNAME( char *buffer,
                 size_t size)
{
    return( gethostname( buffer, size));
}
```

PL/I

LE/VSE Interlanguage Communication (ILC) between C and PL/I is only provided for PL/I for VSE/ESA.

The manual *Writing Interlanguage Communication Applications*, SC33-6686, provides details on how to use ILC calls.

Similar to the Assembler example, there must be a C stub routine with PL/I linkage. Note the following:

- A NULL in C is `x'00000000'` where NULL in PL/I is `x'FF000000'`. Therefore PL/I programs should check for `SYSNULL` (`x'00000000'`) where appropriate.
- A character string in C is logically unbound with a `x'00'` end indicator (last byte).

The stub routine for calling `gethostname()` could therefore look like this:

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

#pragma linkage(GETHOSTNAME, PLI)
#pragma map(GETHOSTNAME, GETHNAM)

int GETHOSTNAME( char **buffer,
                 size_t size)
{
```



```

    return( gethostname( *buffer, size));
}

```

The matching PL/I code fragment, calling the subroutine could look like this:

```

...
DCL GETNAM EXTERNAL ENTRY
    RETURNS(FIXED BIN(31));
DCL HOSTNAME CHAR(20);
DCL HNSIZE FIXED BIN(31);
DCL CRC FIXED BIN(31);
...
HNSIZE = 20;
CRC = GETNAM(ADDR(HOSTNAME), (HNSIZE));
...

```

COBOL

LE/VSE Interlanguage Communication (ILC) between C and COBOL is provided for COBOL for VSE/ESA Release 1

The manual *Writing Interlanguage Communication Applications*, SC33-6686, provides details on how to use ILC calls.

The following example shows how to call the LE C routine `gethostname()` to retrieve the name of the local host:

```

IDENTIFICATION DIVISION.

        PROGRAM-ID.      C2COB2.
        AUTHOR.          INGO ADLUNG.
        INSTALLATION.    BOEBLINGEN GERMANY.
        DATE-WRITTEN.    MAY 19, 1999.
        DATE-COMPILED.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
        SOURCE-COMPUTER.  IBM-370.
        OBJECT-COMPUTER.  IBM-370.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 RESULTS.
   05 RVALUE          PIC S9(9) BINARY.
   05 RDETAIL         PIC S9(9) BINARY.
01 BUFSIZE          PIC S9(9) BINARY.
01 BUFFER.
   05 WORKAREA       PICTURE X(64).

PROCEDURE DIVISION.

MAIN.
*
* Display the name of the host we are running on
*
        MOVE 64 TO BUFSIZE.

        DISPLAY 'Calling C gethostname()' UPON CONSOLE.

        CALL 'COBGHNAME' USING BY REFERENCE WORKAREA
                                BY CONTENT BUFSIZE
                                BY REFERENCE RVALUE, RDETAIL.

```

Introducing Socket Programming

```
DISPLAY WORKAREA UPON CONSOLE.
```

```
STOP RUN.
```

The matching C stub routine for calling `gethostname()` with COBOL linkage could look like this:

```
#include <types.h>
#include <unistd.h>
#pragma linkage(cobol_gethostname, COBOL)
#pragma map(cobol_gethostname, COBGHNAM)

void cobol_gethostname( char  *buffer,
                       size_t size,
                       int   *return)
{
    *return = gethostname( buffer, size);
}
```

A COBOL Example using LE C Socket Services

The next example is based on LE's ability to write interlanguage communication applications.

The complete source code can be obtained as `cobsock.zip` from the z/VSE home page at <http://www.ibm.com/systems/z/os/zvse/downloads/samples.html> following the FTP download link.

The figure shown below contains the COBOL source code for a very basic server application. To reduce complexity it handles a single client only and doesn't include the error recovery necessary if communication problems show up.

```
IDENTIFICATION DIVISION.
```

```
PROGRAM-ID.      C2COB1.
AUTHOR.         INGO ADLUNG.
INSTALLATION.   BOEBLINGEN GERMANY.
DATE-WRITTEN.   MAY 4, 1998.
DATE-COMPILED.
```

```
ENVIRONMENT DIVISION.
```

```
CONFIGURATION SECTION.
SOURCE-COMPUTER.  IBM-370.
OBJECT-COMPUTER.  IBM-370.
```

```
DATA DIVISION.
```

```
WORKING-STORAGE SECTION.
```

```
01 SOCKET-DATA.
   05 DOMAIN      PIC S9(9) BINARY.
   05 SOCKTYPE    PIC S9(9) BINARY.
   05 PROTOCOL    PIC S9(9) BINARY.
   05 LSOCKET     PIC S9(9) BINARY.
   05 RSOCKET     PIC S9(9) BINARY.
01 SOCKADDR-IN.
   05 SIN-FAMILY  PIC S9(2) BINARY.
   05 SIN-PORT    PICTURE S9(4) BINARY.
   05 SIN-ADDR    PIC S9(9) BINARY.
   05 SIN-ZERO    PIC S9(2) BINARY OCCURS 4 TIMES VALUE 0.
01 RESULTS.
   05 RVALUE      PIC S9(9) BINARY.
   05 RDETAIL     PIC S9(9) BINARY.
01 BUFSIZE       PIC S9(9) BINARY.
```

```

01 L-COUNT      PIC S9(9) BINARY.
01 BUFFER.
   05 WORKAREA  PICTURE X(512).

PROCEDURE DIVISION.

MAIN.

*
* Create a TCP stream socket. The socket value will be
* returned in variable RVALUE.
*
* domain type AF_INET is 2
* socket type SOCK_STREAM is 1
* protocol IPPROTO_TCP is 6
*
MOVE 2 TO DOMAIN.
MOVE 1 TO SOCKETTYPE.
MOVE 6 TO PROTOCOL.

DISPLAY 'Calling C socket()'.

CALL 'TCPSOCT' USING BY CONTENT DOMAIN, SOCKETTYPE, PROTOCOL
                  BY REFERENCE RVALUE, RDETAIL.

MOVE RVALUE TO LSOCKET.

*
* Bind the socket to the local port
*
* domain type AF_INET is 2
* local port is 2000
*
MOVE 2 TO SIN-FAMILY.
MOVE 2000 TO SIN-PORT.
MOVE 0 TO SIN-ADDR.
MOVE 16 TO BUFSIZE.

DISPLAY 'Calling C bind()'.

CALL 'TCPBIND' USING BY CONTENT LSOCKET
                   BY REFERENCE SOCKADDR-IN
                   BY CONTENT BUFSIZE
                   BY REFERENCE RVALUE, RDETAIL.

*
* Convert socket to passive mode.
*
MOVE 1 TO L-COUNT.

DISPLAY 'Calling C listen()'.

CALL 'TCPLIST' USING BY CONTENT LSOCKET, L-COUNT
                  BY REFERENCE RVALUE, RDETAIL.

*
* Wait for incoming clients.
*
INITIALIZE SOCKADDR-IN.
MOVE 16 TO BUFSIZE.

DISPLAY 'Calling C accept()'.

CALL 'TCPACCP' USING BY CONTENT LSOCKET,
                   BY REFERENCE SOCKADDR-IN, BUFSIZE,
                   RVALUE, RDETAIL.

```

Introducing Socket Programming

```
*
* Receive a piece of data.
*
    MOVE RVALUE TO RSOCKET.
    MOVE 512 TO BUFSIZE.

    DISPLAY 'Calling C read()'.

    CALL 'TCPREAD' USING BY CONTENT RSOCKET
                        BY REFERENCE WORKAREA
                        BY CONTENT BUFSIZE
                        BY REFERENCE RVALUE, RDETAIL.

*
* Send the data back to the caller
*
    MOVE RVALUE TO BUFSIZE.

    DISPLAY 'Calling C write()'.

    CALL 'TCPWRITE' USING BY CONTENT RSOCKET
                        BY REFERENCE WORKAREA
                        BY CONTENT BUFSIZE
                        BY REFERENCE RVALUE, RDETAIL.

*
* Close the connection
*
    DISPLAY 'Calling C close()'.

    CALL 'TCPCLOSE' USING BY CONTENT RSOCKET
                        BY REFERENCE RVALUE, RDETAIL.

*
* Release the listen socket too.
*
    DISPLAY 'Calling C close()'.

    CALL 'TCPCLOSE' USING BY CONTENT LSOCKET
                        BY REFERENCE RVALUE, RDETAIL.

STOP RUN.
```

The next example shows the corresponding C source, providing the mapping for the socket routines. The generated object deck needs to be link-edited with the generated COBOL object deck.

```
#include <types.h>
#include <unistd.h>
#include <in.h>
#include <socket.h>
#include <errno.h>
#include <stdio.h>

#pragma linkage( cob2c_socket, COBOL)
#pragma linkage( cob2c_bind, COBOL)
#pragma linkage( cob2c_listen, COBOL)
#pragma linkage( cob2c_accept, COBOL)
#pragma linkage( cob2c_read , COBOL)
#pragma linkage( cob2c_write, COBOL)
#pragma linkage( cob2c_close, COBOL)

#pragma map( cob2c_socket, "TCPSOCKET")
#pragma map( cob2c_bind, "TCPBIND" )
#pragma map( cob2c_listen, "TCPLIST" )
#pragma map( cob2c_accept, "TCPACCP" )
```

```

#pragma map( cob2c_read,  "TCPREAD" )
#pragma map( cob2c_write, "TCPWRITE")
#pragma map( cob2c_close, "TCPCLOSE")

void cob2c_socket( int  domain,
                  int  type,
                  int  protocol,
                  int *psocket,
                  int *perr)
{
    printf(
        "socket() called, domain : %d, type : %d, protocol : %d\n",
        domain, type, protocol);

    *psocket = socket( domain, type, protocol);
    *perr    = errno;
}

void cob2c_bind( int          socket,
                const struct sockaddr *address,
                size_t        len,
                int           *pvalue,
                int           *perr)
{
    struct sockaddr_in * sockin = (struct sockaddr_in *)address;

    *pvalue = bind( socket, address, len);
    *perr   = errno;
}

void cob2c_listen( int  socket,
                  int  backlog,
                  int  *pvalue,
                  int  *perr)
{
    *pvalue = listen( socket, backlog);
    *perr   = errno;
}

void cob2c_accept( int          socket,
                  struct sockaddr *address,
                  size_t        *len,
                  int           *pvalue,
                  int           *perr)
{
    *pvalue = accept( socket, address, len);
    *perr   = errno;
}

void cob2c_read( int  socket,
                void *buffer,
                size_t len,
                size_t *pvalue,
                int   *perr)
{
    *pvalue = read( socket, buffer, len);
    *perr   = errno;
}

void cob2c_write( int          socket,
                 const void *buffer,
                 size_t        len,
                 size_t        *pvalue,
                 int           *perr)
{
    *pvalue = write( socket, buffer, len);
}

```

Introducing Socket Programming

```
        *perr = errno;
    }

    void cob2c_close( int      socket,
                    size_t   *pvalue,
                    int      *perr)
    {
        *pvalue = close( socket);
        *perr = errno;
    }
}
```

Exploiting the EZASMI/EZASOKET Programming Interfaces

Applications on z/VSE can use the EZASMI and / or EZASOKET programming interfaces. These programming interfaces are provided both for programming in a batch environment and in a CICS Transaction Server environment.

Following are a few sample programs that show a simple usage of these interfaces. To reduce complexity they do not include any error recovery necessary, if communication problems show up. The first sample shows a client assembler program, which uses the EZASMI macro interface:

Figure 13. Sample Program Using EZASMI Macro (Synchronously)

```
*          PRINT NOGEN
*****
*
*  MODULE NAME:  SAMPCLIE
*
*  FUNCTION:  Sample program for usage of EZASMI macro
*             (Client part)
*
*  ATTRIBUTES:  NON-REUSABLE
*
*  REGISTER USAGE:
*      R3 = BASE REG
*      R13 = SAVE AREA
*
*  INPUT:  NONE
*  OUTPUT:  NONE
*
*****
-----*
*  START OF EXECUTABLE CODE
*-----*
SAMPCLIE START X'78'          adjust addr behind part savearea
SAMPCLIE AMODE ANY
SAMPCLIE RMODE ANY
          USING *,R15          Use Entry Register for base
          B      SAMPCLST
          DC     C'SAMPCLST-00/06/23'
*
SAMPCLST DS      0H
          STM    R14,R12,12(R13)  Save Caller's Registers
          LR     R3,R15           Change base register to R3
          DROP   R15             Done with this register
          USING  SAMPCLIE,R3     Tell assembler about new base
          LA     R15,MYSAVE      Get addr of own save area
          ST     R13,MYSAVE+4     Save caller's save area addr
          ST     R15,8(R13)      Save own save area addr
          LR     R13,R15         Load Reg13
*****
*
*          Issue INITAPI to connect to interface
*
```

Introducing Socket Programming

```

*****
EZASMI TYPE=INITAPI,      Issue INITAPI Macro      X
      MAXSOC=MAXSOC,      Max number of sockets (in) X
      MAXSNO=MAXSNO,      Greatest Descr Number used (out)X
      ERRNO=ERRNO,        ERRNO field      X
      RETCODE=RETCODE     RETCODE field

*
*****
* Issue SOCKET call *
*****
EZASMI TYPE=SOCKET,      Issue SOCKET call      X
      AF='INET',          INTERNET family   X
      SOCTYPE='STREAM',   Stream socket          X
      PROTO=PROTOCOL,     protocol              X
      ERRNO=ERRNO,        ERRNO field            X
      RETCODE=RETCODE     RETCODE field
MVC SOCKET1,RETCODE     Save the socket descriptor

*
*****
* Issue CONNECT *
*****
EZASMI TYPE=CONNECT,    Issue CONNECT call      X
      S=SOCKET1+2,        socket descriptor (halfword) X
      NAME=SAMPSEV,        to SAMPSEV program      X
      ERRNO=ERRNO,        ERRNO field            X
      RETCODE=RETCODE     RETCODE field

*
*****
* Issue WRITE on connected socket *
*****
EZASMI TYPE=WRITE,      Issue WRITE call        X
      S=SOCKET1+2,        on this socket          X
      NBYTE=MSG1L,        Length of first message  X
      BUF=MSG1,           Text of first message   X
      ERRNO=ERRNO,        ERRNO field            X
      RETCODE=RETCODE     RETCODE field
B READ1                  go and read

*
MSG1L DC F'40'
MSG1  DC CL40'DATA SENT FROM SAMPCLIE.'
*****
* Issue READ on connected socket *
*****
READ1 EZASMI TYPE=READ,  Issue READ call        X
      S=SOCKET1+2,        on this socket          X
      NBYTE=READBL,       length of read buffer   X
      BUF=READB,          address of read buffer  X
      ERRNO=ERRNO,        ERRNO field            X
      RETCODE=RETCODE     RETCODE field

*
*****
* Issue CLOSE on connected socket *
*****
EZASMI TYPE=CLOSE,      Issue CLOSE call        X
      S=SOCKET1+2,        on this socket          X
      ERRNO=ERRNO,        ERRNO field            X
      RETCODE=RETCODE     RETCODE field

*
*****
* Issue TERMAPI to disconnect interface *
*****
EZASMI TYPE=TERMAPI     Issue TERMAPI call

*
EOJ
EJECT
*****

```

Introducing Socket Programming

```

* CONSTANTS/VARIABLES USED BY THIS PROGRAM *
*-----*
          EZASMI TYPE=TASK,STORAGE=CSECT Task Storage Area
MYSAVE   DC    18F'0'          Register Save Area
ERRNO    DC    F'0'
RETCODE  DC    F'0'
*-----*
* INITAPI macro parms *
*-----*
MAXSOC   DC    H'256'          MAXSOC parm value
MAXSNO   DC    F'0'          Highest socket descriptor avail
*-----*
* SOCKET macro parms *
*-----*
PROTOCOL DC    F'0'          default protocol
SOCKET1  DC    F'0'          save area for socket descriptor
*
*-----*
* CONNECT Macro Params*
*-----*
          CNOP  0,4
SAMPSEV  DC    0CL16' '      SOCKET NAME structure of SERVER
          DC    AL2(2)        FAMILY (AF-INET)
          DC    H'4000'       Port of SAMPSEV
          DC    AL1(9),AL1(164),AL1(155),AL1(122) IP-Addr of SAMPSEV
          DC    XL8'00'       RESERVED
*
*-----*
* READ MACRO PARMS *
*-----*
READBL   DC    F'40'          SIZE OF READ BUFFER
READB    DC    CL40' '       READ BUFFER
**----- register equates -----**
R0       EQU    0
R1       EQU    1
R2       EQU    2
R3       EQU    3
R4       EQU    4
R5       EQU    5
R6       EQU    6
R7       EQU    7
R8       EQU    8
R9       EQU    9
R10      EQU    10
R11      EQU    11
R12      EQU    12
R13      EQU    13
R14      EQU    14
R15      EQU    15
*
          END    SAMPCLIE

```

The second sample shows a server assembler program using the asynchronous EZASMI macro interface:

Figure 14. Sample Program Using EZASMI Macro (Asynchronously)

```

*          PRINT NOGEN
*****
*
*   MODULE NAME:  SAMPSEV
*
*   FUNCTION:  Sample Program for EZASMI (asynchronous) macro usage
*             (Server Part)
*
*****

```



```

*   ATTRIBUTES: NON-REUSABLE                               *
*                   NON-LE Enabled                         *
*   REGISTER USAGE:                                       *
*       R3 = BASE REG 1                                    *
*       R13 = SAVE AREA                                    *
*   INPUT: NONE                                           *
*   OUTPUT: NONE                                          *
*   *-----*
*   * START OF EXECUTABLE CODE                             *
*   *-----*
SAMPSEV START X'78'                                     adjust addr behind part savearea
SAMPSEV AMODE 31
SAMPSEV RMODE ANY
        USING *,R15                                     Use Entry Register for base
        B     SAMPSTRT
        DC    C'SAMPSEST-00/06/23'
*
SAMPSTRT DS    0H
        STM   R14,R12,12(R13)                          Save Caller's Registers
        LR    R3,R15                                    Change base register to R3
        DROP  R15                                       Done with this register
        USING SAMPSEV,R3                                Tell assembler about new base
        LA    R15,MYSAVE                                Get addr of own save area
        ST    R13,MYSAVE+4                              Save caller's save area addr
        ST    R15,8(R13)                                Save own save area addr
        LR    R13,R15                                   Load Reg13
*-----*
*   Issue INITAPI to connect to interface                 *
*-----*
        EZASMI TYPE=INITAPI,                            Issue INITAPI Macro           X
                MAXSOC=MAXSOC,                          Max number of sockets (in)    X
                MAXSNO=MAXSNO,                          Greatest Descr Number used (out)X
                ASYNC='ECB',                             asynchronous ECB processing   X
                ERRNO=ERRNO,                             ERRNO field                   X
                RETCODE=RETCODE                          RETCODE field
*
*-----*
*   Issue SOCKET call                                    *
*-----*
        XC    ECB,ECB
        EZASMI TYPE=SOCKET,                              Issue SOCKET call             X
                AF='INET',                                INTERNET family                X
                SOCTYPE='STREAM',                        Stream socket                  X
                PROTO=PROTOCOL,                          protocol                       X
                ECB=*ECBA,                                wait on this ECB              X
                ERRNO=ERRNO,                              ERRNO field                   X
                RETCODE=RETCODE                          RETCODE field
*
        WAIT  ECB                                        Wait on ECB
        MVC   SOCKET1,RETCODE                          Save the socket descriptor
*-----*
*   Issue BIND call                                     *
*-----*
        XC    ECB,ECB                                    Clear ECB
        EZASMI TYPE=BIND,                                Issue BIND call               X
                S=SOCKET1+2,                              socket descriptor             X
                NAME=MYNAME,                              Name structure                 X
                ECB=*ECBA,                                wait on this ECB              X
                ERRNO=ERRNO,                              ERRNO field                   X
                RETCODE=RETCODE                          RETCODE field
*
        WAIT  ECB                                        Wait on ECB
*-----*

```

Introducing Socket Programming

```

*          Issue LISTEN                                     *
*****
XC      ECB,ECB          Clear ECB
EZASMI TYPE=LISTEN,    Issue LISTEN call          X
      S=SOCKET1+2,      socket descriptor        X
      BACKLOG=BACKLOG,  max number of backlog msgs X
      ECB=*ECBA,        wait on this ECB          X
      ERRNO=ERRNO,      ERRNO field                X
      RETCODE=RETCODE   RETCODE field

*
      WAIT ECB          Wait on ECB
*****
*          Issue ACCEPT                                     *
*****
XC      ECB,ECB          Clear ECB
EZASMI TYPE=ACCEPT,    Issue ACCEPT call          X
      S=SOCKET1+2,      socket descriptor        X
      NAME=NAMECLIE,    Name structure of client  X
      ECB=*ECBA,        wait on this ECB          X
      ERRNO=ERRNO,      ERRNO field                X
      RETCODE=RETCODE   RETCODE field

*
      WAIT ECB          Wait on ECB
      MVC SOCKETN,RETCODE Save RETCODE (New Socket Descr.)
*****
*          Issue READ                                       *
*****
XC      ECB,ECB          Clear ECB
EZASMI TYPE=READ,      Issue READ call          X
      S=SOCKETN+2,      on this socket            X
      NBYTE=READBUFL,   length of read buffer    X
      BUF=READBUF,      address of read buffer    X
      ECB=*ECBA,        wait on this ECB          X
      ERRNO=ERRNO,      ERRNO field                X
      RETCODE=RETCODE   RETCODE field

*
      WAIT ECB          Wait on ECB
*****
*          Issue WRITE on connected socket                 *
*****
XC      ECB,ECB          Clear ECB
EZASMI TYPE=WRITE,     Issue WRITE call          X
      S=SOCKETN+2,      on this socket            X
      NBYTE=MSGL,       Length of first message   X
      BUF=MSG,          Text of first message     X
      ECB=*ECBA,        wait on this ECB          X
      ERRNO=ERRNO,      ERRNO field                X
      RETCODE=RETCODE   RETCODE field

*
      WAIT ECB          Wait on ECB
      B CLOSE1
*
MSGL   DC F'40'
MSG    DC CL40'SAMPSEV RECEIVED YOUR DATA.'
*****
*          Issue CLOSE socket                             *
*****
CLOSE1 XC      ECB,ECB          Clear ECB
EZASMI TYPE=CLOSE,     Issue CLOSE call          X
      S=SOCKETN+2,      on this socket            X
      ECB=*ECBA,        wait on this ECB          X
      ERRNO=ERRNO,      ERRNO field                X
      RETCODE=RETCODE   RETCODE field

*
      WAIT ECB          Wait on ECB
*****
*          Issue CLOSE socket                             *

```

Introducing Socket Programming

```

*****
          XC   ECB,ECB                Clear ECB
          EZASMI TYPE=CLOSE,          Issue CLOSE call           X
              S=SOCKET1+2,           on this socket             X
              ECB=*ECBA,             wait on this ECB           X
              ERRNO=ERRNO,           ERRNO field                 X
              RETCODE=RETCODE        RETCODE field
*
          WAIT ECB                    Wait on ECB
*****
* Issue TERMAPI to disconnect interface *
*****
          EZASMI TYPE=TERMAPI        Issue TERMAPI Call
          EOJ
          EJECT
*-----*
* CONSTANTS/VARIABLES USED BY THIS PROGRAM *
*-----*
          EZASMI TYPE=TASK,STORAGE=CSECT Task Storage Area
MYSAVE   DC   18F'0'                 Register Save Area
ERRNO    DC   F'0'
RETCODE  DC   F'0'
ECBA     DC   A(ECB)                POINTER to ECB
ECB      DC   F'0'                 ECB
ECBX     DC   XL156'00'             ECB Extension Area
*
*-----*
* INITAPI macro parms *
*-----*
MAXSOC   DC   H'80'                 MAXSOC PARM VALUE
MAXSNO   DC   F'0'                 Highest Socket Descriptor avail
*
*-----*
* SOCKET macro parms *
*-----*
PROTOCOL DC   F'0'                 default protocol
SOCKET1  DC   F'0'                 savearea for socket descriptor
SOCKETN  DC   F'0'                 savearea for socket descriptor
*
*-----*
* BIND MACRO PARMS *
*-----*
          CNOP  0,4
MYNAME   DC   0CL16' '             SOCKET NAME STRUCTURE
          DC   AL2(2)              FAMILY (AF-INET)
MYPORT   DC   H'4000'             bind to this port
MYADDR   DC   AL1(9),AL1(164),AL1(155),AL1(122) and IP address
          DC   XL8'00'             RESERVED
*
*-----*
* LISTEN PARMS *
*-----*
BACKLOG  DC   F'5'                 BACKLOG
*
*-----*
* ACCEPT PARMS *
*-----*
NAMECLIE DC   0CL16' '             SOCKET NAME STRUCTURE of client
          DC   AL2(2)              FAMILY
PORTCLIE DC   H'0'
ADDRCLIE DC   F'0'
          DC   XL8'00'             RESERVED
*
*-----*
* READ MACRO PARMS *
*-----*
READBUFL DC   F'40'                 SIZE OF READ BUFFER
READBUF  DC   CL40'none'           READ BUFFER
* ----- register equates -----
R0       EQU  0

```

Introducing Socket Programming

```

R1      EQU 1
R2      EQU 2
R3      EQU 3
R4      EQU 4
R5      EQU 5
R6      EQU 6
R7      EQU 7
R8      EQU 8
R9      EQU 9
R10     EQU 10
R11     EQU 11
R12     EQU 12
R13     EQU 13
R14     EQU 14
R15     EQU 15
*
        END  SAMPSEV

```

Of course, there is no real need for this simple program to use the asynchronous interface. Asynchronous processing can be helpful, if the program wants to perform other tasks while waiting on a socket call to complete.

The next sample shows a similar server program written in COBOL using the EZASOKET call interface:

Figure 15. Sample Program Using EZASOKET Call Using COBOL

```

CBL LIB APOST RMODE(ANY)                                SAM00010
IDENTIFICATION DIVISION.                                SAM00020
                                                         SAM00030
PROGRAM-ID.      SAMPSEV                                  SAM00040
AUTHOR.          HEINZ HAGEDORN                          SAM00050
INSTALLATION.    HIER.                                    SAM00060
DATE-WRITTEN.    June 23, 2000                           SAM00070
DATE-COMPILED.   .                                       SAM00080
                                                         SAM00090
ENVIRONMENT DIVISION.                                    SAM00100
                                                         SAM00110
CONFIGURATION SECTION.                                   SAM00120
                                                         SAM00130
SOURCE-COMPUTER.  IBM-370.                                SAM00140
OBJECT-COMPUTER.  IBM-370.                                SAM00150
                                                         SAM00160
DATA DIVISION.                                          SAM00170
                                                         SAM00180
                                                         SAM00190
WORKING-STORAGE SECTION.                                SAM00200
01 SOKET-FUNCTIONS.                                     SAM00210
   02 SOKET-ACCEPT      PIC X(16) VALUE 'ACCEPT          ' . SAM00220
   02 SOKET-BIND        PIC X(16) VALUE 'BIND            ' . SAM00230
   02 SOKET-CLOSE       PIC X(16) VALUE 'CLOSE           ' . SAM00240
   02 SOKET-CONNECT     PIC X(16) VALUE 'CONNECT        ' . SAM00250
   02 SOKET-INITAPI     PIC X(16) VALUE 'INITAPI         ' . SAM00260
   02 SOKET-LISTEN      PIC X(16) VALUE 'LISTEN          ' . SAM00270
   02 SOKET-READ        PIC X(16) VALUE 'READ            ' . SAM00280
   02 SOKET-SOCKET      PIC X(16) VALUE 'SOCKET          ' . SAM00290
   02 SOKET-TERMAPI     PIC X(16) VALUE 'TERMAPI         ' . SAM00300
   02 SOKET-WRITE       PIC X(16) VALUE 'WRITE          ' . SAM00310
01 SOKET-FUNCT        PIC X(16) VALUE '                 ' . SAM00320
01 SOKET-ADDR.                                               SAM00330
   02 SOCK-FAMILY       PIC 9(4) BINARY.                  SAM00340
   02 SOCK-PORT         PIC 9(4) BINARY.                  SAM00350
   02 SOCK-IPADDR       PIC 9(8) BINARY.                  SAM00360
   02 SOCK-ZERO         PIC X(8).                          SAM00370
01 SOKET-ID           PIC 9(4) BINARY.                    SAM00380

```

```

01 SOKET-ID-NEW          PIC 9(4) BINARY.          SAM00390
01 MAXSOC                PIC 9(4) BINARY.          SAM00400
01 IDENT.                SAM00410
    02 TCPNAME           PIC X(8).                 SAM00420
    02 ADSNAME           PIC X(8).                 SAM00430
01 SUBTASK               PIC X(8).                 SAM00440
01 MAXSNO                PIC 9(8) BINARY.          SAM00450
                                SAM00460
01 INBUFFL              PIC 9(8) COMP VALUE 40.    SAM00570
                                SAM00580
01 AF-INET              PIC 9(8) COMP VALUE 2.    SAM00470
01 SOCTYPE               PIC 9(8) COMP VALUE 1.    SAM00480
01 PROTO                 PIC 9(8) COMP VALUE 0.    SAM00490
01 BACKLOG               PIC 9(8) COMP VALUE 5.    SAM00500
01 RETCODE               PIC S9(8) BINARY.         SAM00510
01 ERRNO                 PIC 9(8) BINARY.         SAM00520
01 MSG001                PIC X(34)                SAM00530
    VALUE IS ' ... SAMPSEV received your data.'.  SAM00540
01 MSG001L              PIC 9(8) COMP VALUE 34.    SAM00550
01 INBUFF                PIC X(40) VALUE IS ' '.    SAM00560
PROCEDURE DIVISION.    SAM00590
                                SAM00600
BEGIN.                  SAM00610
                                SAM00620
*-----*                    SAM00630
*   CALL EZASOKET - function = INITAPI           *   SAM00640
*           input      = SUBTASK blank          *   SAM00650
*-----*                    SAM00660
                                SAM00670
    MOVE SOKET-INITAPI TO SOKET-FUNCT.          SAM00680
    MOVE '          ' TO TCPNAME.                SAM00690
    MOVE '          ' TO SUBTASK.                SAM00700
    MOVE 99 TO MAXSOC.                           SAM00710
    MOVE 0 TO RETCODE.                           SAM00720
    MOVE 0 TO ERRNO.                             SAM00730
                                SAM00740
    CALL 'EZASOKET' USING SOKET-FUNCT MAXSOC IDENT SUBTASK
                                SAM00750
                                MAXSNO ERRNO RETCODE. SAM00760
                                SAM00770
*-----*                    SAM00780
*   CALL EZASOKET - function = SOCKET           *   SAM00790
*-----*                    SAM00800
                                SAM00810
    MOVE SOKET-SOCKET TO SOKET-FUNCT.           SAM00820
    MOVE 0 TO RETCODE.                           SAM00830
    MOVE 0 TO ERRNO.                             SAM00840
                                SAM00850
    CALL 'EZASOKET' USING SOKET-FUNCT AF-INET SOCTYPE PROTO
                                SAM00860
                                ERRNO RETCODE.    SAM00870
                                SAM00880
    MOVE RETCODE TO SOKET-ID.                   SAM00890
                                SAM00900
*-----*                    SAM00910
*   CALL EZASOKET - function = BIND             *   SAM00920
*           input      = SOKET-ID, SOKET-ADDR *   SAM00930
*-----*                    SAM00940
                                SAM00950
    MOVE SOKET-BIND TO SOKET-FUNCT.             SAM00960
    MOVE AF-INET TO SOCK-FAMILY.                SAM00970
    MOVE 4000 TO SOCK-PORT.                     SAM00980
    MOVE 0 TO SOCK-IPADDR.                      SAM00990
    MOVE 0 TO RETCODE.                           SAM01000
    MOVE 0 TO ERRNO.                             SAM01010
                                SAM01020
    CALL 'EZASOKET' USING SOKET-FUNCT SOKET-ID SOKET-ADDR
                                SAM01030
                                ERRNO RETCODE.    SAM01040
                                SAM01050

```

Introducing Socket Programming

```
*-----*
*   CALL EZASOKET - function = LISTEN   *
*                   input  = backlog=5   *
*-----*

MOVE SOKET-LISTEN TO SOKET-FUNCT.
MOVE 0 TO RETCODE.
MOVE 0 TO ERRNO.

CALL 'EZASOKET' USING SOKET-FUNCT SOKET-ID BACKLOG
ERRNO RETCODE.

*-----*
*   CALL EZASOKET - function = ACCEPT   *
*                   input  = SOKET-ID   *
*-----*

MOVE SOKET-ACCEPT TO SOKET-FUNCT.
MOVE 0 TO RETCODE.
MOVE 0 TO ERRNO.

CALL 'EZASOKET' USING SOKET-FUNCT SOKET-ID SOKET-ADDR
ERRNO RETCODE.

MOVE RETCODE TO SOKET-ID-NEW.

*-----*
*   CALL EZASOKET - function = READ     *
*-----*

MOVE SOKET-READ      TO SOKET-FUNCT.
MOVE 0 TO RETCODE.
MOVE 0 TO ERRNO.

MOVE LOW-VALUES TO INBUFF.
CALL 'EZASOKET' USING SOKET-FUNCT SOKET-ID-NEW INBUFFL
INBUFF ERRNO RETCODE.

*-----*
*   CALL EZASOKET - function = WRITE    *
*-----*

MOVE SOKET-WRITE     TO SOKET-FUNCT.
MOVE 0 TO RETCODE.
MOVE 0 TO ERRNO.

CALL 'EZASOKET' USING SOKET-FUNCT SOKET-ID-NEW MSG001L
MSG001 ERRNO RETCODE.

*-----*
*   CALL EZASOKET - function = CLOSE    *
*-----*

MOVE SOKET-CLOSE     TO SOKET-FUNCT.
MOVE 0 TO RETCODE.
MOVE 0 TO ERRNO.

CALL 'EZASOKET' USING SOKET-FUNCT SOKET-ID-NEW
ERRNO RETCODE.

*-----*
*   CALL EZASOKET - function = CLOSE    *
*-----*

MOVE SOKET-CLOSE     TO SOKET-FUNCT.
```

SAM01060
SAM01070
SAM01080
SAM01090
SAM01100
SAM01110
SAM01120
SAM01130
SAM01140
SAM01150
SAM01160
SAM01170
SAM01180
SAM01190
SAM01200
SAM01210
SAM01220
SAM01230
SAM01240
SAM01250
SAM01260
SAM01270
SAM01280
SAM01290
SAM01300
SAM01310
SAM01320
SAM01330
SAM01340
SAM01350
SAM01360
SAM01370
SAM01380
SAM01390
SAM01400
SAM01410
SAM01420
SAM01430
SAM01440
SAM01450
SAM01460
SAM01470
SAM01480
SAM01490
SAM01500
SAM01510
SAM01520
SAM01530
SAM01540
SAM01550
SAM01560
SAM01570
SAM01580
SAM01590
SAM01600
SAM01610
SAM01620
SAM01630
SAM01640
SAM01650
SAM01660
SAM01670
SAM01680
SAM01690
SAM01700
SAM01710
SAM01720

MOVE 0 TO RETCODE.	SAM01730
MOVE 0 TO ERRNO.	SAM01740
	SAM01750
CALL 'EZASOKET' USING SOKET-FUNCT SOKET-ID	SAM01760
ERRNO RETCODE.	SAM01770
	SAM01780
	SAM01790
-----	SAM01800
* CALL EZASOKET - function = TERMAPI *	SAM01810
-----	SAM01820
	SAM01830
MOVE SOKET-TERMAPI TO SOKET-FUNCT.	SAM01840
	SAM01850
CALL 'EZASOKET' USING SOKET-FUNCT.	SAM01860
	SAM01870
	SAM01880
STOP RUN.	SAM01890
	SAM01900
END PROGRAM SAMPSEV.	SAM01910

LE/VSE 1.4 C Socket Programming

General C Programming Considerations

While the Language Environment intends to cover the same functionality as OS/390® and z/OS, VM/ESA and z/VM in their Language Environment based C runtime libraries, the actual behavior of the C Socket interface routines is dependent on the TCP/IP product that is used with this interface. Therefore a programmer porting an application from another System z operating system environment may eventually find that the VSE socket interfaces require special attention. However, a programmer porting an application for example from z/OS might not expect that source code modifications are eventually required.

The following list is aimed to identify the programming areas that might require special attention, especially when porting applications.

- Applications using the C socket interfaces can safely be written for CICS environments, because the LE Socket support dynamically determines the execution environment and uses CICS services where appropriate, for example *EXEC CICS WAIT* instead of the *VSE WAIT* macro. This implies that, in contrast to z/OS CICS Sockets, no special initialization and termination services need to be called in a C program intending to run in a CICS environment. It is therefore possible to write communication routines, either be called from a batch or CICS application.
- LE/VSE 1.4 does not support multitasking environments if more than a single subtask is supposed to run LE enabled code. This is caused by the fact that z/VSE doesn't support POSIX threads, nor does it support more than 31 subtasks per VSE partition. Nor is it possible to fork() a new process as the necessary UNIX alike system interfaces are not available. Nevertheless, it is possible to have multiple VSE subtasks running, but only one of them can execute LE enabled code.

Therefore, if coding Daemon applications, intending to serve multiple clients concurrently, it is necessary not to become wait bound during an attempt to read data or when waiting for clients to connect. Instead, it is recommended to use *select()* or *selectex()* and check which socket shows activity before calling *recv()* or *accept()* as these calls may block if the no outstanding requests can be served on a specific socket connection at the time of the call.

- Other TCP/IP implementations provide *ioctl()* or *fcntl()* interfaces that allow to operate the socket interfaces in **blocking** or **non-blocking** mode. In blocking

Introducing Socket Programming

mode, a call for example to `recv()` will suspend the task until data for the Socket used arrives. In non-blocking mode, the routine would return -1 and the *errno* variable would be set to **EWOULDBLOCK**. The application can then choose either to process something different, or use `select()` or `selectex()` to wait on one or multiple sockets showing activity.

While TCP/IP for VSE/ESA doesn't provide this mechanism natively, the TCP/IP support for the LE C socket API provides the necessary support. However, the following restrictions apply:

- In a fully BSD conforming stack implementation a default send and receive buffer are allocated for the TCP protocol. If the send buffer was filled faster than the stack being able to transmit the data over the network the `send()` or `sendto()` calls would block. In non-blocking mode those calls would return an error value **EWOULDBLOCK** instead. A call to `select()` or `selectex()` with the write bit string set returns immediately if any buffer space is available. TCP/IP for VSE/ESA doesn't work that way, but buffers any unsent data in partition GETVIS until the GETVIS is exhausted. If there isn't any GETVIS space left to buffer the unsent data the `send()` or `sendto()` calls block. Calling `select()` or `selectex()` with the write bit string do not indicate whether any send buffer space is available, but block until all socket specific unsent data is put on the network.
- Some LE/VSE C socket routines require special attention, as either the TCP/IP implementation behaves differently on z/VSE than on other platforms or only a subset of the functionality is implemented. These differences are described in the individual function descriptions in Chapter 10, "TCP/IP Support for the LE/VSE C Socket Interface," on page 85 as "Stack Characteristics".

Messages

The following list covers the messages possibly be issued by the LE C Socket interface routines. The messages may either issued by the C runtime library, or by phase \$EDCTCPV when mapping LE C Socket calls to the TCP/IP for VSE/ESA BSD-C Socket interface routines.

Messages issued by the LE/VSE 1.4 C runtime Library

- EDCT001I Unable to load phase \$EDCTCPV
Phase \$EDCTCPV could not be loaded. Application is canceled with message CEE3322C.
Most probably the TCP/IP product library (PRD2.TCPIPC) is missing in the application's partition LIBDEF search chain.
- EDCT002I xxxxxxxxx implementation not found
Phase \$EDCTCPV does not contain the body of TCP/IP function xxxxxxxxx due to a build error. Application is canceled with CEE3322C.
- EDCT003I Unsupported C-Runtime function called
Application contains calls to C runtime functions that are not supported in LE/VSE 1.4. Application is canceled with CEE3322C.
This should only happen if a program compiled and prelinked on z/OS or z/VM was link-edited on z/VSE. The precompile step on z/OS or z/VM[®] has included a stub routine to a C runtime function not supported in the LE/VSE 1.4 runtime environment.

Message issued by Phase \$EDCTCPV

- EDCV001I TCP/IP function xxxxxxxxx not implemented

The application has called a TCP/IP socket routine that is not implemented by the TCP/IP programming interface. The application is passed back an appropriate function specific return code. Program execution continues.

Either the function is currently not supported by TCP/IP for VSE/ESA or the partition LIBDEF chain doesn't list the TCP/IP product library prior to the LE product library. Phase \$EDCTCPV from the TCP/IP for VSE/ESA product library (PRD2.TCPIPC) must be found prior to the same phase found in the LE/VSE product library (PRD2.SCEEBASE).

- EDCV002I Unexpected TCP/IP error code: nnnn

The TCP/IP product returned an unexpected error code, the TCP/IP support for the LE C interfaces is not capable to handle. Error value EOPNOTSUPP is passed back to the calling application instead.

TCP/IP Functions Supported by z/VSE

As mentioned in previous chapters, the C socket interface provided by the VSE Language Environment 1.4 is not implemented in the Language Environment (LE) itself, but is mapped to the programming interfaces that come with your TCP/IP stack.

Table 3 covers the LE C Socket routines documented in Chapter 10, “TCP/IP Support for the LE/VSE C Socket Interface,” on page 85, and shows if they are currently available through one of the following TCP/IP stacks:

- TCP/IP for VSE/ESA
- IPv6/VSE
- Linux Fast Path

The table only states, if a function is generally supported. For details on differences and special characteristics refer to the individual stack and function descriptions.

Table 3 also lists the corresponding EZASMI macros and EZASOCKET calls supported by z/VSE. These interfaces are also available within z/OS, with minor differences. For details refer to:

- Chapter 11, “Using the CALL Instruction Application Programming Interface (EZASOCKET API),” on page 199
- Chapter 12, “Using the Macro Application Programming Interface (EZASMI API),” on page 291

Table 3. Supported call functions by Interface and TCP/IP Stack

Call Function	Interface			TCP/IP Stack		
	EZASMI	EZASOCKET	LE/VSE	TCP/IP for VSE/ESA	IPv6/VSE	Linux Fast Path
accept()	ACCEPT	ACCEPT	yes	yes	yes	yes
aio_cancel()	CANCEL	no	yes	yes	yes	yes
aio_error()	no	no	yes	no	yes	yes
aio_read()	no	no	yes	no	yes	yes
aio_return()	no	no	yes	no	yes	yes
aio_suspend()	no	no	yes	no	yes	yes
aio_write()	no	no	yes	no	yes	yes
bind()	BIND	BIND	yes	yes	yes	yes

Introducing Socket Programming

Table 3. Supported call functions by Interface and TCP/IP Stack (continued)

Call Function	Interface			TCP/IP Stack		
	EZASMI	EZASOCKET	LE/VSE	TCP/IP for VSE/ESA	IPv6/VSE	Linux Fast Path
close()	CLOSE	CLOSE	yes	yes	yes	yes
connect()	CONNECT	CONNECT	yes	yes	yes	yes
endhostent()	no	no	yes	no	no	yes
endnetent()	no	no	yes	no	no	yes
endprotoent()	no	no	yes	no	no	yes
endservent()	no	no	yes	no	no	yes
fcntl()	FCNTL	FCNTL	yes	yes	yes	yes
freeaddrinfo()	FREEADDRINFO	FREEADDRINFO	yes	no	yes	yes
gai_strerror()	no	no	yes	no	yes	yes
getaddrinfo()	GETADDRINFO	GETADDRINFO	yes	no	yes	yes
getclientid()	GETCLIENTID	GETCLIENTID	yes	yes	yes	yes
gethostbyaddr()	GETHOSTBYADDR	GETHOSTBYADDR	yes	yes	yes	yes
gethostbyname()	GETHOSTBYNAME	GETHOSTBYNAME	yes	yes	yes	yes
gethostent()	no	no	yes	no	no	yes
gethostid()	GETHOSTID	GETHOSTID	yes	yes	yes	yes
gethostname()	GETHOSTNAME	GETHOSTNAME	yes	yes	yes	yes
getibmopt()	GETIBMOPT	GETIBMOPT	yes	no	yes	yes
getnameinfo()	GETNAMEINFO	GETNAMEINFO	yes	no	yes	yes
getnetbyaddr()	no	no	yes	no	no	yes
getnetbyname()	no	no	yes	no	no	yes
getnetent()	no	no	yes	no	no	yes
getpeername()	GETPEERNAME	GETPEERNAME	yes	yes	yes	yes
getprotobyname()	no	no	yes	no	no	yes
getprotobynumber()	no	no	yes	no	no	yes
getprotoent()	no	no	yes	no	no	yes
getservbyname()	no	no	yes	no	yes	yes
getservbyport()	no	no	yes	no	yes	yes
getservent()	no	no	yes	no	no	yes
getsockname()	GETSOCKNAME	GETSOCKNAME	yes	yes	yes	yes
getsockopt()	GETSOCKOPT	GETSOCKOPT	yes	yes	yes	yes
givesocket()	GIVESOCKET	GIVESOCKET	yes	yes	yes	yes
gsk_free_memory()	GSKFREEMEM	GSKFREEMEM	yes	yes	no	yes
gsk_get_cipher_info()	GSKGETCIPHINF	GSKGETCIPHINF	yes	yes	no	yes
gsk_get_dn_by_label()	GSKGETDNBYLAB	GSKGETDNBYLAB	yes	yes	no	yes
gsk_initialize()	GSKINIT	GSKINIT	yes	yes	no	yes
gsk_secure_soc_close()	GSKSSOCCLOSE	GSKSSOCCLOSE	yes	yes	no	yes
gsk_secure_soc_init()	GSKSSOCINIT	GSKSSOCINIT	yes	yes	no	yes

Table 3. Supported call functions by Interface and TCP/IP Stack (continued)

Call Function	Interface			TCP/IP Stack		
	EZASMI	EZASOCKET	LE/VSE	TCP/IP for VSE/ESA	IPv6/VSE	Linux Fast Path
gsk_secure_soc_read()	GSKSSOCREAD	GSKSSOCREAD	yes	yes	no	yes
gsk_secure_soc_reset()	GSKSSOCRESET	GSKSSOCRESET	yes	yes	no	yes
gsk_secure_soc_write()	GSKSSOCWRITE	GSKSSOCWRITE	yes	yes	no	yes
gsk_uninitialize()	GSKUNINIT	GSKUNINIT	yes	yes	no	yes
gsk_user_set()	no	no	yes	no	no	yes
htonl()	no	no	yes	yes	yes	yes
htons()	no	no	yes	yes	yes	yes
if_freenameindex()	no	no	yes	no	no	yes
if_indextoname()	no	no	yes	no	no	yes
if_nameindex()	no	no	yes	no	no	yes
if_nametoindex()	no	no	yes	no	no	yes
inet_addr()	no	no	yes	yes	yes	yes
inet_lnaof()	no	no	yes	yes	no	yes
inet_makeaddr()	no	no	yes	yes	no	yes
inet_netof()	no	no	yes	yes	no	yes
inet_network()	no	no	yes	yes	no	yes
inet_ntoa()	no	no	yes	yes	yes	yes
inet_ntop()	NTOP	NTOP	yes	no	yes	yes
inet_pton()	PTON	PTON	yes	no	yes	yes
initapi()	INITAPI	INITAPI	yes	yes	yes	yes
ioctl()	IOCTL	IOCTL	yes	yes	yes	yes
listen()	LISTEN	LISTEN	yes	yes	yes	yes
maxdesc()	no	no	yes	no	yes	yes
ntohl()	no	no	yes	yes	yes	yes
ntohs()	no	no	yes	yes	yes	yes
poll()	no	no	yes	no	no	yes
read()	READ	READ	yes	yes	yes	yes
readv()	READV	READV	yes	yes	yes	yes
recv()	RECV	RECV	yes	yes	yes	yes
recvfrom()	RECVFROM	RECVFROM	yes	yes	yes	yes
recvmsg()	no	no	yes	no	no	yes
select()	SELECT	SELECT	yes	yes	yes	yes
selectex()	SELECTEX	SELECTEX	yes	yes	yes	yes
send()	SEND	SEND	yes	yes	yes	yes
sendmsg()	no	no	yes	no	no	yes
sendto()	SENDTO	SENDTO	yes	yes	yes	yes
sethostent()	no	no	yes	no	no	yes

Introducing Socket Programming

Table 3. Supported call functions by Interface and TCP/IP Stack (continued)

Call Function	Interface			TCP/IP Stack		
	EZASMI	EZASOCKET	LE/VSE	TCP/IP for VSE/ESA	IPv6/VSE	Linux Fast Path
setibmopt()	no	no	yes	no	yes	yes
setnetent()	no	no	yes	no	no	yes
setprotoent()	no	no	yes	no	no	yes
setservent()	no	no	yes	no	no	yes
setsockopt()	SETSOCKOPT	SETSOCKOPT	yes	yes	yes	yes
shutdown()	SHUTDOWN	SHUTDOWN	yes	yes	yes	yes
socket()	SOCKET	SOCKET	yes	yes	yes	yes
socketpair()	no	no	yes	no	no	yes
takesocket()	TAKESOCKET	TAKESOCKET	yes	yes	yes	yes
	TASK	no	no	yes	yes	yes
termapi()	TERMAPI	TERMAPI	yes	yes	yes	yes
write()	WRITE	WRITE	yes	yes	yes	yes
writenv()	WRITEV	WRITEV	yes	yes	yes	yes

ERRNO Values

This section gives an overview on all ERRNO values that are returned by the TCP/IP LE/C, the EZASMI/EZASOKET socket interfaces, or both.

- Table 4 shows the ERRNO values sorted by their decimal value.
- Table 5 on page 79 shows the ERRNO values that apply only to EZASMI/EZASOKET socket interfaces sorted by their decimal value.
- Table 6 on page 79 shows the values sorted by ERRNO names.

Table 4. ERRNO Values Sorted by Value

ERRNO	ERRNO Value from LE/C or EZASMI/EZASOKET	Description
EDOM	1	Domain error.
ERANGE	2	Range error.
ELOAD	83	Load error.
EACCES	111	Permission denied.
EAGAIN	112	Resource temporarily unavailable.
EBADF	113	Bad socket descriptor.
EBUSY	114	Resource busy.
ECHILD	115	No child processes.
EDEADLK	116	Resource deadlock avoided.
EEXIST	117	File exists.
EFAULT	118	Bad address or buffer address not accessible.

Table 4. *ERRNO Values Sorted by Value (continued)*

ERRNO	ERRNO Value from LE/C or EZASMI/EZASOKET	Description
EFBIG	119	File too large.
EINTR	120	Interrupted function call.
EINVAL	121	Invalid parameter.
EIO	122	Socket closed.
EISDIR	123	Is a directory.
EMFILE	124	Too many open files.
EMLINK	125	Too many links.
ENAMETOOLONG	126	File name too long.
ENFILE	127	Too many open sockets.
ENODEV	128	No such device.
ENOENT	129	No such socket.
ENOEXEC	130	Exec format error.
ENOLCK	131	No locks available.
ENOMEM	132	Not enough memory to fulfill the request.
ENOSPC	133	No space left on device.
ENOSYS	134	Function not implemented.
ENOTDIR	135	Not a directory.
ENOTEMPTY	136	Directory not empty.
ENOTTY	137	Inappropriate I/O control operation.
ENXIO	138	No such device or address.
EPERM	139	Operation not permitted.
EPIPE	140	Broken pipe.
EROFS	141	Read-only file system.
ESPIPE	142	Invalid seek.
ESRCH	143	No such process.
EXDEV	144	A link to a file on another file system was attempted.
E2BIG	145	Argument list too long.
ELOOP	146	A loop exists in symbolic links encountered during resolution of the path argument.
EILSEQ	147	Illegal byte sequence.
ENODATA	148	No message available.
EOVERFLOW	149	Value too large to be stored in data type.
EMVSNOTUP	150	OpenEdition is not active.
EMVSDYNALC	151	Dynamic allocation error.
EMVSCVAF	152	Catalog Volume Access Facility error.
EMVSCATLG	153	Catalog obtain error.

Introducing Socket Programming

Table 4. *ERRNO Values Sorted by Value (continued)*

ERRNO	ERRNO Value from LE/C or EZASMI/EZASOKET	Description
EMVSINITIAL	156	Process initialization error.
EMVSERR	157	An internal error has occurred.
EMVSPARM	158	Bad parameters.
EMVSPFSFILE	159	Permanent file error.
EMVSBADCHAR	160	Bad character in environment variable name.
EMVSPFSPERM	162	System error.
EMVSSAFEXTRERR	163	SAF/RACF extract error.
EMVSSAF2ERR	164	SAF/RACF error.
EMVSTODNOTSET	165	System TOD clock not set.
EMVSPATHOPTS	166	Access mode argument conflicts with PATHOPTS parameter.
EMVSNORTL	167	Access to the OpenEdition version of the C RTL is denied.
EMVSEXPIRE	168	Password has expired.
EMVSPASSWORD	169	Password is invalid.
EVSE	183	Not supported under VSE.
ELENOFORK	200	Language Environment member language cannot tolerate a fork().
ELEMSGERR	201	Message file was not found in the hierarchical file system.
EIBMBADCALL	1000	A bad socket call constant in IUCV header.
EIBMBADPARM	1001	Other IUCV header error.
EIBMSOCKOUTOFRANGE	1002	Assigned socket number out of range.
EIBMSOCKINUSE	1003	Assigned socket number already in use.
EIBMIUCVERR	1004	Request failed due to IUCV error.
EOFFLOADboxERROR	1005	Offload box error.
EOFFLOADboxRESTART	1006	Offload box restarted.
EOFFLOADboxDOWN	1007	Offload box down.
EIBMCONFLICT	1008	Conflicting call outstanding on socket.
EIBMCANCELLED	1009	Request cancelled.
ENOTBLK	1100	Block device required.
ETXTBSY	1101	Text file busy.
EWOULDBLOCK	1102	Request would block. An operation on a socket marked as non blocking has encountered a situation such as no data available that otherwise would have caused the function to suspend execution.

Table 4. *ERRNO Values Sorted by Value (continued)*

ERRNO	ERRNO Value from LE/C or EZASMI/EZASOKET	Description
EINPROGRESS	1103	Socket connection in progress. O_NONBLOCK is set for the socket descriptor and the connection cannot be immediately established.
EALREADY	1104	Connection request already in progress. A connection request is already in progress for the specified socket.
ENOTSOCK	1105	Descriptor does not refer to a socket.
EDESTADDRREQ	1106	Destination address required. No bind address was specified.
EMSGSIZE	1107	Message too long.
EPROTOTYPE	1108	The socket type is not supported by the protocol.
ENOPROTOOPT	1109	No Option recognized. The option specified to setsockopt() is not supported.
EPROTONOSUPPORT	1110	The protocol is not supported by the address family, or the protocol is not supported by the implementation.
ESOCKTNOSUPPORT	1111	Socket type not supported.
EOPNOTSUPP	1112	Socket call not supported.
EPFNOSUPPORT	1113	Protocol family not supported.
EAFNOSUPPORT	1114	Address family not supported (other than AF_INET). The implementation does not support the specified address family, or the specified address is not a valid address for the address family of the specified socket.
EADDRINUSE	1115	Specified address or port is already in use.
EADDRNOTAVAIL	1116	Address not available.
ENETDOWN	1117	The local interface to use or reach the destination is down.
ENETUNREACH	1118	Network unreachable.
ENETRESET	1119	Network dropped connection on reset.
ECONNABORTED	1120	Connection aborted.
ECONNRESET	1121	Connection was forcibly closed/reset by the peer.
ENOBUFS	1122	No buffers available. Insufficient buffer resources were available in the system to perform the socket operation.
EISCONN	1123	Specified socket is already connected.
ENOTCONN	1124	Socket is not connected.
ESHUTDOWN	1125	Cannot send after socket shutdown.
ETOMANYREFS	1126	Too many references, cannot splice.

Introducing Socket Programming

Table 4. *ERRNO Values Sorted by Value (continued)*

ERRNO	ERRNO Value from LE/C or EZASMI/EZASOKET	Description
ETIMEDOUT	1127	Connection request timed out. The connection to a remote machine has timed out. If the connection timed out during execution of the function that reported this error (as opposed to timing out prior to the function being called), it is unspecified whether the function has completed some or all of the behavior associated with a successful completion of the function.
ECONNREFUSED	1128	Connection refused.
EHOSTDOWN	1129	Host is down.
EHOSTUNREACH	1130	Destination host cannot be reached.
EPROCLIM	1131	Too many processes.
EUSERS	1132	Too many users.
EDQUOT	1133	Reserved.
ESTALE	1134	The file handle has expired.
EREMOTE	1135	Too many levels of remote in path.
ENOSTR	1136	Not a stream.
ETIME	1137	Stream ioctl() timeout.
ENOSR	1138	No stream resource.
ENOMSG	1139	No message of desired type.
EBADMSG	1140	Bad message.
EIDRM	1141	Identifier removed.
ENONET	1142	Machine is not on the network.
ERREMOTE	1143	Object is remote.
ENOLINK	1144	The link has been cut.
EADV	1145	advertise error.
ESRMNT	1146	srmount error.
ECOMM	1147	Communication error on send.
EPROTO	1148	Protocol error.
EMULTIHOP	1149	Multihop is not allowed.
EDOTDOT	1150	Cross mount point (not an error).
EREMCHG	1151	Remote address changed.
ECANCELED	1152	The asynchronous I/O request has been canceled.

Table 5. EZASMI/EZASOKET only - ERRNO Values Sorted by Value

ERRNO	ERRNO Value from EZASMI/EZASOKET	Description	See Note
EZAINVFU	20000	Invalid Function used with EZASOKET call.	1
EZAINVPA	20001	Incorrect Parameter with EZASOKET call.	1
EZAERL00	20100	Error loading phase EZASOH00.	
EZAERGTV	20107	Not enough partition GETVIS.	
EZAERNIN	20108	First call not INITAPI.	
EZAERREC	20111	Recursive entry of EZA interface.	
EZAETRNA	20112	EZATRUE not active (CICS only).	
EZAERLIF	20113	LOAD of TCP/IP interface routine failed.	
EZAERBRI	20114	Bad return code from TCP/IP I/F routine.	

Note:

1. Used by EZASOKET interface only.

Programming Notes:

1. C Language definitions for ERRNOs (other than those returned by EZASMI/EZASOKET) can be found in ERRNO.H as shipped in PRD2.SCEEBASE.
2. Assembler equates for ERRNOs that may be returned from the EZASMI macro or the EZASOKET call interface can be included in your assembler program by EZASMI TYPE=TASK,STORAGE=DSECT.

Table 6. ERRNO Values sorted by Name

ERRNO	Value
EACCESS	111
EADDRINUSE	1115
EADDRNOTAVAIL	1116
EAFNOSUPPORT	1114
EAGAIN	112
EALREADY	1104
EBADF	113
EBADMSG	1140
EBUSY	114
ECANCELED	1152
ECHILD	115
ECOMM	1147
ECONNABORTED	1120
ECONNREFUSED	1128

Introducing Socket Programming

Table 6. *ERRNO Values sorted by Name (continued)*

ERRNO	Value
ECONNRESET	1121
EDEADLK	116
EDESTADDRREQ	1106
EDOM	1
EDOTEDOT	1150
EDQUOT	1133
EEXIST	117
EFAULT	118
EFBIG	119
EHOSTDOWN	1129
EHOSTUNREACH	1130
EIBMBADCALL	1000
EIBMBADPARM	1001
EIBMCANCELLED	1009
EIBMCONFLICT	1008
EIBMIUCVERR	1004
EIBMSOCKINUSE	1003
EIBMSOCKOUTOFRANGE	1002
EIDRM	1141
EILSEQ	147
EINPROGRESS	1103
EINTR	120
EINVAL	121
EIO	122
EISCONN	1123
EISDIR	123
ELEMSGERR	201
ELENOFORK	200
ELOAD	83
ELOOP	146
EMFILE	124
EMLINK	125
EMSGSIZE	1107
EMULTIHOP	1149
EMVSBADCHAR	160
EMVSCATLG	153
EMVSCVAF	152
EMVSSDYNALC	151
EMVSERR	157
EMVSEXPIRE	168

Table 6. ERRNO Values sorted by Name (continued)

ERRNO	Value
EMVSINITIAL	156
EMVSNORTL	167
EMVSNOTUP	150
EMVSPARM	158
EMVSPASSWORD	169
EMVSPATHOPTS	166
EMVSPFSFILE	159
EMVSPFSPERM	162
EMVSSAF2ERR	164
EMVSSAFEXTRERR	163
EMVSTODNOTSET	165
ENAMETOOLONG	126
ENETDOWN	1117
ENETRESET	1119
ENETUNREACH	1118
ENFILE	127
ENOBUFS	1122
ENODATA	148
ENODEV	128
ENOENT	129
ENOEXEC	130
ENOLINK	1144
ENOLCK	131
ENOMEM	132
ENOMSG	1139
ENONET	1142
ENOPROTOPT	1109
ENOSPC	133
ENOSR	1138
ENOSTR	1136
ENOSYS	134
ENOTBLK	1100
ENOTCONN	1124
ENOTDIR	135
ENOTEMPTY	136
ENOTSOCK	1105
ENOTTY	137
ENXIO	138
EOFFLOADboxDOWN	1007
EOFFLOADboxERROR	1005

Introducing Socket Programming

Table 6. *ERRNO Values sorted by Name (continued)*

ERRNO	Value
EOFFLOADboxRESTART	1006
EOPNOTSUPP	1112
EOVERFLOW	149
EPERM	139
EPFNOSUPPORT	1113
EPIPE	140
EPROCLIM	1131
EPROTO	1148
EPROTONOSUPPORT	1110
EPROTOTYPE	1108
ERANGE	2
EREMCHG	1151
EREMOTE	1135
EROFS	141
ERREMOTE	1143
ESHUTDOWN	1125
ESOCKTNOSUPPORT	1111
ESPIPE	142
ESRCH	143
ESRMNT	1146
ESTALE	1134
ETIME	1137
ETIMEDOUT	1127
ETOOMANYREFS	1126
ETXTBSY	1101
EUSERS	1132
EVSE	183
EWOULDBLOCK	1102
EXDEV	144
E2BIG	145
EZAERBRI	20114
EZAERGTV	20107
EZAERL00	20100
EZAERLIF	20113
EZAERNIN	20108
EZAERREC	20111
EZAETRNA	20112
EZAINVFU	20000
EZAINVPA	20001

CICS Considerations

The C Socket programming interface supports writing applications for either a CICS or batch execution environment. This is also true for the EZASMI macro and the EZASOKET call interface.

However, while the Assembler SOCKET macro and the TCP/IP for VSE/ESA HLL preprocessor (resolving EXEC TCP calls) allow to explicitly specify the execution environments, this is not possible with the BSD-C socket interfaces.

A programmer can write bimodal modules or applications, being called from either a CICS or batch program. The TCP/IP runtime services will act according to the execution environment's requirements, i.e. they will eventually use CICS services (for example EXEC CICS WAIT) where appropriate.

To force an application to dynamically determine the environment it is running in, you need to include the following 2 OBJ files in the application's link-edit step:

- IPCICSRQ (TCP/IP for VSE/ESA only)
- DFHECI

Omitting those two files will cause the application to act CICS unfriendly even if running under CICS' control, for example by issuing VSE GETVIS requests instead of CICS GETMAIN.

Note: This is true for non-LE socket applications using the BSD-C interface of TCP/IP for VSE/ESA. Using the C socket interfaces provided by the VSE Language Environment 1.4 C runtime does not require these modules to be linked for the purpose described above. This is already covered by the TCP/IP for VSE/ESA support for the LE/VSE 1.4 C socket interfaces, transparently to the application. This support is described in Chapter 10, "TCP/IP Support for the LE/VSE C Socket Interface," on page 85.

CICS Considerations for the EZA Interfaces

Before the EZA API (EZASMI macro and EZASOKET call interface) can be used in a CICS TS Transaction Environment, its "task-related-user-exit" (TRUE) routine has to be started. This EZA "task-related-user-exit" is named EZATRUE. It is responsible for allocating task-related working storage to the EZA API processing environment and for possible cleanup processing during CICS end-of-task processing.

The "task-related-user-exit" EZATRUE is started/stopped with program EZASTRUE by one of the following means:

- transaction EZAT (EZAT START starts EZATRUE, EZAT STOP stops it)
- entry of EZASTRUE to the PLTPI (for auto-startup during CICS startup) and to the PLTSD (for auto-shutdown during CICS shutdown)
- an EXEC CICS LINK to program EZASTRUE with the following COMMAREA parameter list:

Table 7. COMMAREA parameter list

Offset	Length	Description
0	8	Eyecatcher "EZATRUE"

Table 7. *COMMAREA* parameter list (continued)

Offset	Length	Description	
8	1	Request Type:	
		"S"	Start Request
		"T"	Termination Request
9	1	Return Code from EZASTRUE:	
		0	EZATRUE start/termination successful
		4	EZATRUE already in requested state
		8	Failure to start/terminate EZATRUE
		16	Invalid parameter list

Executing TCP/IP Application Programs

Connecting to TCP/IP

By default, your TCP/IP application will attempt to connect with the TCP/IP partition that has been assigned ID=00. Refer to the corresponding TCP/IP stack documentation on how to assign the ID to your TCP/IP stack. The default ID value is "00". If you want to connect to a TCP/IP partition with an ID not equal to "00", you can do so by including an appropriate OPTION statement in your JCL:

```
// OPTION SYSPARM='xx'
```

In the above, xx is the two-digit ID number, coded exactly as in the TCP/IP startup JCL or parameters.

Preparation and Setup for SSL

Before using the LE/VSE C, EZASMI and EZASOKET function calls for secured socket communication, the VSE system must be prepared to use SSL for VSE.

Note: SSL for VSE can only be used together with TCP/IP for VSE/ESA.

This preparation work includes

- (Optional) Creation of the library and sublibrary where private key and certificates are to be stored (if default files on disk are not to be used).
- (Optional) Definition of library, sublibrary and member name to be used for private key and certificates (if default files on disk are not to be used)
- Creation of private key.
- Creation of server certificate.
- Creation of root certificate.
- (Optional) Verification of SSL for VSE Certificate.

Refer to *z/VSE e-business Connectors User's Guide* for default SSL setup and to *TCP/IP for VSE 1.5 Optional Features* for a detailed description of this preparation work.

Chapter 10. TCP/IP Support for the LE/VSE C Socket Interface

Overview

BSD or "Berkeley" Sockets is a method for using TCP/IP programming interfaces that was developed for UNIX platforms. Only a subset of the routines you might know from other, especially UNIX alike platforms is implemented. The BSD-C alike interfaces provided by TCP/IP for VSE/ESA are primarily aimed for users of non-LE enabled C compilers, for example, the IBM C/370 compiler. This interface is described in the *TCP/IP for VSE 1.5 Programmer's Reference*.

If you use the IBM C for VSE/ESA Release 1 (5686-A01) compiler together with the IBM Language Environment for z/VSE (LE/VSE) 1.4 C runtime environment we strongly recommend the usage of the LE/VSE 1.4 socket interfaces. These are compatible with the OS/390 X/Open (XPG4.2) compliant socket interfaces. This assures the maximum on compatibility and portability for cross platform development.

Note:

1. The LE/VSE 1.4 runtime environment does not implement the socket routines itself, but dynamically calls phase \$EDCTCPV which is part of the TCP/IP for VSE/ESA product stored in PRD2.TCPIP. Therefore, the socket application is decoupled from the TCP/IP product (see Figure 12 on page 52 for details). The LE/VSE 1.4 runtime dynamically picks up new service levels, by calling this phase, while applications using the native TCP/IP BSD- C socket routines eventually need to be relinked when TCP/IP service is applied.
2. LE/VSE 1.4 C base ships a default \$EDCTCPV phase in PRD2.SCEEBASE aimed for systems where TCP/IP for VSE/ESA is either not installed or deleted. This default phase does nothing but defining a function-specific return code and issuing message EDCV001I, stating that the called function is not implemented.

If you receive this message, check your application's LIBDEF for correctness and check whether the routine is supposed to be available.

3. While the LE/VSE 1.4 C runtime provides the same range of socket routines as OS/390, TCP/IP for VSE/ESA has only implemented a subset. This means that when you use a LE/VSE C runtime interface, you need this for reference and implementation details.

TCP/IP Callable Functions — Function Descriptions

accept() — Accept a New Connection on a Socket

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <socket.h>
```

```
int accept(int socket, struct sockaddr *address, size_t *address_len);
```

General Description

The `accept()` call is used by a server to accept a connection request from a client. For details, refer to the functional description of your TCP/IP provider. When a

accept

connection is available, the socket created is ready for use to read data from the process that requested the connection. The call accepts the first connection on its queue of pending connections for the given socket *socket*. The `accept()` call creates a new socket descriptor with the same properties as *socket* and returns it to the caller. The original socket, *socket*, remains available to accept more connection requests.

Parameter

Description

socket The socket descriptor.

address The socket address of the connecting client that is filled in by `accept()` before it returns. The format of *address* is determined by the domain that the client resides in. This parameter can be NULL if the caller is not interested in the client address.

address_len

Must initially point to an integer that contains the size in bytes of the storage pointed to by *address*. On return that integer contains the size of the data returned in the storage pointed to by *address*. If *address* is NULL, *address_len* is ignored.

The *socket* parameter is a stream socket descriptor created with the `socket()` call. It is usually bound to an address with 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 places an upper boundary on the size of the queue.

The *address* parameter is a pointer to a buffer into which the connection requester's address is placed. The *address* parameter is optional and can be set to be the NULL pointer. If set to NULL, the requester's address is not copied into the buffer. The exact format of *address* depends on the addressing domain from which the communication request originated.

For example, if the connection request originated in the AF_INET domain, *address* points to a `sockaddr_in` structure, or if the connection request originated in the AF_INET6 domain, *address* points to a `sockaddr_in6` structure. The `sockaddr_in` and `sockaddr_in6` structures are defined in `in.h`. The *address_len* parameter is used only if *name* is not NULL. Before calling `accept()`, you must set the integer pointed to by *address_len* to the size of the buffer, in bytes, pointed to by *address*. On successful return, the integer pointed to by *address_len* contains the actual number of bytes copied into the buffer. If the buffer is not large enough to hold the address, up to *address_len* bytes of the requester's address are copied. If the actual length of the address is greater than the length of the supplied `sockaddr`, the stored address is truncated. The `sa_len` member of the store structure contains the length of the untruncated address.

Note: This call is used only with SOCK_STREAM sockets. There is no way to screen requesters without calling `accept()`. The application cannot tell the system the requesters from which it will accept connections. 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.

Returned Value

A nonnegative socket descriptor indicates success; the value -1 indicates an error. The value of the error code indicates the specific error.

Error Code

Description

EBADF

The *socket* parameter is not within the acceptable range for a socket descriptor.

EFAULT

Using *address* and *address_len* would result in an attempt to copy the address into a portion of the caller's address space into which information cannot be written.

EINVAL

`listen()` was not called for socket descriptor *socket*.

ENFILE

The maximum number of socket descriptors in the system are already open.

ENOBUFS

Insufficient buffer space is available to create the new socket.

EOPNOTSUPP

The socket type of the specified socket does not support accepting connections.

EWouldBlock

The socket descriptor *socket* is in nonblocking mode, and no connections are in the queue.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1, and `errno` is set to `EOPNOTSUPP`. In this case the message `EDCV001I` or `EDCT002I` is issued.

Example

The 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 wish to have the requester's address returned.

```
int clientsocket;
int s;
struct sockaddr clientaddress;
int address_len;
int accept(int s, struct sockaddr *addr, int *address_len);
/* socket(), bind(), and listen() have been called */

/* EXAMPLE 1: I want the address now */
address_len = sizeof(clientaddress);
clientsocket = accept(s, &clientaddress, &address_len);

/* EXAMPLE 2: I can get the address later using getpeername() */
clientsocket = accept(s, (struct sockaddr *) 0,
(int *) 0);
```

aio_cancel() — Cancel an Asynchronous I/O Request

Format

```
#define _OPEN_SYS SOCK_EXT
#include <aio.h>

int aio_cancel(int socket, struct aiocb *aiocbp);
```

General Description

The `aio_cancel()` function attempts to cancel one or more asynchronous I/O requests currently outstanding against socket descriptor `socket`. The `aiocbp` argument points to an `aiocb` structure for a particular request to be canceled, or is NULL to cancel all outstanding cancelable requests against `socket`.

Normal asynchronous notification occurs for asynchronous I/O operations that are successfully canceled. The associated error status is set to `ECANCELED` and the return status is set to `-1` for the canceled requests.

For requests that cannot be canceled, the normal asynchronous completion process takes place when their I/O completes. In this case the `aiocb` is not modified by `aio_cancel()`.

An asynchronous operation is cancelable if it is currently blocked or becomes blocked. Once an outstanding request can be completed, it is allowed to complete. For example, an `aio_read()` is cancelable if there is no data available when `aio_cancel()` is called.

`socket` must be a valid socket descriptor, but when `aiocbp` is not NULL, `socket` does not have to match the socket descriptor with which the asynchronous operation was initiated. However, for maximum portability it should match.

The `aio_cancel()` function always waits for the request being canceled to either complete or be canceled. When control returns from `aio_cancel()`, the program may safely free the original request's `aiocb` and buffer.

Canceling all requests on a given descriptor does not stop new requests from being made or otherwise effect the descriptor. The program may start again or close the descriptor depending on why it issued the cancel.

An individual request can only be canceled once. Subsequent attempts to explicitly cancel the same request will fail with `EALREADY`.

Returned Value

The `aio_cancel()` function returns one of the following values:

Return Value

Description

AIO_CANCELED

The requested operations were canceled.

AIO_NOTCANCELED

At least one of the requested operations cannot be canceled because it is in progress. In this case, the state of the other operations, if any, referenced in

the call to `aio_cancel()` is not indicated by the return value of `aio_cancel()`. The application can determine the status of these operations by using `aio_error()`.

AIO_ALLDONE

The operations have already completed. This is returned when there are no outstanding requests found that match the criteria specified. This is also the result returned when a file associated with *socket* does not support the asynchronous I/O function because there are no outstanding requests to be found that match the criteria specified.

-1 An error has occurred. `errno` is set to indicate the type of error.

The `aio_cancel()` function will fail if:

errno Description

EBADF

The *socket* argument is not a valid socket descriptor.

EALREADY

The operation to be canceled is already being canceled.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and `errno` is set to EBADF. In this case the message EDCV001I or EDCT002I is issued.

aio_error() — Retrieve Error Status for an Asynchronous I/O Operation

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <aio.h>
```

```
int aio_error(const struct aiocb *aiocbp);
```

General Description

The `aio_error()` function returns the error status associated with the *aiocb* structure referenced by the *aiocbp* argument. The error status for an asynchronous I/O operation is the `errno` value that would be set by the corresponding `read()`, or `write()` operation. If the operation has not yet completed, the error status is equal to EINPROGRESS.

Returned Value

If the asynchronous I/O operation has completed successfully, then 0 is returned. If the asynchronous I/O operation has completed unsuccessfully, then the error status as described for `read()`, or `write()` is returned. If the asynchronous I/O operation has not yet completed, then EINPROGRESS is returned.

The `aio_error()` function does not set `errno`.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1. In this case the message EDCV001I or EDCT002I is issued.

aio_read() — Asynchronous Read from a Socket

Format

```
#define _OPEN_SYS SOCK_EXT
#include <aio.h>

int aio_read(struct aiocb *aiocbp);
```

General Description

The `aio_read()` function initiates an asynchronous read operation as described by the `aiocb` structure (the asynchronous I/O control block).

The `aiocbp` argument points to the `aiocb` structure. This structure contains the following members:

`aio_filedes`
socket descriptor

`aio_offset`
file offset

`aio_buf` location of buffer

`aio_nbytes`
length of transfer

`aio_reqprio`
request priority offset

`aio_sigevent`
signal number and value

`aio_lio_opcode`
operation to be performed

The operation reads up to `aio_nbytes` from the socket associated with `aio_filedes` into the buffer pointed to by `aio_buf`. The call to `aio_read()` returns when the request has been initiated or queued (even if the data cannot be delivered immediately).

Asynchronous I/O is currently only supported for sockets. The `aio_offset` field can be set but is ignored.

With a stream socket an asynchronous read may be completed when the first packet of data arrives and the application may have to issue additional reads, either asynchronously or synchronously, to get all the data it wants. A datagram socket has message boundaries and the operation will not complete until an entire message has arrived.

The `aiocbp` value may be used as an argument to `aio_error()` and `aio_return()` functions in order to determine the error status and return status, respectively, of the asynchronous operation. While the operation is proceeding, the error status retrieved by `aio_error()` is `EINPROGRESS`; the return status retrieved by `aio_return()` however is unpredictable.

If an error condition is encountered during the queuing, the function call returns without having initiated or queued the request.

The program can occasionally poll the *aio*cb with `aio_error()` until the result is no longer `EINPROGRESS`.

Be aware that the operation might complete, before control returns from the call to `aio_read()`. Even if the operation does complete this quickly, the return value from the call to `aio_read()` is zero, reflecting the queueing of the I/O request not the results of the I/O itself.

An asynchronous operation may be canceled with `aio_cancel()` prior to its completion. Canceled operations complete with an error status of `ECANCELED`. Due to timing, the operation may still complete naturally, either successfully or unsuccessfully, before it can be canceled by `aio_cancel()`.

If the socket descriptor of this operation is closed, the operation will be deleted if it has not completed or is not just about to complete. `Close()` will wait for asynchronous operations in progress for the descriptor to be deleted or completed.

You can use `aio_suspend()` to wait for the completion of asynchronous operations.

Sockets must be in blocking state or the operation may fail with `EWOULDBLOCK`.

If the control block pointed by *aio*cbp or the buffer pointed to by *aio*buf becomes an illegal address prior to the asynchronous I/O completion, then the behavior of `aio_read()` is unpredictable.

Simultaneous asynchronous operations using the same *aio*cbp, asynchronous operations using an invalid *aio*cbp, or any system action that changes the process memory space while asynchronous I/O is outstanding to that address range, will produce unpredictable results.

The *aio*lio_opcode field is set to `LI0_READ` by the function `aio_read()`.

`_POSIX-PRIORITIZED_IO` is not supported. The *aio*reqprio field can be set but is ignored.

`_POSIX_SYNCHRONIZED_IO` is not supported.

Returned Value

The `aio_read()` function returns the value of zero to the calling process if the I/O operation is successfully queued; otherwise, the function returns the value -1 and sets `errno` to indicate the error. The `aio_read()` function will fail if:

errno Description

ENOSYS

The file associated with *aio*filedes does not support the `aio_read()` function.

Each of the following conditions might be detected synchronously at the time of the call to `aio_read()`, or asynchronously. If any of the conditions below are detected synchronously, the `aio_read()` function returns -1 and sets the `errno` to the corresponding value. If any of the conditions below are detected asynchronously, the return status of the asynchronous operation is set to -1, and the error status of the asynchronous operation is set to the corresponding value.

Error Status
Description

EBADF

The *aio_filedes* argument is not a valid socket descriptor open for reading.

EWOULDBLOCK

The file associated with *aio_filedes* is in non-blocking state and there is no data available.

EINVAL

aio_sigevent contains an invalid value.

If the `aio_read()` function successfully queues the I/O operation but the operation is subsequently canceled or encounters an error, the return status of the asynchronous operations is set to -1, and the error status of the asynchronous operation is set to the error status normally set by the `read()` function call, or to the following value:

Error Status
Description

ECANCELED

The requested I/O was canceled before the I/O completed due to an explicit call to `aio_cancel()`.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and `errno` is set to `ENOSYS`. In this case the message `EDCV001I` or `EDCT002I` is issued.

aio_return() — Retrieve Status for an Asynchronous I/O Operation

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <aio.h>

int aio_return(const struct aiocb *aiocbp);
```

General Description

The `aio_return()` function returns the return status associated with the *aiocb* structure referenced by the *aiocbp* argument. The return status for an asynchronous I/O operation is the value that would be set by the corresponding `read()` or `write()` operation. While the operation is proceeding, the error status retrieved by `aio_error()` is `EINPROGRESS`; the return status retrieved by `aio_return()` however is unpredictable. The `aio_return()` function may be called to retrieve the return status of a given asynchronous operation; once `aio_error()` has returned with 0.

Returned Value

If the asynchronous I/O operation has completed successfully, then the return status as described for `read()` or `write()` is returned. If the asynchronous I/O operation has not yet completed, then the return status is unpredictable.

The `aio_return()` does not set `errno`.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1. In this case the message EDCV001I or EDCT002I is issued.

aio_suspend() — Wait for an Asynchronous I/O Request

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <aio.h>

int aio_suspend(const struct aiocb * const list[ ],
               int nent, const struct timespec * timeout);
```

General Description

The `aio_suspend()` function suspends the calling thread when the `timeout` is a null pointer until at least one of the asynchronous I/O operations referenced by the `list` argument has completed. Or, if `timeout` is not null, it is suspended until the time interval specified by `timeout` has passed. If the time interval indicated in the `timespec` structure pointed to by `timeout` passes before any of the I/O operations referenced by `list`, then `aio_suspend()` returns with an error. If any of the `aiocb` structures in the list correspond to completed asynchronous I/O operations (that is, the error status for the operation is not equal to `EINPROGRESS`) at the time of the call, the function returns without suspending the calling thread.

The `list` argument is an array of pointers to asynchronous I/O control blocks (AIOCBs). The `nent` argument indicates the number of elements in the array. Each `aiocb` structure pointed to will have been used in initiating an asynchronous I/O request. This array may contain null pointers, which are ignored. If this array contains pointers that refer to `aiocb` structures that have not been used in submitting asynchronous I/O or `aiocb` structures that are not valid, the results are unpredictable.

Returned Value

If the `aio_suspend()` function returns after one or more asynchronous I/O operation have completed, the function returns zero. Otherwise, the function returns a value of -1 and sets `errno` to indicate the error. The application may determine which asynchronous I/O completed by scanning the associated error and return status using `aio_error()` or `aio_return()`, respectively. The value of `errno` indicates the specific error.

errno Description

ENOSYS

z/VSE does not support the `aio_suspend` function.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and `errno` is set to `ENOSYS`. In this case the message EDCV001I or EDCT002I is issued.

Usage Notes

1. The AIOCBs represented by the list of AIOCB pointers must reside in the same storage key as the key of the invoker of `aio_suspend`. If the AIOCB Pointer List or any of the AIOCBs represented in the list are not accessible by the invoker an EFAULT may occur.

2. AIOCB pointers in the list with a value of zero are ignored.
3. A timeout value of zero (seconds+nanoseconds) means that the `aio_suspend()` call will not wait at all. It will check for any completed asynchronous I/O requests. If none are found it will return with a `EAGAIN`. If at least one is found `aio_suspend()` will return with success.
4. A timeout value of a *timespec* with the `tv_sec` field set with `INT_MAX`, as defined in `<limits.h>` will cause the `aio_suspend` service to wait until a asynchronous I/O request completes.
5. The AIOCBs passed to `aio_suspend()` must not be freed or reused while this service is still in progress. This service may use the AIOCBs even after the asynchronous I/O completes. Modifying the AIOCB during an `aio_suspend()` will produce unpredictable results.

aio_write() — Asynchronous Write to a Socket

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <aio.h>

int aio_write(struct aiocb *aiocbp);
```

General Description

The `aio_write()` function initiates an asynchronous write operation as described by the *aiocb* structure (the asynchronous I/O control block).

The *aiocbp* argument points to the *aiocb* structure. This structure contains the following members:

aio_filedes
socket descriptor

aio_offset
file offset

aio_buf location of buffer

aio_nbytes
length of transfer

aio_repprio
request priority offset

aio_sigevent
signal number and value

aio_lio_opcode
operation to be performed

The operation will write *aio_nbytes* from the buffer pointed to by *aio_buf* to the socket associated with *aio_filedes*. The call to `aio_write()` returns when the request has been initiated or queued (even if the data cannot be delivered immediately).

Asynchronous I/O is currently only supported for sockets. The *aio_offset* field may be set but is ignored.

The *aiocbp* value may be used as an argument to `aio_error()` and `aio_return()` functions in order to determine the error status and return status, respectively, of

the asynchronous operation. While the operation is proceeding, the error status retrieved by `aio_error()` is `EINPROGRESS`; the return status retrieved by `aio_return()` however is unpredictable.

If an error condition is encountered during the queueing, the function call returns without having initiated or queued the request.

The program can occasionally poll the *aio*cb with `aio_error()` until the result is no longer `EINPROGRESS`.

Be aware that the operation might complete before control returns from the call to `aio_read()`. Even if the operation does complete this quickly, the return value from the call to `aio_read()` is zero, reflecting the queueing of the I/O request not the results of the I/O itself.

An asynchronous operation can be canceled with `aio_cancel()` prior to its completion. Canceled operations complete with an error status of `ECANCELED`. Due to timing, the operation might still complete naturally, either successfully or unsuccessfully, before it can be canceled by `aio_cancel()`.

If the socket descriptor of this operation is closed, the operation is deleted if it has not completed or is not just about to complete. `Close()` will wait for asynchronous operations in progress for the descriptor to be deleted or completed.

You can use `aio_suspend()` to wait for the completion of asynchronous operations. Sockets must be in blocking state or the operation may fail with `EWOULDBLOCK`.

If the control block pointed by *aio*cbp or the buffer pointed to by *aio*_buf becomes an illegal address prior to the asynchronous I/O completion, then the behavior of `aio_read()` is unpredictable.

Simultaneous asynchronous operations using the same *aio*cbp, attempting asynchronous operations using an invalid *aio*cbp, or any system action that changes the process memory space while asynchronous I/O is outstanding to that address range, will produce unpredictable results.

The *aio*_lio_opcode field must be set to `LIO_WRITE`.

`_POSIX-PRIORITIZED_IO` is not supported. The *aio*_reqprio field may be set but is ignored.

`_POSIX_SYNCHRONIZED_IO` is not supported.

Returned Value

The `aio_write()` function returns the value of zero to the calling process if the I/O operation is successfully queued; otherwise, the function returns the value -1 and sets `errno` to indicate the error. The `aio_write()` function will fail if:

errno Description

ENOSYS

The file associated with *aio*_files does not support the `aio_write()` function.

Each of the following conditions may be detected synchronously at the time of the call to `aio_write()`, or asynchronously. If any of the conditions below are detected

aio_write

synchronously, the `aio_write()` function returns -1 and sets the `errno` to the corresponding value. If any of the conditions below are detected asynchronously, the return status of the asynchronous operation is set to -1, and the error status of the asynchronous operation is set to the corresponding value.

Error Status / <code>errno</code>	Description
-----------------------------------	-------------

EBADF	The <code>aio_filedes</code> argument is not a valid socket descriptor open for writing.
--------------	--

EWouldBlock	The file associated with <code>aio_filedes</code> is in non-blocking state and there is no data available.
--------------------	--

EINval	The <code>aio_nbytes</code> is not a valid value or <code>aio_sigevent</code> contains an invalid value.
---------------	--

If the `aio_write()` function successfully queues the I/O operation, but the operation is subsequently canceled or encounters an error, the return status of the asynchronous operations is set to -1, and the error status of the asynchronous operation is set to the error status normally set by the `write()` function call, or to the following value:

Error Status	Description
--------------	-------------

ECanceled	The requested I/O was canceled before the I/O completed due to an explicit call to <code>aio_cancel()</code> .
------------------	--

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and `errno` is set to `ENOSYS`. In this case the message `EDCV001I` or `EDCT002I` is issued.

bind() — Bind a Name to a Socket

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <socket.h>
```

```
int bind(int socket, const struct sockaddr *name, size_t namelen);
```

General Description

The `bind()` call binds a unique local name to the socket with descriptor `socket`. After calling `socket()`, a descriptor does not have a name associated with it. However, it does belong to a particular address family as specified when `socket()` is called. The exact format of a name depends on the address family.

Parameter

Description

<code>socket</code>	The socket descriptor returned by a previous <code>socket()</code> call.
---------------------	--

<code>name</code>	The pointer to a <code>sockaddr</code> structure containing the name that is to be bound to <code>socket</code> .
-------------------	---

<code>namelen</code>	The size of <code>name</code> in bytes.
----------------------	---

The *socket* parameter is a socket descriptor of any type created by calling `socket()`.

The *name* parameter is a pointer to a buffer containing the name to be bound to *socket*. The *namelen* parameter is the size, in bytes, of the buffer pointed to by *name*.

Socket Descriptor Created in the AF_INET Domain

If the socket descriptor *socket* was created in the AF_INET domain, the format of the name buffer is expected to be `sockaddr_in`, as defined in the include file `in.h`. The structure is defined as follows:

```
struct in_addr
{
    ip_addr_t s_addr;
};

struct sockaddr_in {
    unsigned char  sin_len;
    unsigned char  sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    unsigned char  sin_zero[8];
};
```

The *sin_family* field must be set to AF_INET.

The *sin_port* field is set to 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 leaves it to the system to assign an available port. The application can call `getsockname()` to discover the port number assigned.

The *sin_addr.s_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 is to bind. Subsequently, only UDP packets and TCP connection requests from this interface (which match 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 (which match 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 for its port, regardless of the network interface on which the requests arrived.

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

Socket Descriptor Created in the AF_INET6 Domain

If the socket descriptor *socket* was created in the AF_INET6 domain, the format of the name buffer is expected to be `sockaddr_in6`, as defined in the include file `in.h`. The structure is defined as follows:

```
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;
};
```

bind

The *sin6_len* field is set to the size of this structure. The `SIN6_LEN` macro is defined to indicate the version of the `sockaddr_in6` structure being used.

The *sin6_family* field identifies this as a `sockaddr_in6` structure. This field overlays the *sa_family* field, if the buffer is cast to a `sockaddr` structure. The value of this field must be `AF_INET6`.

The *sin6_port* field contains the 16-bit UDP or TCP port number. This field is used in the same way as the *sin_port* field of the `sockaddr_in` structure. The port number is stored in network byte order.

The *sin6_flowinfo* field is a 32-bit field that contains the traffic class and the flow label.

The *sin6_addr* field is a single `in6_addr` structure. This field holds one 128-bit IPv6 address. The address is stored in network byte order.

The *sin6_scope_id* field is a 32-bit integer that identifies a set of interfaces as appropriate for the scope of the address carried in the *sin6_addr* field. For a link scope *sin6_addr*, *sin6_scope_id*, this would be an interface index. For a site scope *sin6_addr*, *sin6_scope_id*, this would be a site identifier.

Returned Value

The value 0 indicates success; the value -1 indicates an error. The value of the error code indicates the specific error.

Error Code

Description

EADDRINUSE

The address is already in use.

EAFNOSUPPORT

The address family is not supported (it is not `AF_INET` or `AF_INET6`).

EBADF

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

EINVAL

The socket is already bound to an address—for example, trying to bind a name to a socket that is already connected. Or the socket was shut down.

ENOBUFS

`bind()` is unable to obtain a buffer due to insufficient storage.

EOPNOTSUPP

The socket type of the specified socket does not support binding to an address.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1, and `errno` is set to `EOPNOTSUPP`. In this case the message `EDCV001I` or `EDCT002I` is issued.

Example

The following example illustrates the `bind()` call binding to interfaces in the `AF_INET` domain. 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. It is a good idea to zero the structure before using it to ensure that the name requested does not set any reserved fields.

```
int rc;
int s;
struct sockaddr_in myname;

/* 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.s_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));
aslr.* 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.s_addr = inet_addr("129.5.24.1");
/* specific interface */
myname.sin_port = 0;
:
rc = bind(s, (struct sockaddr *) &myname,
sizeof(myname));
```

close() — Close a Socket

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
```

```
int close(int socket);
```

General Description

`close()` call shuts down the socket associated with the socket descriptor *socket*, and frees resources allocated to the socket. If *socket* 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.

Parameter

Description

socket The descriptor of the socket to be closed.

Note: All sockets should be closed before the end of your process.

close

For AF_INET and AF_INET6 stream sockets (SOCK_STREAM) using SO_LINGER socket option, the socket does not immediately end if data is still present when a close is issued. The following structure is used to set or unset this option, it is defined in **socket.h**. It is to be used with the *setsockopt* routine.

```
struct linger {
    int l_onoff;      /* zero=off, nonzero=on */
    int l_linger;    /* time is seconds to linger */
};
```

If the *l_onoff* switch is nonzero, the system attempts to deliver any unsent messages. If a linger time is specified, the system waits for *n* seconds before flushing the data and terminating the socket.

Returned Value

If successful, *close()* returns 0. If unsuccessful, it returns -1 and sets *errno* to one of the following:

EBADF

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

EIO An I/O error occurred while reading from or writing to the socket.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1, and *errno* is set to EBADF. In this case the message EDCV001I or EDCT002I is issued.

connect() — Connect a Socket

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <socket.h>
```

```
int connect(int socket, const struct sockaddr *name, size_t namelen);
```

General Description

For stream sockets, the *connect()* call attempts to establish a connection between two sockets. For datagram sockets, the *connect()* call specifies the peer for a socket. The *socket* 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 make a connection to another socket.

Parameter

Description

socket The socket descriptor.

name The pointer to a socket address structure containing the address of the socket to which a connection is attempted.

namelen

The size of the socket address pointed to by *name* in bytes.

The *connect()* call on a stream socket is used by the client application to establish a connection to a server. The server must have a passive open pending. A server

that is using sockets must successfully call `bind()` and `listen()` before a connection can be accepted by the server with `accept()`.

If *socket* 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, `connect()` returns -1 with the error code set to `EINPROGRESS` to indicate that the connection has been initiated but is not yet complete (if no errors occurred). The caller can test the completion of the connection setup by calling `select()` and testing for the ability to write to the socket.

When called for a datagram 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()`, `writenv()`, `send()`, and `recv()` calls are then available in addition to `sendto()` and `recvfrom()` 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 the null address (all fields zeroed).

The *name* parameter is a pointer 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*.

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 include file `in.h`.

```
struct in_addr
{
    ip_addr_t s_addr;
};

struct sockaddr_in {
    unsigned char  sin_len;
    unsigned char  sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    unsigned char  sin_zero[8];
};
```

The *sin_family* field must be set to `AF_INET`. The *sin_port* field is set to the port to which the server is bound. It must be specified in network byte order. The *sin_zero* field is not used and must be set to all zeros.

Servers in the AF_INET6 domain

If the server is in the `AF_INET6` domain, the format of the name buffer is expected to be `sockaddr_in6`, as defined in the include file `in6.h`.

```
:
struct sockaddr_in6 {
    uint8_t char 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` must be set to `AF_INET6`.

Returned Value

The value 0 indicates success; the value -1 indicates an error. The value of the error code indicates the specific error.

Error Code

Description

EAFNOSUPPORT

The address family is not supported.

EALREADY

The socket descriptor *socket* is marked nonblocking, and a previous connection attempt has not completed.

EBADF

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

EFAULT

Using *name* and *namelen* would result in an attempt to copy the address into a portion of the caller's address space to which data cannot be written.

EINPROGRESS

`O_NONBLOCK` is set for the socket descriptor for the socket, and the connection cannot be established immediately. The connection is established asynchronously. The `EINPROGRESS` value does not indicate an error condition.

EINVAL

The *namelen* parameter is not a valid length.

EISCONN

The socket descriptor *socket* is already connected.

EOPNOTSUPP

The *socket* parameter is not of type `SOCK_STREAM`.

ETIMEDOUT

The connection establishment timed out before a connection was made.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1, and `errno` is set to `EOPNOTSUPP`. In this case the message `EDCV001I` or `EDCT002I` is issued.

Example

The following are examples of the `connect()` 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 zero the structure 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() — Bind a Name to a Socket" on page 96.


```

#define _XOPEN_SOURCE_EXTENDED 1
#include <socket.h>
#include <in.h>

int s;
struct sockaddr_in inet_server;
int rc;

/* Connect to server bound to a specific interface in the
Internet domain */
/* make sure the sin_zero field is cleared */
memset(&inet_server, 0, sizeof(inet_server));
inet_server.sin_family = AF_INET;
inet_server.sin_addr = inet_addr("129.5.24.1");
/* specific interface */
inet_server.sin_port = htons(1024);
:
:
rc = connect(s, (struct sockaddr *) &inet_server, sizeof(inet_server));

```

endhostent() — Work with a Host Entry

Format

```

#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

void endhostent(void);

```

General Description

The `endhostent()` call closes the data set which contains information about known hosts.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the message EDCV001I or EDCT002I is issued.

endnetent() — Close Network Information Data Sets

Format

```

#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

void endnetent(void);

```

General Description

The `endnetent()` call closes the data set, which contains information about known networks.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the message EDCV001I or EDCT002I is issued.

endprotoent() — Work with a Protocol Entry

Format

```

#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

void endprotoent(void);

```

General Description

The `endprotoent()` call closes the data set which contains information about the networking protocols.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the message EDCV001I or EDCT002I is issued.

endservent() — Close Network Services Information Data Sets

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

void endservent(void);
```

General Description

The `endservent()` call closes the data set which contains information about network services. Example services are name server, File Transfer Protocol (FTP), and telnet.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the message EDCV001I or EDCT002I is issued.

fcntl() — Control Open Socket Descriptors

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <types.h>
#include <unistd.h>
#include <fcntl.h>

int fcntl(int socket, int cmd, ... /* arg */);
```

General Description

The operating characteristics of sockets can be controlled with the `fcntl()` call. The operations to be controlled are determined by *cmd*. The *arg* parameter is a variable with a meaning that depends on the value of the *cmd* parameter.

Parameter

Description

socket The socket descriptor.

cmd The command to perform.

arg The data associated with *cmd*.

The *cmd* argument can be one of the following symbols:

F_GETFL

This command gets the status flags of socket descriptor *socket*. With the `_XOPEN_SOURCE_EXTENDED 1` feature test macro you can query the `O_NDELAY` flag. The `O_NDELAY` flag marks *socket* as being in nonblocking mode. If data is not present on calls that can block, such as `read()`, `readv()`, and `recv()`, the call returns with -1, and the error code is set to `EWOULDBLOCK`.

F_SETFL

This command sets the status flags of socket descriptor *socket*. With the `_XOPEN_SOURCE_EXTENDED 1` feature test macro you can set the `O_NDELAY` flag.

Returned Value

If successful, the value returned will depend on the *cmd* that was specified. If unsuccessful, `fcntl()` returns -1 and sets `errno` to one of the following:

Error Code**Description****EBADF**

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

EINVAL

The *arg* parameter is no a valid flag, or the *cmd* parameter is not a valid command.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1, and `errno` is set to `EINVAL`. In this case the message `EDCV001I` or `EDCT002I` is issued.

Example

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <types.h>
#include <unistd.h>
#include <fcntl.h>
int s;
int rc;
int flags;
:
/* Place the socket into nonblocking mode */
rc = fcntl(s, F_SETFL, O_NDELAY);

/* See if asynchronous notification is set */
flags = fcntl(s, F_GETFL, 0);
if (flags & O_NDELAY)
    /* it is set */
else
    /* it is not */
```

freeaddrinfo() — Free addrinfo storage**Format**

```
#define _OPEN_SYS_SOCKET_IPV6
#include <socket.h>
#include <netdb.h>
```

```
void *freeaddrinfo(struct addrinfo *ai);
```

General Description

The `freeaddrinfo()` function frees one or more `addrinfo` structures returned by `getaddrinfo()`, along with any additional storage associated with those structures. If the `ai_next` field of the structure is not null, the entire list of structures is freed.

Returned Value

No return value is defined.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the message EDCV001I or EDCT002I is issued.

gai_strerror() — Address and name information error description**Format**

```
#define _OPEN_SYS_SOCKET_IPV6
#include <netdb.h>
```

```
char *gai_strerror(int ecode);
```

General Description

The `gai_strerror()` function returns a pointer to a text string describing the error value returned by a failure return from either the `getaddrinfo()` or `getnameinfo()` function. If the *ecode* is not one of the `EAI_xxx` values from the `<netdb.h>` header, then `gai_strerror()` returns a pointer to a string indicating an unknown error.

Subsequent calls to `gai_strerror()` will overwrite the buffer containing the text string.

Returned Value

If successful, `gai_strerror()` returns a pointer to a string describing the error. Upon failure, `gai_strerror()` will return `NULL` and set `errno` to one of the following:

Error Code**Description****ENOMEM**

Insufficient memory to allocate buffer for text string describing the error.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the message EDCV001I or EDCT002I is issued.

getaddrinfo() — Get address information**Format**

```
#define _OPEN_SYS_SOCKET_IPV6
#include <socket.h>
#include <netdb.h>
```

```
int getaddrinfo(const char *nodename,
               const char *servname,
               const struct addrinfo *hints,
               struct addrinfo **res);
```

General Description

The `getaddrinfo()` function translates the name of a service location (for example, a host name) and/or service name and returns a set of socket addresses and associated information to be used in creating a socket with which to address the specified service.

The *nodename* and *servname* arguments are either pointers to null-terminated strings or null pointers. One or both of these two arguments must be specified as a non-null pointer.

The format of a valid name depends on the protocol family or families. If a specific family is not given and the name could be interpreted as valid within multiple supported families, the function attempts to resolve the name in all supported families. When no errors are detected, all successful results are returned.

If the *nodename* argument is not null, it can be a descriptive name or it can be an address string. If the specified address family is `AF_INET`, `AF_INET6`, or `AF_UNSPEC`, valid descriptive names include host names. If the specified address family is `AF_INET` or `AF_UNSPEC`, address strings using standard dot notation as specified in `inet_addr()` are valid. If the specified address family is `AF_INET6` or `AF_UNSPEC`, standard IPv6 text forms described in `inet_pton()` are valid. In addition, scope information can be appended to the descriptive name or the address string using the format *nodename%scope* information. Scope information can be either an interface name or the numeric representation of an interface index suitable for use on this system.

If *nodename* is not null, the requested service location is named by *nodename*; otherwise, the requested service location is local to the caller.

If *servname* is null, the call returns network-level addresses for the specified *nodename*. If *servname* is not null, it is a null-terminated character string identifying the requested service. This can be either a descriptive name or a numeric representation suitable for use with the address family or families. If the specified address family is `AF_INET`, `AF_INET6`, or `AF_UNSPEC`, the service can be specified as a string specifying a decimal port number.

If the argument *hints* is not null, it refers to a structure containing input values that may direct the operation by providing options and by limiting the returned information to a specific socket type, address family and/or protocol. In the hints structure every member other than *ai_flags*, *ai_family*, *ai_socktype*, and *ai_protocol* must be zero or a null pointer. A value of `AF_UNSPEC` for *ai_family* means that the caller will accept any protocol family. A value of zero for *ai_socktype* means that the caller will accept any socket type. A value of zero for *ai_protocol* means that the caller will accept any protocol. If *hints* is a null pointer, the behavior must be as if it referred to a structure containing the value zero for the *ai_flags*, *ai_socktype*, and *ai_protocol* fields, and `AF_UNSPEC` for the *ai_family* field.

The *ai_flags* member to which the *hints* argument points can be set to 0 or be the bitwise inclusive OR of one or more of the following values:

- `AI_PASSIVE`
- `AI_CANONNAME`
- `AI_NUMERICHOST`
- `AI_NUMERICSERV`
- `AI_V4MAPPED`

getaddrinfo

- AI_ALL
- AI_ADDRCONFIG
- AI_EXTFLAGS

If the AI_PASSIVE bit is set in the ai_flags member of the hints structure, then the caller plans to use the returned socket address structure in a call to bind(). In this case, if the nodename argument is a null pointer, then the IP address portion of the socket address structure is set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6 address. If the AI_PASSIVE bit is not set in the ai_flags member of the hints structure, then the returned socket address structure is ready for a call to connect() (for a connection-oriented protocol) or either connect(), sendto(), or sendmsg() (for a connectionless protocol). In this case, if the nodename argument is a null pointer, then the IP address portion of the socket address structure is set to the loopback address.

If the AI_CANONNAME bit is set in the ai_flags member of the hints structure, then upon successful return the ai_canonname member of the first addrinfo structure in the linked list will point to a null-terminated string containing the canonical name of the specified nodename.

If the AI_NUMERICHOST bit is set in the ai_flags member of the hints structure, then a non-null nodename string must be a numeric host address string. Otherwise an error code of EAI_NONAME is returned. This flag prevents any type of name resolution service (for example, the DNS) from being called.

If the AI_NUMERICSERV flag is specified then a non-null servname string must be a numeric port string. Otherwise an error code EAI_NONAME is returned. This flag prevents any type of name resolution service, for example, NIS+ from being invoked.

If the AI_V4MAPPED 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 a value of AF_INET6 or the AF field contains AF_UNSPEC but IPv6 is not supported on the system, this flag is ignored. When the AF field has a value of AF_INET6 and AI_ALL is set, the AI_V4MAPPED flag must also be set to indicate that the caller will accept all addresses (IPv6 and IPv4-mapped IPv6 addresses).

If the AF field has a value of AF_UNSPEC 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.

If the AI_ADDRCONFIG flag is specified then a query for IPv6 address records should occur only if the node has at least one IPv6 source address configured. A

query for IPv4 address records will always occur, whether or not any IPv4 addresses are configured. The loopback address is not considered for this case as valid as a configured sources address.

All of the information returned by `getaddrinfo()` is dynamically allocated: the `addrinfo` structures, and the socket address structures and canonical node name strings pointed to by the `addrinfo` structures. To return this information to the system the function `freeaddrinfo()` is called.

Usage Notes

1. If the caller handles only TCP and not UDP, for example, then the `ai_protocol` member of the hints structure should be set to `IPPROTO_TCP` when `getaddrinfo()` is called.
2. If the caller handles only IPV4 and not IPv6, then the `ai_family` member of the hints structure should be set to `AF_INET` when `getaddrinfo()` is called.
3. Scope information is only pertinent to IPv6 link-local addresses. It is ignored for resolved IPv4 addresses and IPv6 addresses that are not link-local addresses.

Returned Value

If successful, `getaddrinfo()` returns 0 and a pointer to a linked list of one or more `addrinfo` structures through the `res` argument. The caller can process each `addrinfo` structure in this list by following the `ai_next` pointer, until a null pointer is encountered. In each returned `addrinfo` structure the three members `ai_family`, `ai_socktype`, and `ai_protocol` are the corresponding arguments for a call to the `socket()` function. In each `addrinfo` structure the `ai_addr` member points to a filled-in socket address structure whose length is specified by the `ai_addrlen` member. Upon failure, `getaddrinfo()` returns a non-zero error code. The error codes are as follows:

Error Code

Description

EAI_AGAIN

The name specified by the `Node_Name` or `Service_Name` parameter could not be resolved within the configured time interval, or the resolver address space has not been started. The request can be retried later. .

EAI_BADEXTFLAGS

The extended flags parameter had an incorrect setting.

EAI_BADFLAGS

The flags parameter had an incorrect setting.

EAI_FAIL

An unrecoverable error occurred.

EAI_FAMILY

The family parameter had an incorrect setting.

EAI_MEMORY

A memory allocation failure occurred during an attempt to acquire an `Addr_Info` structure.

EAI_NONAME

One of the following conditions occurred:

1. The name does not resolve for the specified parameters. At least one of the `Name` or `Service` operands must be specified.

getaddrinfo

2. The request name parameter is valid, but it does not have a record at the name server.

EAI_SERVICE

The service that was passed was not recognized for the specified socket type.

EAI_SOCKTYPE

The intended socket type was not recognized.

EAI_SYSTEM

A system error occurred.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the message EDCV001I or EDCT002I is issued.

getclientid() — Get the Identifier for the Calling Application

Format

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

```
int getclientid(int domain, struct clientid *clientid);
```

General Description

The `getclientid()` function call returns the identifier by which the calling application is known to the TCP/IP partition. The *clientid* can be used in the `givesocket()` and `takesocket()` calls.

Parameter

Description

domain The address domain requested.

clientid The pointer to a *clientid* structure to be filled.

The *clientid* structure is filled in by the call and returned as follows:

The *clientid* structure:

```
struct clientid {
    int domain;
    union {
        char name[8];
        struct {
            int NameUpper;
            pid_t pid;
        } c_pid;
    } c_name;
    char subtaskname[8];

    struct {
        char type;
        union {
            char specific[19];
            struct {
                char unused[3];
                int SockToken;
            } c_func;
        } c_reserved;
    };
};
```


Element**Description**

domain The input *domain* value returned in the domain field of the clientid structure.

c_name.name

The application program's partition name, left-justified and padded with blanks.

subtaskname

The calling program's task identifier.

c_reserved

Specifies binary zeros.

Returned Value

The value 0 indicates success. The value -1 indicates an error. The value of errno indicates the specific error.

errno Description**EFAULT**

Using the *clientid* parameter as specified would result in an attempt to access storage outside the caller's partition, or storage not modifiable by the caller.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1, and errno is to EFAULT. In this case the message EDCV001I or EDCT002I is issued.

gethostbyaddr() — Get a Host Entry by Address**Format**

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>
extern int h_errno;

struct hostent *gethostbyaddr(const void *address,
                              size_t     address_len,
                              int        domain);
```

General Description

The gethostbyaddr() call tries to resolve the host address through a name server, if one is present.

Parameter**Description**

address The pointer to a structure containing the address of the host. (An unsigned long for AF_INET or AF_INET6.)

address_len

The size of *address* in bytes.

domain The address domain supported (AF_INET or AF_INET6).

The gethostbyaddr() call returns a pointer to a **hostent** structure for the host address specified on the call.

gethostbyaddr

gethostbyaddr(), and gethostbyname() all use the same static area to return the **hostent** structure. This static area is only valid until the next one of these functions is called on the same thread.

The **netdb.h** include file defines the **hostent** structure and contains the following elements:

Element

Description

h_addr A pointer to the network address of the host.

h_addrtype

The type of address returned (AF_INET or AF_INET6).

h_aliases

A zero-terminated array of alternative names for the host.

h_length

The length of the address in bytes.

h_name

The official name of the host.

The following function is defined in **netdb.h** and should be used by multithreaded applications when attempting to reference *h_errno* return on error:

```
int *__h_errno(void);
```

This function returns a pointer to a thread-specific value for the *h_errno* variable.

Returned Value

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.

On unsuccessful completion, this function sets *h_errno* to indicate the error as follows:

Error Code

Description

HOST_NOT_FOUND

No such host is known.

TRY_AGAIN

A temporary error such as no response from a server, indicating the information is not available now but may be at a later time.

NO_RECOVERY

An unexpected server failure occurred from which there is no recovery.

NO_DATA

The server recognized the request and the name but no address is available. Another type of request to the name server might return an answer.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns a NULL pointer, and sets *h_errno* to NO_RECOVERY and *errno* to EVSE. In this case the message EDCV001I or EDCT002I is issued.

gethostbyname() — Get a Host Entry by Name

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>
extern int h_errno;

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

General Description

The `gethostbyname()` call tries to resolve the host name through a name server, if one is present. When a call is made to convert a symbolic name to an IP address, TCP/IP for VSE/ESA searches the local names table (created by DEFINE NAME) first. If this search fails, the name is passed to the specified DNSs (set with SET DNSx). TCP/IP for VSE/ESA will try each DNS, beginning with DNS1, until a response is received or all servers have been polled. The first server to respond determines if the request succeeds or fails. If the search within a DNS fails, the default domain string (as specified with SET DEFAULT_DOMAIN) is appended to the name (following a period) and the DNS is consulted the last time for the name resolution.

Parameter

Description

name The name of the host.

The `gethostbyname()` call returns a pointer to a **hostent** structure for the host name specified on the call.

`gethostent()`, `gethostbyaddr()`, and `gethostbyname()` all use the same static area to return the **hostent** structure. This static area is only valid until the next one of these functions is called on the same thread.

The **netdb.h** include file defines the **hostent** structure and contains the following elements:

Element

Description

h_addr A pointer to the network address of the host.

h_addrtype

The type of address returned; currently, it is always set to AF_INET.

h_aliases

A zero-terminated array of alternative names for the host.

h_length

The length of the address in bytes.

h_name

The official name of the host.

The following function is defined in **netdb.h** and should be used by multithreaded applications when attempting to reference *h_errno* return on error:

```
int *__h_errno(void);
```

Returned Value

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.

On unsuccessful completion, this function sets *h_errno* to indicate the error as follows:

Error Code

Description

HOST_NOT_FOUND

No such host is known.

TRY_AGAIN

A temporary error such as no response from a server, indicating the information is not available now but may be at a later time.

NO_RECOVERY

An unexpected server failure occurred from which there is no recovery.

NO_DATA

The server recognized the request and the name but no address is available. Another type of request to the name server might return an answer.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns a NULL pointer, and sets *h_errno* to **NO_RECOVERY** and *errno* to **EVSE**. In this case the message EDCV001I or EDCT002I is issued.

gethostent() — Get the Next Host Entry

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct hostent *gethostent(void);
```

General Description

The `gethostent()` call reads the next line of the data set which contains information about known hosts.

The **netdb.h** include file defines the **hostent** structure which contains the following elements:

Element

Description

h_name

The official name of the host.

h_aliases

A zero-terminated array of alternative names for host.

h_addrtype

The type of address.

h_length

The length of the address in bytes.

h_addr

A pointer to the network address of the host.

Returned Value

A pointer to a **hostent** structure indicates success. A NULL pointer indicates an error or end-of-file.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns a NULL pointer. In this case the message EDCV001I or EDCT002I is issued.

gethostid() — Get the Unique Identifier of the Current Host

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
```

```
long gethostid(void);
```

General Description

The `gethostid()` call gets the unique 32-bit identifier for the current host. This value is the default home Internet address.

Returned Value

The `gethostid()` call returns the 32-bit identifier of the current host, which should be unique across all hosts.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value 0. In this case the message EDCV001I or EDCT002I is issued.

gethostname() — Get the Name of the Host Processor

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
```

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

General Description

The `gethostname()` call returns the name of the host processor that the program is running on. Up to *namelen* characters are copied into the name array. The returned name is null-terminated unless there is insufficient room in the name array.

Parameter

Description

name The character array to be filled with the host name.

namelen

The length of *name*.

Returned Value

The value 0 indicates success; the value -1 indicates an error. The value of the error code indicates the specific error.

Error Code

Description

EFAULT

Using *name* and *namelen* would result in an attempt to copy the address into a portion of the caller's address space to which data cannot be written.

EMVSPARM

Incorrect parameters were passed to the service or function is not implemented.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1, and `errno` is set to EMVSPARM. In this case the message EDCV001I or EDCT002I is issued.

getibmopt() — Get IBM TCP/IP image

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <socket.h>
```

```
int getibmopt(int cmd, struct ibm_gettcpinfo *bfrp);
```

General Description

The `getibmopt()` call returns the number of TCP/IP images installed on a given z/VSE system and their status, versions, and names. With this information, the caller can dynamically choose the TCP/IP image with which to connect by using the `setibmopt()` call. The `getibmopt()` call is optional. If you do not use the `getibmopt()` call, follow the standard method to determine the connecting TCP/IP image.

Parameter

Description

cmd A value or the address of a fullword binary number specifying the command to be processed. The only valid value is 1.

bfrp The pointer to an `ibm_gettcpinfo` structure.

To set the TCP/IP image for a socket, the application should set values in the `ibm_tpcimage` structure as follows:

Element

Description

status 0 means is not known and need not be checked. Currently, this is the only value with meaning.

version

0 means the version is to be set on return if known.

name The name must be left justified, uppercase, padded with blanks, and be the name of an active TCP stack.

Returned Value

On successful return, the struct `ibm_tcpimage` buffer contains the status, version, and name of up to eight active TCP/IP images.

Error Code

Description

EOPNOTSUPP

This function is not supported.

EFAULT

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

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the message EDCV001I or EDCT002I is issued.

getnameinfo() — Get name information

Format

```
#define _OPEN_SYS_SOCKET_IPV6
#include <socket.h>
#include <netdb.h>
```

```
int getnameinfo(const struct sockaddr *sa, socklen_t salen,
                char *host, socklen_t hostlen,
                char *serv, socklen_t servlen,
                int flags);
```

General Description

The `getnameinfo()` function translates a socket address to a node name and service location. The `getnameinfo()` function looks up an IP address and port number provided by the caller in the DNS and system-specific database, and returns text strings for both in buffers provided by the caller.

The *sa* argument points to either a `sockaddr_in` structure (for IPv4) or a `sockaddr_in6` structure (for IPv6) that holds the IP address and port number. The `sockaddr_in6` structure may also contain a zone index value, if the IPv6 address represented by this `sockaddr_in6` structure is a link-local address. The *salen* argument gives the length of the `sockaddr_in` or `sockaddr_in6` structure.

If the socket address structure contains an IPv4-mapped IPv6 address or an IPv4-compatible IPv6 address, the embedded IPv4 address is extracted and the lookup is performed on the IPv4 address.

Note: The IPv6 unspecified address (“::”) and the IPv6 loopback address (“::1”) are not IPv4-compatible addresses. If the address is the IPv6 unspecified address, a lookup is not performed, and the `EAI_NONAME` error code is returned.

The node name associated with the IP address is returned in the buffer pointed to by the *host* argument. The caller provides the size of this buffer in the *hostlen* argument. The caller specifies not to return the node name by specifying a zero value for *hostlen* or a null *host* argument. If the node's name cannot be located, the numeric form of the node's address is returned instead of its name. If a zone index value was present in the `sockaddr_in6` structure, the numeric form of the zone

getnameinfo

index, or the interface name associated with the zone index, is appended to the node name returned, using the format node name%scope information.

If the size of the buffer specified in the *hostlen* argument is insufficient to contain the entire node name, or node name and scope information combination, up to *hostlen* characters are copied into the buffer as a null terminated string.

The service name associated with the port number is returned in the buffer pointed to by the *serv* argument, and the *servlen* argument gives the length of this buffer. The caller specifies not to the service name by specifying a zero value for *servlen* or a null *serv* argument. If the service's name cannot be located, the numeric of the service address (for example, its port number) is returned instead of its name.

If the size of the buffer specified in the *servlen* argument is insufficient to contain the entire service name, up to *servlen* characters are copied into the buffer as a null terminated string.

The final argument, *flags*, is a flag that changes the default actions of this function. By default the fully-qualified domain name (FQDN) for the host is returned.

If the flag bit NI_NOFQDN is set, only the node name portion of the FQDN is returned for local hosts.

If the flag bit NI_NUMERICHOST is set, the numeric form of the host's address is returned instead of its name.

If the flag bit NI_NAMEREQD is set, an error is returned if the host's name cannot be located.

If the flag bit NI_NUMERICSERV is set, the numeric form of the service address is returned (for example, its port number) instead of its name.

If the flag bit NI_NUMERICSCOPE is set, the numeric form of the scope identifier is returned (for example, zone index) instead of its name. This flag is ignored if the *sa* argument is not an IPv6 address.

If the flag bit NI_DGRAM is set, this specifies that the service is a datagram service, and causes `getservbyport()` to be called with a second argument of "udp" instead of its default of "tcp". This flag is required for the few ports (for example, [512,514]) that have different services for UDP and TCP.

Note: The three NI_NUMERICxxx flags are required to support the "-n" flag that many commands provide.

Returned Value

Upon successful completion, `getnameinfo()` returns the node and service names, if requested, in the buffers provided. The returned names are always null-terminated strings. A zero return value for `getnameinfo()` indicates successful completion; a non-zero return value indicates failure. The possible values for the failures are as follows:

Error Code	Description
------------	-------------

EAI_AGAIN

The specified host address could not be resolved within the configured time interval, or the resolver address space has not been started. The request can be retried later.

EAI_BADFLAGS

The flags parameter had an incorrect value.

EAI_FAIL

An unrecoverable error occurred.

EAI_FAMILY

The address family was not recognized, or the address length was not valid for the specified family.

EAI_MEMORY

A memory allocation failure occurred.

EAI_NONAME

The name does not resolve for the supplied parameter. One of the following occurred:

1. NI_NAMEREQD is set, and the host name cannot be located.
2. Both host name and service name were null.
3. The requested address is valid, but it does not have a record at the name server.

EAI_OVERFLOW

An argument buffer overflowed. The buffer specified for the host name or the service name was not sufficient to contain the entire resolved name, and the caller previously specified `_EDC_SUSV3=1`, indicating that truncation was not permitted.

EAI_SYSTEM

An unrecoverable error occurred.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the message EDCV001I or EDCT002I is issued.

getnetbyaddr() — Get a Network Entry by Address

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct netent *getnetbyaddr(ip_addr_t net, int type);
```

General Description

The `getnetbyaddr()` call searches the data set which contains information about known networks for the specified network address.

Parameter

	Description
<i>net</i>	The network address.
<i>type</i>	The address domain.

The `netent` structure is defined in the `netdb.h` include file and contains the following elements:

getnetbyaddr

Element	Description
<i>n_addrtype</i>	The type of network address.
<i>n_aliases</i>	An array, terminated with a NULL pointer, of alternative names for the network.
<i>n_name</i>	The official name of the network.
<i>n_net</i>	The network number, returned in host byte order.

Returned Value

A pointer to a **netent** structure indicates success. A NULL pointer indicates an error.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns a NULL pointer. In this case the message EDCV001I or EDCT002I is issued.

getnetbyname() — Get a Network Entry by Name

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct netent *getnetbyname(const char *name);
```

General Description

The `getnetbyname()` call searches the data set which contains information about known networks for the specified network name.

Parameter

Description

name The pointer to a network name.

The **netent** structure is defined in the **netdb.h** include file and contains the following elements:

Element	Description
<i>n_addrtype</i>	The type of network address.
<i>n_aliases</i>	An array, terminated with a NULL pointer, of alternative names for the network.
<i>n_name</i>	The official name of the network.
<i>n_net</i>	The network number, returned in host byte order.

Returned Value

A pointer to a **netent** structure indicates success. A NULL pointer indicates an error.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns a NULL pointer. In this case the message EDCV001I or EDCT002I is issued.

getnetent() — Get the Next Network Entry

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct netent *getnetent(void);
```

General Description

The `getnetent()` call reads the next entry of the data set which contains information about known networks.

The **netent** structure is defined in the **netdb.h** include file and contains the following elements:

Element

Description

n_addrtype

The type of network address.

n_aliases

An array, terminated with a NULL pointer, of alternative names for the network.

n_name

The official name of the network.

n_net

The network number, returned in host byte order.

Returned Value

A pointer to a **netent** structure indicates success. A NULL pointer indicates an error or end-of-file.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns a NULL pointer. In this case the message EDCV001I or EDCT002I is issued.

getpeername() — Get the Name of the Peer Connected to a Socket

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <socket.h>

int getpeername(int socket, struct sockaddr *name, size_t *namelen);
```

General Description

The `getpeername()` call returns the name of the peer connected to socket descriptor *socket*. *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. The size of the peer name is returned in bytes. If the actual length of the address is greater than the length of the supplied *sockaddr*, the stored address is truncated. The *sa_len* field of structure *sockaddr* contains the length of the untruncated address.

Parameter

Description

socket The socket descriptor.

name The Internet address of the connected socket that is filled by `getpeername()` before it returns. The exact format of *name* is determined by the domain in which communication occurs.

namelen

The size of the address structure pointed to by *name* in bytes.

Sockets in the AF_INET6 domain: For an AF_INET6 socket, the address is returned in a `sockaddr_in6` address structure. The `sockaddr_in6` structure is defined in the header file `in.h`.

Returned Value

The value 0 indicates success; the value -1 indicates an error. The value of the error code indicates the specific error.

Error Code

Description

EBADF

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

EFAULT

Using the *name* and *namelen* parameters as specified would result in an attempt to access storage outside of the caller's address space.

EINVAL

The *namelen* parameter is not a valid length.

ENOBUFS

`getpeername()` is unable to process the request due to insufficient storage.

ENOTCONN

The socket is not in the connected state.

EOPNOTSUPP

The operation is not supported for the socket protocol.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1, and `errno` is set to EBADF. In this case the message EDCV001I or EDCT002I is issued.

getprotobyname() — Get a Protocol Entry by Name

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct protoent *getprotobyname(const char *name);
```

General Description

The `getprotobyname()` call searches the data set, which contains information about known protocols, for the specified protocol name.

Parameter

Description

name The name of the protocol.

The **protoent** structure is defined in the **netdb.h** include file and contains the following elements:

Element

Description

p_aliases

An array, terminated with a NULL pointer, of alternative names for the protocol.

p_name

The official name of the protocol.

p_proto

The protocol number.

Returned Value

A pointer to a **protoent** structure indicates success. A NULL pointer indicates an error.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns a NULL pointer. In this case the message EDCV001I or EDCT002I is issued.

getprotobynumber() — Get a Protocol Entry by Number

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct protoent *getprotobynumber(int proto);
```

General Description

The `getprotobynumber()` call searches the data set which contains information about known protocols for the specified protocol number.

Parameter

Description

proto The protocol number.

getprotobynumber

The **protoent** structure is defined in the **netdb.h** include file and contains the following elements:

Element

Description

p_aliases

An array, terminated with a NULL pointer, of alternative names for the protocol.

p_name

The official name of the protocol.

p_proto

The protocol number.

Returned Value

A pointer to a **protoent** structure indicates success. A NULL pointer indicates an error.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns a NULL pointer. In this case the message EDCV001I or EDCT002I is issued.

getprotoent() — Get the Next Protocol Entry

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct protoent *getprotoent(void);
```

General Description

The `getprotoent()` call reads the data set which contains information about known protocols. The `getprotoent()` call returns a pointer to the next entry in the data set.

The **protoent** structure is defined in the **netdb.h** include file and contains the following elements:

Element

Description

p_aliases

An array, terminated with a NULL pointer, of alternative names for the protocol.

p_name

The official name of the protocol.

p_proto

The protocol number.

Returned Value

A pointer to a **protoent** structure indicates success. A NULL pointer indicates an error or end-of-file.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns a NULL pointer. In this case the message EDCV001I or EDCT002I is issued.

getservbyname() — Get a Service Entry by Name

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct servent *getservbyname(const char *name, const char *proto);
```

General Description

The `getservbyname()` call searches the data set, which contains information about known services for the first entry that matches the specified service name and protocol name. If *proto* is NULL, only the service name must match.

Parameter

Description

name The service name.
proto The protocol name.

The **servent** structure is defined in the **netdb.h** include file and contains the following elements:

Element

Description

s_aliases An array, terminated with a NULL pointer, of alternative names for the service.
s_name The official name of the service.
s_port The port number of the service.
s_proto The protocol required to contact the service.

Returned Value

A pointer to a **servent** structure indicates success. A NULL pointer indicates an error.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns a NULL pointer. In this case the message EDCV001I or EDCT002I is issued.

getservbyport() — Get a Service Entry by Port

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct servent *getservbyport(int port, const char *proto);
```

General Description

The `getservbyport()` call searches the data set, which contains information about services, for the first entry that matches the specified port number and protocol name. If *proto* is NULL, only the port number must match.

Parameter

	Description
<i>port</i>	The port number.
<i>proto</i>	The protocol name.

The **servent** structure is defined in the `netdb.h` include file and contains the following elements:

Element

	Description
<i>s_aliases</i>	An array, terminated with a NULL pointer, of alternative names for the service.
<i>s_name</i>	The official name of the service.
<i>s_port</i>	The port number of the service.
<i>s_proto</i>	The protocol required to contact the service.

Returned Value

A pointer to a **servent** structure indicates success. A NULL pointer indicates an error.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns a NULL pointer. In this case the message EDCV001I or EDCT002I is issued.

getservent() — Get the Next Service Entry

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct servent *getservent(void);
```

General Description

The `getservent()` call reads the next line of the data set and returns a pointer to the next entry in the data set which contains information about services.

The **servent** structure is defined in the `netdb.h` include file and contains the following elements:

Element

	Description
<i>s_aliases</i>	An array, terminated with a NULL pointer, of alternative names for the service.

s_name The official name of the service.

s_port The port number of the service.

s_proto The protocol required to contact the service.

Returned Value

A pointer to a **servent** structure indicates success. A NULL pointer indicates an error or end-of-file.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns a NULL pointer. In this case the message EDCV001I or EDCT002I is issued.

getsockname() — Get the Name of a Socket

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <socket.h>

int getsockname(int socket, struct sockaddr *name, size_t *namelen);
```

General Description

The `getsockname()` call stores the current name for the socket specified by the *socket* 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 the rest of the structure set to zero. For example, an unbound 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 zeroed.

If the actual length of the address is greater than the length of the supplied *sockaddr*, the stored address is truncated. The *sa_len* field of structure *sockaddr* contains the length of the untruncated address.

Parameter

Description

socket The socket descriptor.

name The address of the buffer into which `getsockname()` copies the name of *socket*.

namelen Must initially point to an integer that contains the size in bytes of the storage pointed to by *name*. Upon return that integer contains the size of the data returned in the storage pointed to by *name*.

Sockets in the AF_INET6 domain: For an AF_INET6 socket, the address is returned in a `sockaddr_6` address structure. The `sockaddr_in6` structure is defined in the header file **in.h**.

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

getsockname

can be discovered with a call to `getsockname()`.

Returned Value

The value 0 indicates success; the value -1 indicates an error. The value of the error code indicates the specific error.

Error Code

Description

EBADF

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

EFAULT

Using the *name* and *namelen* parameters as specified would result in an attempt to access storage outside of the caller's address space.

ENOBUFS

`getsockname()` is unable to process the request due to insufficient storage.

ENOTCONN

The socket is not in the connected state.

EOPNOTSUPP

The operation is not supported for the socket protocol.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and `errno` is set to `EOPNOTSUPP`. In this case the message `EDCV001I` or `EDCT002I` is issued.

getsockopt() — Get the Options Associated with a Socket

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <socket.h>

int getsockopt(int    socket,
               int    level,
               int    option_name,
               void   *option_value,
               size_t *option_len);
```

General Description

The `getsockopt()` call gets options associated with a socket. Not all options are supported by all address families. See each option for details. Options can exist at multiple protocol levels; they are always present at the highest socket level.

Parameter

Description

socket The socket descriptor.

level The level for which the option is set.

option_name

The name of a specified socket option.

option_value

The pointer to option data.

option_len

The pointer to the length of the option data.

If 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 IPv4 or IPv6 level, the level parameter must be set to `IPPROTO_IP` as defined in `socket.h` or `IPPROTO_IPV6` as defined in `in.h`. To manipulate options at any other level, such as the TCP level, supply the appropriate protocol number for the protocol controlling the option. The `getprotobyname()` call can be used to return the protocol number for a named protocol.

The *option_value* and *option_len* parameters are used to return data used by the particular get command. The *option_value* parameter points to a buffer that is to receive the data requested by the get command. The *option_len* parameter points to the size of the buffer pointed to by the *option_value* parameter. It must be initially set to the size of the buffer before calling `getsockopt()`. On return it is set to the actual size of the data returned.

All the socket level options except `SO_LINGER`, `SO_RCVTIMEO`, and `SO_SNDTIMEO` expect *option_value* to point to an integer and *option_len* to be set to the size of an integer. When the integer is nonzero, the option is enabled. When it is zero, the option is disabled. The `SO_LINGER` option expects *option_value* 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 zero 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 options are recognized at the socket level:

Option Description

SO_ACCEPTCONN

The socket had a `listen()` call.

SO_BROADCAST

Toggles the ability to broadcast messages. If this option is enabled, it allows the application to send broadcast messages over socket, if the interface specified in the destination supports the broadcasting of packets. This option has no meaning for stream sockets. This option is valid only for the `AF_INET` domain.

SO_DEBUG

Reports whether debugging information is being recorded. This option stores an int value.

SO_ERROR

Returns any pending error on the socket and clears the error status. You can use `SO_ERROR` to check for asynchronous errors on connected datagram sockets or for other asynchronous errors (errors that are not returned explicitly by one of the socket calls).

SO_KEEPALIVE

Toggles the TCP keep-alive mechanism for a stream socket. When activated, the keep-alive mechanism periodically sends a packet on an otherwise idle connection. If the remote TCP does not respond to the packet or to retransmissions of the packet, the connection is ended with the error ETIMEDOUT. Processes writing to that socket are notified with a SIGPIPE signal. This option stores an int value. This option is valid only for the AF_INET and AF_INET6 domains.

SO_LINGER

Lingers on close, if data is present. If 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 TCP/IP address space 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 only a finite amount of time trying to send the data. The `close()` call returns without blocking the caller. This option has meaning only for stream sockets.

SO_OOBINLINE

Toggles reception of out-of-band data. If this option is enabled, out-of-band data is placed in the normal data input queue as it is received. It is then available to `recv()`, `recvfrom()`, and `recvmsg()` without the need to specify the MSG_OOB flag in those calls. If this option is disabled, out-of-band data is placed in the priority data input queue as it is received. It is then available to `recv()`, `recvfrom()`, and `recvmsg()` only if the MSG_OOB flag is specified in those calls. This option has meaning only for stream sockets.

SO_RCVTIMEO

Reports the timeout value with the amount of time an input function waits until it completes. If a receive operation has blocked for this much time without receiving additional data, it returns with a partial count or `errno` set to EWOULDBLOCK if no data is received. The default for this option is zero, which indicates that a receive operation does not time out.

SO_REUSEADDR

Toggles local address reuse. If enabled, this option allows local addresses that are already in use to be bound. SO_REUSEADDR alters the normal algorithm used in the `bind()` call. The system checks at connect time to ensure that the local address and port do not have the same foreign address and port. The error EADDRINUSE is returned if the association already exists. If the 'SO_REUSEADDR' option is active, the following situation is 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. This option is valid only for the AF_INET and AF_INET6 domains.

SO_SNDBUF

Reports send buffer size information. This option stores an int value.

SO_SNDTIMEO

Reports the timeout value specifying the amount of time that an output function blocks due to flow control preventing data from being sent. If a send operation has blocked for this time, it returns with a partial count or with `errno` set to EWOULDBLOCK if no data is sent. The default for this option is zero, which indicates that a send operation does not time out.

SO_TYPE

This option returns the type of the socket. On return, the integer pointed to by `option_value` is set to `SOCK_STREAM` or `SOCK_DGRAM`. This option is valid for the `AF_INET` and `AF_INET6` domains.

The following options are recognized at the IPv4 level:

Option Description**IP_MULTICAST_IF**

Returns the interface IP address used for sending outbound multicast datagrams. The IP address is passed back using `struct in_addr`.

IP_MULTICAST_LOOP

Determines whether loopback is enabled or disabled. The loopback indicator is passed back as `u_char`. 0 means loopback is disabled and 1 means it is enabled.

IP_MULTICAST_TTL

Returns the IP time-to-live of outgoing multicast datagrams. The TTL value is passed back as `u_char`.

The following options are recognized at IPv6 level:

Option Description**IPV6_MULTICAST_HOPS**

Returns the hop limit value for outbound multicast datagrams. The hop limit value is passed back as `int`.

IPV6_MULTICAST_IF

Returns the interface index for the interface used for sending outbound multicast datagrams. The interface index is passed back using `struct u_int`.

IPV6_MULTICAST_LOOP

Determines whether loopback of outgoing multicast packets is enabled or disabled. The loopback indicator is passed back as `u_int`. 0 means the function is disabled and 1 means it is enabled.

IPV6_UNICAST_HOPS

Returns the hop limit value for outbound unicast datagrams. The hop limit value is passed back as `int`.

IPV6_V6ONLY

Determines whether a socket is restricted to IPv6 communications only. The option value is passed back as `int`. A nonzero value means the option is enabled (socket can only be used for IPv6 communications). 0 means the option is disabled.

Returned Value

The value 0 indicates success; the value -1 indicates an error. The value of the error code indicates the specific error.

Error Code**Description****EBADF**

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

getsockopt

EFAULT

Using *option_value* and *option_len* parameters would result in an attempt to access storage outside the caller's address space.

EINVAL

The specified option is invalid at the specified socket level.

ENOBUFS

Buffer space is not available to send the message.

ENOPROTOOPT

The *option_name* parameter is unrecognized, or the *level* parameter is not SOL_SOCKET.

ENOSYS

The function is not implemented. You attempted to use a function that is not yet available.

EOPNOTSUPP

The operation is not supported by the socket protocol.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time returns the value -1 and *errno* is set to EOPNOTSUPP. In this case the message EDCV001I or EDCT002I is issued.

Stack characteristics

TCP/IP for VSE/ESA supports option SO_LINGER only.

Example

The following are examples of the `getsockopt()` call. See “`setsockopt()` — Set Options Associated with a Socket” on page 183 for examples of how the `setsockopt()` call options are set.

```
int rc;
int s;
int option_value;
int option_len;
struct linger l;

:
/* Do I linger on close? */
option_len = sizeof(l);
rc = getsockopt( s,
                SOL_SOCKET,
                SO_LINGER,
                (char *)&l,
                &option_len);

if (rc == 0)
{
    if (option_len == sizeof(l))
    {
        if (l.l_onoff)
            /* yes I linger */
        else
            /* no I do not */
    }
}
```

givesocket() — Make the Specified Socket Available

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <socket.h>

int givesocket(int d, struct clientid *clientid);
```

General Description

The `givesocket()` call makes the specified socket available to a `takesocket()` call issued by another program. Any socket can be given. Typically, `givesocket()` is used by a master program that obtains sockets by means of `accept()`, and gives them to application programs that handle one socket at a time.

Parameter

Description

socket The descriptor of a socket to be given to another application.

clientid A pointer to a client ID structure which specifies the program to which the socket is to be given.

To pass a socket, the giving program first calls `givesocket()` with the client ID structure filled in as follows:

The `clientid` structure:

```
struct clientid {
    int domain;
    union {
        char name[8];
        struct {
            int NameUpper;
            pid_t pid;
        } c_pid;
    } c_name;
    char subtaskname[8];

    struct {
        char type;
        union {
            char specific[19];
            struct {
                char unused[3];
                int SockToken;
            } c_close;
        } c_func;
    } c_reserved;
};
```

Element Description

Element

Description

domain The *domain* of the input socket descriptor.

c_name.name

If the *clientid* was set by a `getclientid()` call, *c_name.name* can be:

- set to the application program's partition name, left-justified and padded with blanks. The application program can run in the same partition as the master program, in which case this field is set to the master program's partition.

givesocket

- set to blanks, so any z/VSE partition can take the socket.

subtaskname

If the *clientid* was set by a `getclientid()` call, *subtaskname* can be:

- set to the task identifier of the taker. This, combined with a *c_name.name* value, allows only a process with this *c_name.name* and *subtaskname* to take the socket.
- set to blanks. If *c_name.name* has a value and *subtaskname* is blank, any task with that *c_name.name* can take the socket.
- if *c_name_name* is set to blanks, the *subtaskname* parameter is set to blanks.

c_reserved.type

When set to `S0_CLOSE`, this indicates the socket should be automatically closed by `givesocket()`, and a unique socket identifying token is to be returned in *c_close.SockToken*. The *c_close.SockToken* should be passed to the taking program to be used as input to `takesocket()` instead of the socket descriptor. The now closed socket descriptor could be re-used by the time the `takesocket()` is called, so the *c_close.SockToken* should be used for `takesocket()`.

c_close.SockToken

The unique socket identifying token returned by `givesocket` to be used as input to `takesocket()`, instead of the socket descriptor when *c_reserved.type* has been set to `S0_CLOSE`.

c_reserved

Specifies binary zeros if an automatic close of a socket is not to be done by `givesocket()`.

Using Name and Subtaskname for Givesocket/Takesocket

1. The giving program calls `getclientid()` to obtain its client ID. The giving program calls `givesocket()` to make the socket available for a `takesocket()` call. The giving program passes its client ID along with the descriptor of the socket to be given to the taking program by the taking program's startup parameter list.
2. The taking program calls `takesocket()`, specifying the giving program's client ID and socket descriptor.
3. Waiting for the taking program to take the socket, the giving program uses `select()` to test the given socket for an exception condition. When `select()` reports that an exception condition is pending, the giving program calls `close()` to free the given socket.
4. If the giving program closes the socket before a pending exception condition is indicated, the connection is immediately reset, and the taking program's call to `takesocket()` is unsuccessful. Calls other than the `close()` call issued on a given socket return -1, with `errno` set to `EBADF`.

Note: For backward compatibility, a client ID can point to the struct client ID structure obtained when the target program calls `getclientid()`. In this case, only the target program, and no other programs in the target program's partition, can take the socket.

Returned Value

The value 0 indicates success. The value -1 indicates an error. The value of `errno` indicates the specific error.

Error Code	Description
-------------------	--------------------

EBADF

The *d* parameter is not a valid socket descriptor. The socket has already been given.

EFAULT

Using the *clientid* parameter as specified would result in an attempt to access storage outside the caller's partition.

EINVAL

The *clientid* parameter does not specify a valid client identifier or the *clientid* domain does not match the *domain* of the input socket descriptor.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1, and *errno* is set to *EINVAL*. In this case the message EDCV001I or EDCT002I is issued.

gsk_free_memory() — Free memory allocated for SSL

Format

```
#include <gskssl.h>

void gsk_free_memory(void *pointer,
                    void *future_use);
```

General Description

gsk_free_memory() frees the memory that is allocated for SSL.

Note: The distinguished name returned in the null-terminated string by the *gsk_get_dn_by_label()* call must be freed using *gsk_free_memory()*.

Parameter**Description**

pointer The address of the memory, returned to the application from a previous call to a SSL function that is to be freed.

future_use

Reserved for future use by SSL.

gsk_get_cipher_info() — Query Cipher Related Information

Format

```
#include <gskssl.h>

int gsk_get_cipher_info(int level,
                      gsk_sec_level *sec_level,
                      void *Reserved_for_future_use);
```

General Description

Queries cipher related information for SSL. *gsk_get_cipher_info()* determines the encryption level that the system can support and returns a list of cipher specs SSL can use. This allows an application to determine, at runtime, the level of SSL encryption that the installed application can request. This function is useful for programs that run on systems running across the globe.

gsk_get_cipher_info

You can use `gsk_get_cipher_info()` to determine the valid values that may be specified in the cipher specs of the `gsk_soc_init_data` structure used by `gsk_secure_soc_init()`.

Parameter

Description

level Determines the type of cipher information returned. Specify either `GSK_LOW_SECURITY` or `GSK_HIGH_SECURITY`. `GSK_LOW_SECURITY` causes only exportable cipher information to be returned. `GSK_HIGH_SECURITY` causes exportable and domestic cipher information to be returned. `GSK_LOW_SECURITY` is useful when setting up SSL communications with systems that may be located outside of the US and Canada where strong cryptographic functions are not available.

sec_level

The pointer to a `gsk_sec_level` structure.

Reserved_for_future_use

Reserved for future use by SSL.

The `gsk_sec_level` structure is defined in the `gskssl.h` header file as follows:

```
typedef struct _gsk_sec_level {
    int version; /* Output - SSL toolkit version */
    char v3cipher_specs [64]; /* Output - The sslv3 cipher specs */
    char v2cipher_specs [32]; /* Output - The sslv2 cipher specs */
    int security_level; /* Output - initially one of */
                        /* GSK_SEC_LEVEL_US, */
                        /* GSK_SEC_LEVEL_EXPORT, */
                        /* GSK_SEC_LEVEL_EXPORT_FR */
} gsk_sec_level;
```

The `gsk_sec_level` structure specifies information about the level of cryptography that is available on the system. The application must allocate the memory necessary for this structure. On successful return, the contents of the structure is set.

Returned Value

The `gsk_get_cipher_info()` call returns an integer. A value greater or equal to 0 indicates successful completion. A negative value indicates an error.

If `GSK_ERROR_IO` is returned, a general I/O error occurred and the value of `errno` indicates the specific error.

Note: `errno` might change during this operation. However, `errno` is not explicitly used by the SSL interface nor can `errno` be used to determine the cause of the error. The return value is the exclusive indicator of any potential errors from a SSL API.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific SSL function, the corresponding dummy routine in C Run-Time always returns the value `GSK_ERROR_UNSUPPORTED`, and `errno` is set to `EOPNOTSUPP`. In this case the message `EDCV001I` or `EDCT002I` is issued.

gsk_get_dn_by_label() — Get Distinguished Name Based on the Label

Format

```
#include <gskssl.h>

char * gsk_get_dn_by_label(char *label);
```

General Description

Returns the distinguished name for a key based on the label. The `gsk_initialize()` routine must be called before the `gsk_get_dn_by_label()` routine can be called. You can use this value for the `DName` field of the `gsk_soc_init_data` structure, which is used on calls to `gsk_secure_soc_init()`.

Note: The distinguished name returned in the null-terminated string must be freed using `gsk_free_memory()`.

Parameter

Description

label Specifies a null-terminated character string that contains the label for the key.

Returned Value

The `gsk_get_dn_by_label()` call returns a pointer to the distinguished name upon successful completion. A NULL value is returned if an error is encountered finding the specified label.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific SSL function, the corresponding dummy routine in C Run-Time always returns the value NULL, and `errno` is set to `EOPNOTSUPP`. In this case the message `EDCV001I` or `EDCT002I` is issued.

gsk_initialize() — Initialize the SSL Environment

Format

```
#include <gskssl.h>

int gsk_initialize(gsk_init_data *init_data);
```

General Description

Sets up the overall SSL environment for the current partition. Upon successful completion of `gsk_initialize()`, the application is ready to call SSL interfaces and to begin creating and using secure socket connections.

Note: Multiple calls to `gsk_initialize()` can be made as long as the existing SSL environment is cleaned up by a call to `gsk_uninitialize()` before the next call to `gsk_initialize()` is made.

Parameter

Description

init_data The pointer to a `gsk_init_data` structure.

The `gsk_init_data` structure is defined in the `gskssl.h` header file as follows:

gsk_initialize

```
typedef struct _gsk_init_data { /* Basic gsk SSL Toolkit
                               * initialization data
                               */
    char * sec_types;           /* Security protocol choice      */
                               /* (SSLV2|SSLV3|...|ALL          */
    char * keyring;            /* Keyring file name             */
                               /* Default roots used when NULL  */
    char * keyring_pw;        /* Keyring password              */
                               /* Ignored when keyring=NULL     */
    char * keyring_stash;
    long V2_session_timeout;   /* Number of seconds for SSLV2   */
                               /* session data to time out. 0-100 */
    long V3_session_timeout;   /* Number of seconds for SSLV3   */
                               /* session data to time out.     */
                               /* 0-86400 (1 day)              */
    char * LDAP_server;       /* Name or IP address of X500 host */
    int LDAP_port;           /* Port number of X500 host      */
    char * LDAP_user;         /* User name for X500 host       */
    char * LDAP_password;     /* Password of X500 host         */
    gsk_ca_roots LDAP_CA_roots; /* Which CA roots to use        */
    gsk_auth_type auth_type;   /* Client authentication type    */
} gsk_init_data;
```

The *sec_types* field specifies a null-terminated character string that identifies the security protocols that are to be used.

Note: SSLV2 is currently not used under VSE.

The *keyring* field specifies a null-terminated character string that identifies the sub library (format: "lib.sublib") used for keys and certificates.

The *keyring_pw* field is currently not used under VSE.

The *keyring_stash* field is currently not used under VSE.

The *V2_session_timeout* field is currently not used under VSE.

The *V3_session_timeout* field specifies the number of seconds for the SSLV3 session identifier to expire. The range is 0-86400 seconds (1 day).

The *LDAP_server* field is currently not used under VSE.

The *LDAP_port* field is currently not used under VSE.

The *LDAP_user* field is currently not used under VSE.

The *LDAP_password* field is currently not used under VSE.

The *LDAP_CA_roots* field specifies which CA roots to use for certificate verification. The supported values are: GSK_CA_ROOTS_LOCAL_ONLY and GSK_CA_ROOTS_LOCAL_AND_X500.

The *auth_type* field specifies the method to use for verifying the client's certificate. This field is only used when the *LDAP_CA_roots* field is set to GSK_CA_ROOTS_LOCAL_AND_X500. The supported values are: GSK_CLIENT_AUTH_LOCAL, GSK_CLIENT_AUTH_STRONG_OVER_SSL, GSK_CLIENT_AUTH_STRONG and GSK_CLIENT_AUTH_PASSTHRU.

Note: The *gsk_init_data* structure, along with the data it refers to, should remain accessible for the entire time the application makes use of SSL. In particular,

pointers in the *gsk_init_data* structure should not point to storage that is freed by the application or that is on the call stack.

Returned Value

The `gsk_initialize()` call returns an integer. The value `GSK_INITIALIZE_OK` indicates successful SSL initialization.

If `GSK_ERROR_IO` is returned, a general I/O error occurred and the value of `errno` indicates the specific error.

Note: `errno` may change during this operation. However, `errno` is not explicitly used by the SSL interface nor can `errno` be used to determine the cause of the error. The return value is the exclusive indicator of any potential errors from a SSL API.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific SSL function, the corresponding dummy routine in C Run-Time always returns the value `GSK_ERROR_UNSUPPORTED`, and `errno` is set to `EOPNOTSUPP`. In this case the message `EDCV001I` or `EDCT002I` is issued.

gsk_secure_soc_close() — Close a Secure Socket Connection

Format

```
#include <gskssl.h>

void gsk_secure_soc_close(gsk_soc_data *user_socket);
```

General Description

The function `gsk_secure_soc_close()` ends a secure socket connection and frees all the SSL resources for that secure socket connection.

Note:

1. If you do not call `gsk_secure_soc_close()`, the storage referenced by the *user_socket* parameter is not be freed.
2. The user application must close all socket descriptors opened by any socket API. `gsk_secure_soc_close()` does not close any open socket descriptors.

Parameter

Description

user_socket
The pointer to a *gsk_soc_data* structure.

Returned Value

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific SSL function, the message `EDCV001I` or `EDCT002I` is issued.

gsk_secure_soc_init() — Initialize Data Areas for a Secure Socket Connection

Format

```
#include <gskssl.h>

gsk_soc_data * gsk_secure_soc_init(gsk_soc_init_data *soc_init_data);
```

General Description

The function `gsk_secure_soc_init()` initializes the data areas necessary for SSL to initiate or accept a secure socket connection. Upon successful completion of `gsk_secure_soc_init()`, a handle is returned to the application. Then other calls using this secure socket connection can use this handle.

A complete SSL handshake is performed during this call based on the input specified in the `gsk_soc_init_data` structure. While SSL performs the mechanics of the SSL handshake, the application must supply the routines necessary to transport the SSL data during the SSL handshake, as well as for all subsequent read/write operations.

Note: These routines must be supplied as an external entry-point generated with `fetchep()`.

Parameter

Description

`soc_init_data`

The pointer to a `gsk_soc_init_data` structure.

The `gsk_soc_init_data` structure is defined in the `gskssl.h` header file as follows:

```
typedef struct _gsk_soc_init_data {
    int fd; /* file descriptor */
    gsk_handshake hs_type; /* client or server handshake */
    char * DName; /* keyring entry Distinguished
                 /* name. When NULL the default
                 /* keyring entry is used
    char * sec_type; /* Type of security protocol used
                 /* to protect this socket
    char * cipher_specs; /* SSLV2 cipher specs preference
    char * v3cipher_specs; /* SSLV3 cipher specs preference
                 /* and order
    int (* skread) /* User supplied READ function ptr
        (int fd, void * buffer, int num_bytes);
    int (* skwrite) /* User supplied WRITE function ptr
        (int fd, void * buffer, int num_bytes);
    unsigned char cipherSelected[3]; /* Cipher Spec used
    unsigned char v3cipherSelected[2]; /* Cipher Spec used
    int failureReasonCode; /* failure reason code
    gsk_cert_info * cert_info; /* This information is read from
                 /* from the client certificate
                 /* when client authentication is
                 /* enabled
    gsk_init_data * gsk_data;
} gsk_soc_init_data;
```

The `gsk_soc_init_data` structure specifies information about the characteristics for the secure sockets connection. In addition, SSL uses this structure to return information about the secure socket connection after it has been established.

The *fd* field specifies the socket descriptor for this connection. The socket descriptor is passed to the application routines specified in the *skread* and *skwrite* fields. These application-supplied routines can use the socket descriptor to perform the required reading/writing of the SSL data.

Note: The socket must be created, opened, and connected prior to calling `gsk_secure_soc_init()`. This implies that a client must perform the `socket()` and `connect()` calls prior to calling `gsk_secure_soc_init()`. For servers, this implies that the server must perform the `socket()`, `bind()`, `listen()`, and `accept()` calls prior to calling `gsk_secure_soc_init()`.

The *hs_type* field specifies how to perform the SSL handshake. The supported values are:

- `GSK_AS_CLIENT` to perform the SSL handshake as a client with authentication.
- `GSK_AS_SERVER` to perform the SSL handshake as a server.
- `GSK_AS_SERVER_WITH_CLIENT_AUTH` to perform the SSL handshake as a server that requires client authentication.
- `GSK_AS_CLIENT_NO_AUTH` to perform the SSL handshake as a client without authentication.

The *DName* field specifies a character string that is the Distinguished Name or label of the desired entry (certificate) in the key database file. The default key database file entry can be used by specifying a `NULL`.

The *sec_type* field specifies a null-terminated character string that identifies the security protocol that is used.

The *cipher_specs* field is currently not used under VSE.

The *v3cipher_specs* field specifies a null-terminated character string that contains the list of SSL Version 3.0 ciphers in the order of usage preference. Some values may not be valid depending on the level of cryptography that is installed on the system. Any combination of valid values may be used in any order. Refer to “`gsk_get_cipher_info()` — Query Cipher Related Information” on page 135 for information about determining the cipher specs supported by the system. If you specify a `NULL` value for *cipher_specs*, the default SSL Version 3.0 cipher specs are used.

The *skread* field specifies an entry point of an application provided I/O routine that performs a read function for SSL. This application must use `fetchep()` to register the entry point of this I/O routine, if this routine or any called subroutine refers to writable static or global variables. Parameters for this routine must be defined as specified in *skread*. SSL uses the *skread* routine while performing the SSL handshake during the `gsk_secure_soc_init()` call and the `gsk_secure_soc_read()` call. The *skread* routine can be implemented as follows:

```
int skread(int fd, void *data, int len){
    return(recv(fd, data, len, 0));
}
```

The *skwrite* field specifies an entry point of an application provided I/O routine that performs a write function for SSL. This application must use `fetchep()` to register the entry point of this I/O routine, if this routine or any called subroutine refers to writable static or global variables. Parameters for this routine must be as defined as specified in *skwrite*. SSL uses the *skwrite* routine while performing the

gsk_secure_soc_init

SSL handshake during the `gsk_secure_soc_init()` call and the `gsk_secure_soc_write()` call. The `skwrite` routine can be implemented as follows:

```
int skwrite(int fd, void *data, int len){
    return(send(fd, data, len, 0));
}
```

The `cipherSelected` field is currently not used under VSE.

The `v3cipherSelected` field specifies the architected SSL version 3.0 cipher spec value selected for this session.

The `failureReasonCode` field specifies the failure reason code for `gsk_secure_soc_init()`.

The `cert_info` field specifies the Distinguished Name components from the client's certificate. This parameter is only valid when client authentication is requested for a server using SSL. The `gsk_cert_info` structure is defined in the `gskssl.h` header file as follows:

```
typedef struct _gsk_cert_info { /* Client certificate information */
    char * cert_body;           /* Certificate body */
    int   cert_body_len;       /* Lenth of certificate body */
    char * sessionID;          /* Current session ID */
    int   newSessionID;        /* TRUE if sid is new */
    char * serial_num;         /* Serial number */
    char * common_name;        /* Common name of client */
    char * locality;           /* Locality */
    char * state_or_province;   /* State or Province */
    char * country;            /* Country */
    char * org;                /* Organization */
    char * org_unit;           /* Organizational Unit */
    char * issuer_common_name; /* Issuer's common name */
    char * issuer_locality;     /* Issuer's locality */
    char * issuer_state_or_province; /* Issuer's state or province */
    char * issuer_country;      /* Issuer's country */
    char * issuer_org;          /* Issuer's organization */
    char * issuer_org_unit;     /* Issuer's organizational unit */
} gsk_cert_info;
```

The `gsk_data` field specifies the `gsk_init_data` structure pointer. This field should point to the same `gsk_init_data` structure that was used during the `gsk_initialize()` function call.

Returned Value

Upon successful completion, `gsk_secure_soc_init()` returns a pointer to a structure of type `gsk_soc_data`. Save this pointer because this structure is used in subsequent SSL operations. The `gsk_soc_data` structure is defined in the `gskssl.h` header file as follows:

```
typedef struct _gsk_soc_data {
    void * sk_SSLHandle; /* gskssl connector to SSLHandlestr */
} gsk_soc_data;
```

If an error occurs the `failureReasonCode` field of the `gsk_soc_init_data` structure is used to indicate the error.

If the `failureReasonCode` field is set to `GSK_ERROR_IO`, a general I/O error occurred and the value of `errno` indicates the specific error.

Note: `errno` may change during this operation. However, `errno` is not explicitly used by the SSL interface nor can `errno` be used to determine the cause of the error. The `failureReasonCode` field of the `gsk_soc_init_data` structure is the exclusive indicator of any potential errors from a SSL API.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific SSL function, the corresponding dummy routine in C Run-Time always returns the value `NULL`, and `errno` is set to `EOPNOTSUPP`. In this case the message `EDCV001I` or `EDCT002I` is issued.

gsk_secure_soc_read() — Receive Data on a Secure Socket Connection

Format

```
#include <gskssl.h>

int gsk_secure_soc_read(gsk_soc_data *user_socket,
                      void *data_buffer,
                      int buffer_length);
```

General Description

The function `gsk_secure_soc_read()` receives data on a secure socket connection using the application specified read routine.

Parameter

Description

user_socket

The pointer to `gsk_soc_data` returned from `gsk_secure_soc_init()` that initialized the secure socket connection over which data is to be read.

data_buffer

The pointer to the user-supplied buffer in which the data is to be stored.

buffer_length

The number of bytes to be read. This must be less or equal to the length of the `data_buffer`.

The maximum length of the data returned will not exceed 32KB because SSL is a record level protocol and the largest record allowed is 32KB minus the necessary SSL record headers.

Improperly mixing calls to `gsk_secure_soc_read()` and any of the sockets read functions (`recv()`, `read()`, `readv()`, ...), while possible, is not recommended. This requires very close matching of operations between client and server programs. If any portion of an SSL record is read using a socket read function, a fatal SSL protocol error is detected when the next `gsk_secure_soc_read()` is performed.

SSL and socket reads and writes can be mixed, but they must be performed in matched sets. If a client application writes 100 bytes of data using one or more of the socket `send()` calls, the server application must read exactly 100 bytes of data using one or more of the socket `recv()` calls. This is also true for `gsk_secure_soc_read()` and `gsk_secure_soc_write()`.

Since SSL is a record-oriented protocol, SSL must receive an entire record before it can be decrypted and any data returned to the application. Thus, a `select()` may indicate that data is available to be read, but a subsequent `gsk_secure_soc_read()` may hang waiting for the remainder of the SSL record to be received.

Returned Value

The `gsk_secure_soc_read()` call returns an integer. A value of 0 or greater indicates the number of bytes read. A value of less than 0 indicates that an error occurred.

If `GSK_ERROR_IO` is returned, a general I/O error occurred and the value of `errno` indicates the specific error.

Note: `errno` may change during this operation. However, `errno` is not explicitly used by the SSL interface nor can `errno` be used to determine the cause of the error. The return value is the exclusive indicator of any potential errors from a SSL API.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific SSL function, the corresponding dummy routine in C Run-Time always returns the value `GSK_ERROR_UNSUPPORTED`, and `errno` is set to `EOPNOTSUPP`. In this case the message `EDCV001I` or `EDCT002I` is issued.

gsk_secure_soc_reset() — Refresh the Security Parameters

Format

```
#include <gskssl.h>

int gsk_secure_soc_reset(gsk_soc_data *user_socket);
```

General Description

The function `gsk_secure_soc_reset()` refreshes the security parameters, such as encryption keys, for this session.

Use `gsk_secure_soc_reset()` when a client or server needs to reset the SSL environment. Call `gsk_secure_soc_reset()` only after a successful call to `gsk_secure_soc_init()`. Also, use `gsk_secure_soc_reset()` when resuming or restarting a connection for an SSL session that was cached and when resetting the keys used for that connection.

Parameter

Description

user_socket

The pointer to *gsk_soc_data* structure returned from `gsk_secure_soc_init()`.

Returned Value

The `gsk_secure_soc_reset()` call returns an integer. A value of 0 indicates success. A value less than 0 indicates that an error occurred.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific SSL function, the corresponding dummy routine in C Run-Time always returns the value `GSK_ERROR_UNSUPPORTED`, and `errno` is set to `EOPNOTSUPP`. In this case the message `EDCV001I` or `EDCT002I` is issued.

Related Information

- “`gsk_secure_soc_init()` — Initialize Data Areas for a Secure Socket Connection” on page 140
- For more details refer to *TCP/IP for VSE 1.5 Optional Features*.

gsk_secure_soc_write() — Send Data on a Secure Socket Connection

Format

```
#include <gskssl.h>

int gsk_secure_soc_write(gsk_soc_data *user_socket,
                        void *data_buffer,
                        int buffer_length);
```

General Description

The function `gsk_secure_soc_write()` sends data on a secure socket connection using the application specified write routine is used to send the data over the secure socket connection.

Parameter

Description

user_socket

The pointer to *gsk_soc_data* returned from `gsk_secure_soc_init()` that initialized the secure socket connection over which data is to be written.

data_buffer

The pointer to the user-supplied buffer in which the data to be written is stored.

buffer_length

The the number of bytes to be written. This must be less or equal to the length of the *data_buffer*.

Note: SSL for VSE currently supports a maximum of 64KB to be sent with one `gsk_secure_soc_write()` call.

If the application data sent to a SSL application is greater than 32KB, multiple calls to `gsk_secure_soc_read()` must be made in order to read the entire block of application data.

SSL and socket reads and writes can be mixed, but they must be performed in matched sets. If a client application writes 100 bytes of data using one or more of the socket send calls, the server application must read exactly 100 bytes of data using one or more of the socket receive calls. This is also true for `gsk_secure_soc_read()` and `gsk_secure_soc_write()`. If a write buffer is separated into multiple buffers, the remote site of the secure socket connection must perform enough `gsk_secure_soc_read()` operations to read the complete buffer.

Returned Value

The `gsk_secure_soc_write()` call returns an integer. A value of 0 or greater indicates the number of bytes written. A value of less than 0 indicates that an error occurred.

If `GSK_ERROR_IO` is returned, a general I/O error occurred and the value of `errno` indicates the specific error.

Note: `errno` may change during this operation. However, `errno` is not explicitly used by the SSL interface nor can `errno` be used to determine the cause of the error. The return value is the exclusive indicator of any potential errors from a SSL API.

gsk_secure_soc_write

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific SSL function, the corresponding dummy routine in C Run-Time always returns the value `GSK_ERROR_UNSUPPORTED`, and `errno` is set to `EOPNOTSUPP`. In this case the message `EDCV001I` or `EDCT002I` is issued.

gsk_uninitialize() — Remove Current Settings for the SSL Environment

Format

```
#include <gskssl.h>

int gsk_uninitialize(void);
```

General Description

The function `gsk_uninitialize()` removes the current overall settings for the SSL environment. `gsk_uninitialize()` removes settings such as session timeout values, and SSL protocols.

Use `gsk_uninitialize()` when it is required to reset the SSL environment settings. Then, use `gsk_initialize()` to create a new set of SSL environment settings.

Note: Before calling `gsk_uninitialize()`, all SSL sessions created using the current SSL environment should be closed.

Returned Value

The `gsk_uninitialize()` call returns an integer. A value of 0 indicates success. A value less than 0 indicates that an error occurred.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific SSL function, the corresponding dummy routine in C Run-Time always returns the value `GSK_ERROR_UNSUPPORTED`, and is set to `EOPNOTSUPP`. In this case the message `EDCV001I` or `EDCT002I` is issued.

gsk_user_set() — Provide Callback Routines

Format

```
#include <gskssl.h>

int gsk_user_set(int user_data_fid,
                 void *user_input_data,
                 void *reserved);
```

General Description

The function `gsk_user_set()` allows the SSL application to provide callbacks rather than using the default SSL implementation.

Note: The function `gsk_user_set()` is currently not used under VSE.

Parameter

Description

user_data_fid

The integer value to specify the action to perform.

user_input_data

The pointer to specify the action specific information.

reserved

Reserved for future use by SSL and should be specified as NULL.

Returned Value

The `gsk_user_set()` call returns an integer. A value of 0 indicates success. A value less than 0 indicates that an error occurred.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific SSL function, the corresponding dummy routine in C Run-Time always returns the value `GSK_ERROR_UNSUPPORTED`, and `errno` is set to `EOPNOTSUPP`. In this case the message `EDCV001I` or `EDCT002I` is issued.

htonl() — Translate Address Host to Network Long

Format

```
#define _XOPEN_SOURCE_EXTENDED 1

#include <inet.h>
in_addr_t htonl (in_addr_t hostlong);
```

General Description

The `htonl()` call translates a long integer from host byte order to network byte order.

Parameter

Description

hostlong

Is typed to the unsigned long integer to be put into network byte order.

Note: For System *z*, host byte order and network byte order are the same. However, for cross platform portability reasons, it is recommended to use the routine whenever host to network byte order translation is required.

Returned Value

`htonl()` returns the translated long integer.

htons() — Translate an Unsigned Short Integer into Network Byte Order

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <inet.h>

in_port_t htons(in_port_t hostshort);
```

General Description

The `htons()` call translates a short integer from host byte order to network byte order.

Parameter

Description

hostshort

Is typed to the unsigned short integer to be put into network byte order.

Note: For System *z*, host byte order and network byte order are the same. However, for cross platform portability reasons, it is recommended to use the routine whenever host to network byte order translation is required.

Returned Value

htons() returns the translated short integer.

if_freenameindex() — Free the Memory Allocated by if_nameindex()

Format

```
#define _OPEN_SYS_SOCKET_IPV6
#include <net/if.h>
```

```
void if_freenameindex(struct if_nameindex *ptr);
```

General Description

The if_freenameindex() function frees the memory allocated by if_nameindex(). The *ptr* argument must be a pointer that was returned by if_nameindex().

Returned Value

No return value is defined.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the message EDCV001I or EDCT002I is issued.

if_indextoname() — Map a Network Interface Index to its Corresponding Name

Format

```
#define _OPEN_SYS_SOCKET_IPV6
#include <net/if.h>
```

```
char *if_indextoname(unsigned int ifindex, char *ifname);
```

General Description

The if_indextoname() function maps an interface index to its corresponding interface name. When this function is called, *ifname* must point to a buffer of at least IF_NAMESIZE bytes into which the interface name corresponding to interface index *ifindex* is returned. Otherwise, the function shall return a NULL pointer and set errno to indicate the error.

Returned Value

Error Code

Description

EINVAL

The ifindex parameter was zero, or the *ifname* parameter was NULL, or both.

ENOMEM

Insufficient storage is available to obtain the information for the interface name.

ENXIO

The *ifindex* does not yield an interface name.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the message EDCV001I or EDCT002I is issued.

if_nameindex() — Return all Network Interface Names and Indexes

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <net/if.h>
```

```
struct if_nameindex *if_nameindex(void);
```

General Description

The `if_nameindex()` function returns an array of `if_nameindex` structures, one structure per interface. The end of the array is indicated by a structure with an `if_index` of zero and an `if_name` of `NULL`. The `if_nameindex` structure holds the information about a single interface and is defined as a result of including the `<net/if.h>` header.

```
struct if_nameindex {
    unsigned int if_index; /* 1, 2, ... */
    char *if_name; /* null terminated name: "le0", ... */
};
```

The memory used for this array of structures along with the interface names pointed to by the `if_name` members is obtained dynamically. This memory is freed by calling the `if_freenameindex()` function.

Returned Value

If successful, `if_nameindex()` returns a pointer to an array of `if_nameindex` structures. Upon failure, `if_nameindex()` returns `NULL` and sets `errno` to one of the following:

Error Code**Description****ENOMEM**

Insufficient storage is available to supply the array.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the message EDCV001I or EDCT002I is issued.

if_nametoindex() — Map a Network Interface Name to its Corresponding Index

Format

```
#define _OPEN_SYS_SOCKET_IPV6
#include <net/if.h>
```

```
unsigned int if_nametoindex(const char *ifname);
```

General Description

The `if_nametoindex()` function returns the interface index corresponding to the interface name *ifname*.

Returned Value

If successful, `if_nametoindex()` returns the interface index corresponding to the interface name *ifname*. Upon failure, `if_nametoindex()` returns zero and sets `errno` to one of the following:

Error Code

Description

EINVAL

Non-valid parameter was specified. The *ifname* parameter was NULL.

ENOMEM

Insufficient storage is available to obtain the information for the interface name.

ENXIO

The specified interface name provided in the *ifname* parameter does not exist.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the message EDCV001I or EDCT002I is issued.

inet_addr() — Translate an Internet Address into Network Byte Order

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <inet.h>
#include <in.h>
```

```
in_addr_t inet_addr(const char *cp);
```

General Description

The `inet_addr()` call interprets character strings representing host addresses expressed in standard dotted-decimal notation and returns host addresses suitable for use as an Internet address.

Parameter

Description

cp A character string in standard dotted-decimal (.) notation.

Values specified in standard dotted-decimal notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When a 4-part address is specified, each part is interpreted as a byte of data and assigned, from left to right, to one of the 4 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 format convenient 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 format convenient 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 in C language syntax. A leading 0x implies hexadecimal; a leading 0 implies octal. A number without a leading 0 implies decimal.

Returned Value

The Internet address is returned in network byte order. If the Internet address is returned in error—for example, not in the correct format—INADDR_NONE (-1) is the returned value. INADDR_NONE is defined in the **in.h** include file.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C runtime always returns the value INADDR_NONE (-1).

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value INADDR_NONE (-1). In this case the message EDCV001I or EDCT002I is issued.

inet_lnaof() — Translate a Local Network Address into Host Byte Order

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <inet.h>

in_addr_t inet_lnaof(struct in_addr in);
```

General Description

The `inet_lnaof()` call breaks apart the Internet host address and returns the local network address portion.

Parameter	Description
<i>in</i>	The host Internet address.

Returned Value

The local network address is returned in host byte order.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1. In this case the message EDCV001I or EDCT002I is issued.

inet_makeaddr() — Create an Internet Host Address

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <inet.h>

struct in_addr inet_makeaddr(in_addr_t net, in_addr_t lna);
```

General Description

The `inet_makeaddr()` call takes a network number and a local network address and constructs an Internet address.

Parameter	Description
<i>net</i>	The network number.
<i>lna</i>	The local network address.

Returned Value

The Internet address is returned in network byte order.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns an `in_addr` struct where the field `s_addr` is set to -1. In this case the message EDCV001I or EDCT002I is issued.

inet_netof() — Get the Network Number from the Internet Host Address

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <inet.h>

in_addr_t inet_netof(struct in_addr in);
```

General Description

The `inet_netof()` call breaks apart the Internet host address and returns the network number portion.

Parameter	Description
-----------	-------------

in The Internet address in network byte order.

Returned Value

The network number is returned in host byte order.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1. In this case the message EDCV001I or EDCT002I is issued.

inet_network() — Get the Network Number from the Decimal Host Address

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <inet.h>

in_addr_t inet_network(const char *cp);
```

General Description

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.

Parameter

Description

cp A character string in standard, dotted decimal (.) notation.

Returned Value

The network number is returned in host byte order.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1. In this case the message EDCV001I or EDCT002I is issued.

inet_ntoa() — Get the Decimal Internet Host Address

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <inet.h>

char *inet_ntoa(struct in_addr in);
```

General Description

The `inet_ntoa()` call returns a pointer to a string expressed in the dotted-decimal notation. `inet_ntoa()` accepts an Internet address expressed as a 32-bit quantity in network byte order and returns a string expressed in dotted-decimal notation.

Parameter

Description

in The host Internet address.

Returned Value

Returns a pointer to the Internet address expressed in dotted-decimal notation. The storage pointed to exists on a per-thread basis and is overwritten by subsequent calls.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns a NULL pointer. In this case the message EDCV001I or EDCT002I is issued.

inet_ntop() — Convert Internet Address Format from Binary to Text

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <inet.h>
```

```
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
```

General Description

The `inet_ntop()` function converts from an Internet address in binary format, specified by *src*, to standard text format, and places the result in *dst*, if *size*, the space available in *dst*, is sufficient. The argument *af* specifies the family of the Internet address. This can be `AF_INET` or `AF_INET6`.

The argument *src* points to a buffer holding an IPv4 Internet address, if the *af* argument is `AF_INET`, or an IPv6 Internet address, if the *af* argument is `AF_INET6`. The address must be in network byte order.

The argument *dst* points to a buffer where the function will store the resulting text string. The *size* argument specifies the size of this buffer. The application must specify a non-NULL *dst* argument. For IPv6 addresses, the buffer must be at least 46 bytes. For IPv4 addresses, the buffer must be at least 16 bytes.

To allow applications to easily declare buffers of the proper size to store IPv4 and IPv6 addresses in string form, the following two constants are defined in `in.h`.

```
#define INET_ADDRSTRLEN 16
#define INET6_ADDRSTRLEN 46
```

Note: The `inet_ntop()` function has a dependency on the level of the Enhanced ASCII Extensions.

Returned Value

If successful, `inet_ntop()` returns a pointer to the buffer containing the converted address. If unsuccessful, `inet_ntop()` returns NULL and sets `errno` to one of the following values:

Error Code

Description

EAFNOSUPPORT

The address family specified in *af* is unsupported.

ENOSPC

The destination buffer *size* is too small.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the message EDCV001I or EDCT002I is issued.

inet_pton() — Convert Internet Address Format from Text to Binary

Format

```
#define _OPEN_SYS_SOCKET_IPV6
#include <inet.h>
```

```
int inet_pton(int af, const char *src, void *dst);
```

General Description

The `inet_pton()` function converts an Internet address in its standard text format into its numeric binary form. The argument *af* specifies the family of the address.

Note: AF_INET and AF_INET6 address families are currently supported.

The argument *src* points to the string being passed in. The argument *dst* points to a buffer into which `inet_pton()` stores the numeric address. The address is returned in network byte order. The caller must ensure that the buffer pointed to by *dst* is large enough to hold the numeric address.

If the *af* argument is AF_INET, `inet_pton()` accepts a string in the standard IPv4 dotted-decimal form:

```
ddd.ddd.ddd.ddd
```

where *ddd* is a 1 to 3 digit decimal number between 0 and 255.

If the *af* argument is AF_INET6, the *src* string must be in one of the following standard IPv6 text forms:

1. The preferred form is *x::x::x::x::x::x::x::x*, where the *x*'s are the hexadecimal values of the eight 16-bit pieces of the address. Leading zeros in individual fields can be omitted, but there should be at least one numeral in every field.
2. A string of contiguous zero fields in the preferred form can be shown as *::*. The *::* can only appear once in an address. Unspecified addresses (*0:0:0:0:0:0:0:0*) can be represented simply as *::*.
3. A third form that is sometimes more convenient when dealing with a mixed environment of IPv4 and IPv6 is *x::x::x::x::x::x.d.d.d.d.*, where *x*'s are the hexadecimal values of the six high-order 16-bit pieces of the address, and the *d*'s are the decimal values of the four low-order 8-bit pieces of the address (standard IPv4 representation).

Note: The `inet_pton()` function has a dependency on the level of the Enhanced ASCII Extensions.

Returned Value

If successful, `inet_pton()` returns 1 and stores the binary form of the Internet address in the buffer pointed to by *dst*.

inet_pton

If unsuccessful, because the input buffer pointed to by *src* is not a valid string, `inet_pton()` returns 0.

If unsuccessful, because the *af* argument is unknown, `inet_pton()` returns -1 and sets `errno` to one of the following values:

Error Code

Description

EAFNOSUPPORT

The address family specified in *af* is unsupported.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the message EDCV001I or EDCT002I is issued.

initapi() — Initialize Socket API for a Subtask

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <socket.h>

int initapi(int maxsock, const char *taskid);
```

General Description

The `initapi()` function initializes the socket API and sets the maximum number of sockets and the ID for a VSE subtask.

Parameter

Description

maxsock

Maximum number of sockets to use for the subtask.

taskid Name to set as ID for the subtask.

Returned Value

The function `initapi` returns the greatest descriptor number that could be assigned to the application. A positive value indicates success; the value -1 indicates an error. The value of `errno` indicates the specific error.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and `errno` is set to `EINVAL`. In this case the message EDCV001I or EDCT002I is issued.

ioctl() — Control Socket

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ioctl.h>
int ioctl(int socket, int cmd, ... /* arg */);
```

General Description

`ioctl()` performs a variety of control functions on sockets.

The *cmd* argument selects the control function to be performed and will depend on the socket being addressed.

The *arg* argument represents additional information that is needed by this specific device to perform the requested function. The type of *arg* depends upon the particular control request, but it is either an integer or a pointer to a request-specific data structure.

Sockets

The following ioctl() commands are used with sockets:

Command

Description

FIONBIO

Sets or clears nonblocking I/O for a socket. *arg* is a pointer 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.

SIOCGHOMEIF6

Gets the IPv6 home interfaces. *arg* is a pointer to a **NetConfHdr** structure, as defined in **ioctl.h**. A pointer to a **HomeIf** structure that contains a list of home interfaces is returned in the **NetConfHdr** pointed to by the argument.

SIOCGIFADDR

Gets the network interface address. *arg* is a pointer to an **ifreq** structure, as defined in **if.h**. The interface address is returned in the argument. This option is valid only for the AF_INET domain. This macro is protected by the **_OPEN_SYS_IF_EXT** feature.

SIOCGIFBRDADDR

Gets the network interface broadcast address. *arg* is a pointer to an **ifreq** structure, as defined in **if.h**. The interface broadcast address is returned in the argument. This option is valid only for the AF_INET domain. This macro is protected by the **_OPEN_SYS_IF_EXT** feature.

SIOCGIFCONF

Gets the network interface configuration. *arg* is a pointer to an **ifconf** structure, as defined in **if.h**. The interface configuration is returned in the buffer pointed to by the **ifconf** structure. The returned data's length is returned in the field that originally contained the length of the buffer. This option is valid only for the AF_INET domain. This macro is protected by the **_OPEN_SYS_IF_EXT** feature.

SIOCGIFDSTADDR

Gets the network interface destination address. *arg* is a pointer to an **ifreq** structure, as defined in **if.h**. The interface destination (point-to-point) address is returned in the argument. This option is valid only for the AF_INET domain. This macro is protected by the **_OPEN_SYS_IF_EXT** feature.

Terminal and Sockets Returned Value

The value 0 indicates success; the value -1 indicates an error. The value of the error code indicates the specific error.

Error Code

Description

EBADF

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

ioctl

EINVAL

The request is invalid or not supported.

EMVSPARM

Incorrect parameters were passed to the service.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and `errno` is set to `EINVAL`. In this case the message `EDCV001I` or `EDCT002I` is issued.

Example

The following is an example of the `ioctl()` call.

```
int s;
int dontblock;
int rc;
:
:
/* Place the socket into nonblocking mode */
dontblock = 1;
rc = ioctl(s, FIONBIO, (char *) &dontblock);
:
:
```

listen() — Prepare the Server for Incoming Client Requests

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <socket.h>

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

General Description

The `listen()` call applies only to stream sockets. It establishes a readiness to accept client connection requests, and creates a connection request queue of length *backlog* to queue incoming connection requests. Once full, additional connection requests are rejected.

Parameter

Description

socket The socket descriptor.

backlog Defines the maximum length for the queue of pending connections. This parameter is ignored. A value of 1 is always assumed.

The `listen()` call indicates a readiness to accept client connection requests. It transforms an active socket into a passive socket. Once called, *socket* can never be used as an active socket to initiate connection requests. Calling `listen()` is the third of four steps that a server performs to accept a connection. It is called after allocating a stream socket with `socket()`, and after binding a name to *socket* with `bind()`. It must be called before calling `accept()`.

If the *backlog* is less than 0, *backlog* is set to 0. If the *backlog* is greater than `SOMAXCONN`, as defined in `socket.h`, *backlog* is set to `SOMAXCONN`.

The value cannot exceed the maximum number of connections allowed by the installed TCP/IP.

Returned Value

The value 0 indicates success; the value -1 indicates an error. The value of the error code indicates the specific error.

Error Code

Description

EBADF

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

EDESTADDRREQ

The socket is not bound to a local address, and the protocol does not support listening on an unbound socket.

EINVAL

An invalid argument was supplied. The socket is not named (a `bind()` has not been done), or the socket is ready to accept connections (a `listen()` has already been done). The socket is already connected.

ENOBUFS

Insufficient system resources are available to complete the call.

EOPNOTSUPP

The *socket* parameter is not a socket descriptor that supports the `listen()` call.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and `errno` is set to `EOPNOTSUPP`. In this case the message `EDCV001I` or `EDCT002I` is issued.

maxdesc() — Get Socket Numbers to Extend Beyond the Default Range

Format

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

```
int maxdesc(int *totdesc, int *inetdesc);
```

General Description

Bulk mode sockets are not supported. Do not use this function.

Returned Value

If successful, `maxdesc()` returns 0.

If unsuccessful, `maxdesc()` returns -1 and sets `errno` to one of the following values:

Error Code

Description

EALREADY

Your program called `maxdesc()` after creating a socket, after a call to `setibmssockopt()`, or after a previous call to `maxdesc()`.

EFAULT

Using the `totdesc` parameter as specified results in an attempt to access storage outside of the caller's address space, or storage not modifiable by the caller.

ENOMEM

Your address space has insufficient storage.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the message EDCV001I or EDCT002I is issued.

ntohl() — Translate a Long Integer into Host Byte Order

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <inet.h>

in_addr_t ntohl(in_addr_t netlong);
```

General Description

The `ntohl()` call translates a long integer from network byte order to host byte order.

Parameter**Description**

netlong

Is typed to the unsigned long integer to be put into host byte order.

Note: For System *z*, host byte order and network byte order are the same. However, for cross platform portability reasons, it is recommended to use the routine whenever host to network byte order translation is required.

Returned Value

`ntohl()` returns the translated long integer.

ntohs() — Translate an Unsigned Short Integer into Host Byte Order

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <inet.h>

in_port_t ntohs(in_port_t netshort);
```

General Description

The `ntohs()` call translates a short integer from network byte order to host byte order.

Parameter**Description**

netshort

Is typed to the unsigned short integer to be put into host byte order.

Note: For System *z*, host byte order and network byte order are the same. However, for cross platform portability reasons, it is recommended to use the routine whenever host to network byte order translation is required.

Returned Value

`ntohs()` returns the translated short integer.

`poll()` — Monitor Activity on Socket Descriptors

Format 1

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <poll.h>

int poll(struct pollfd listptr[], nfds_t nmsgsfds, int timeout);
```

Format 2

```
#define _XOPEN_SOURCE_EXTENDED 1
#define _OPEN_MSGQ_EXT
#include <types.h>
#include <time.h>
#include <poll.h>

int poll(void *listptr, nmsgsfds_t nmsgsfds, int timeout);
```

General Description

The `poll()` function provides applications with a mechanism for multiplexing input/output over socket descriptors.

For each member of the array(s) pointed to by *listptr*, `poll()` examines the given socket descriptor for the event(s) specified in the member. The number of *pollfd* structures in the arrays are specified by *nmsgsfds*. The `poll()` function identifies those socket descriptors on which an application can read or write data, or on which an error event has occurred.

Parameter

Description

listptr A pointer to an array of *pollfd* structures. Each structure specifies a socket descriptor and the events of interest for this socket. To monitor socket descriptors, set the high-order halfword of *nmsgsfds* to 0, the low-order halfword to the number of *pollfd* structures to be provided, and pass a pointer to an array of *pollfd* structures.

nmsgsfds

The number of *pollfd* structures pointed to by *listptr*. This parameter is divided into two parts. The first half (the high-order 16 bits) is reserved for message queue identifiers. The second half (the low-order 16 bits) gives the number of *pollfd* structures containing socket descriptors to check. If either half of the *nmsgsfds* parameter is equal to a value of 0, the corresponding structures is assumed not to be present.

timeout

The amount of time, in milliseconds, to wait for an event to occur. If none of the defined events have occurred on any selected descriptor, `poll()` waits at least *timeout* milliseconds for an event to occur on any of the selected descriptors. If the value of *timeout* is 0, `poll()` returns immediately. If the value of *timeout* is -1, `poll()` blocks until a requested event occurs or until the call is interrupted.

poll

Each *pollfd* structure contains the following fields:

fd open socket descriptor
events requested events
revents returned events

The *events* and *revents* fields are bitmasks constructed by OR-ing a combination of event flags.

The following macros are provided to manipulate the *nmsgsfds* parameter and the return value from `poll()`:

Macro Description

`_SET_FDS_MSGS(nmsgsfds, nmsgs, nfds)`

Sets the high-order halfword of *nmsgsfds* to *nmsgs*, and sets the low-order halfword of *nmsgsfds* to *nfds*.

`_NFDS(n)`

If the return value *n* from `poll()` is non-negative, returns the number of socket descriptors that meet the read, write, and exception criteria. A descriptor may be counted multiple times if it meets more than one given criterion.

`_NMSGSGS(n)`

If the return value *n* from `poll()` is non-negative, returns the number of message queues that meet the read, write, and exception criteria. A message queue may be counted multiple times if it meets more than one given criterion.

Returned Value

Upon successful completion, `poll()` returns a non-negative value. A positive value indicates the total number of events that were found to be ready among the socket descriptors. The return value is similar to *nmsgsfds* in that the high-order 16 bits of the return value give the number associated with message queues, and the low-order 16 bits give the number associated with socket descriptors.

A value of 0 indicates that the call timed out and no socket descriptors have been selected. Upon failure, `poll()` returns -1 and sets `errno` to indicate the error.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and `errno` is set to `EINVAL`. In this case the message EDCV001I or EDCT002I is issued.

read() — Read From a Socket

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

ssize_t read(int fs, void *buf, ssize_t N );
```

General Description

From the socket indicated by the socket descriptor *fs*, the `read()` function reads *N* bytes of input into the memory area indicated by *buf*. If successful, `read()` changes the file offset by the number of bytes read. *N* should not be greater than `INT_MAX` (defined in the `limits.h` header file).

`Read()` is equivalent to `recv()` with no flags set.

Parameter

	Description
<i>fs</i>	The socket descriptor.
<i>buf</i>	The pointer to the buffer that receives the data.
<i>N</i>	The length in bytes of the buffer pointed to by the <i>buf</i> parameter.

Behavior for Sockets

The `read()` call reads data on a socket with descriptor *fs* and stores it in a buffer. The `read()` call applies only to connected sockets. This call returns up to *N* bytes of data. If there are fewer bytes available than requested, the call returns the number currently available. If data is not available for the socket *fs*, and the socket is in blocking mode, the `read()` call blocks the caller until data arrives. If data is not available, and the socket is in nonblocking mode, `read()` returns a -1 and sets the error code to `EWOULDBLOCK`. See “`ioctl()` — Control Socket” on page 156 or “`fcntl()` — Control Open Socket Descriptors” on page 104 for a description of how to set nonblocking mode.

For datagram sockets, this call returns the entire datagram that was sent, provided that the datagram fits into the specified buffer. Excess datagram data is discarded. 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.

Returned Value

If successful, `read()` returns the number of bytes actually read and placed in *buf*. This number is less than or equal to *N*. The value -1 indicates an error. The value 0 indicates the connection is closed.

If `read()` fails, it returns the value -1 and sets `errno` to one of the following:

EBADF

fs is not a valid socket descriptor.

ECONNRESET

A connection was forcibly closed by a peer.

EFAULT

Using the *buf* and *N* parameters would result in an attempt to access memory outside the caller's address space.

EINVAL

N contains a value that is less than 0, or the request is invalid or not supported, or the `STREAM` or multiplexer referenced by *fs* is linked (directly or indirectly) downstream from a multiplexer.

read

EIO An I/O error occurred.

ENOBUFS

Insufficient system resources are available to complete the call.

ENOTCONN

A receive was attempted on a connection-oriented socket that is not connected.

ETIMEDOUT

The connection timed out during connection establishment, or due to a transmission timeout on active connection.

EWOULDBLOCK

The socket is in nonblocking mode and data is not available to read.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and `errno` is set to `EINVAL`. In this case the message `EDCV001I` or `EDCT002I` is issued.

Example

The following are examples of the `read()` call.

```
#include <stdio.h>

/* Read from the socket aSocket
   and print number of byte read and string read.
   Return number of bytes read or -1 for no success.
 */
int readFromSocket(int aSocket)
{ int numberOfBytesReceived;
  char dataBuffer 255 ;          /* data to read */

  numberOfBytesReceived=
  read(aSocket, dataBuffer, sizeof(dataBuffer));
  if (numberOfBytesReceived < 0)
  { perror("read"); return -1; }
  else
  { dataBuffer[numberOfBytesReceived] = 0;
    printf("Read string '%s' (length %d).\n",
           dataBuffer,numberOfBytesReceived);
    return numberOfBytesReceived;
  }
}
```

readv() — Read Data on a Socket and Store in a Set of Buffers

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <uio.h>

ssize_t readv(int fs, const struct iovec *iov, int iovcnt);
```

General Description

The `readv()` call reads data from a socket with descriptor `fs` and stores it in a set of buffers. The data is scattered into the buffers specified by `iov[0]...iov[iovcnt-1]`.

Parameter

Description

`fs` The socket descriptor.

iov A pointer to an **iovec** structure.

iovcnt The number of buffers pointed to by the *iov* parameter.

The **iovec** structure is defined in **uio.h** and contains the following fields:

Element	Description
<i>iov_base</i>	The pointer to the buffer.
<i>iov_len</i>	The length of the buffer.

The descriptor refers to a connected socket.

This call returns a number of bytes of data equal to but not exceeding the sum of all the *iov_len* fields. 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 *fs*, and the socket is in blocking mode, `readv()` call blocks the caller until data arrives. If data is not available and *fs* is in nonblocking mode, `readv()` returns a -1 and sets the error code to `EWOULDBLOCK`.

Returned Value

If successful, the number of bytes read into the buffer is returned. The value -1 indicates an error. The value of `errno` indicates the specific error.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and `errno` is set to `EINVAL`. In this case the message `EDCV001I` or `EDCT002I` is issued.

recv() — Receive Data on a Socket

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <socket.h>

ssize_t recv( int    socket,
              void   *buf,
              size_t len,
              int    flags);
```

General Description

The `recv()` call receives data on a socket with descriptor *socket* 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 a datagram packet is too long to fit in the supplied buffer, datagram sockets discard excess bytes. If data is not available for the socket *socket*, and *socket* is in blocking mode, the `recv()` call blocks the caller until data arrives. If data is not available and *socket* is in nonblocking mode, `recv()` returns a -1 and sets the error code to `EWOULDBLOCK`. See “`fcntl()` — Control Open Socket Descriptors” on page 104 or “`ioctl()` — Control Socket” on page 156 for a description of how to set nonblocking mode.

For datagram sockets, this call returns the entire datagram that was sent, provided that the datagram fits 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.

Parameter**Description**

<i>socket</i>	The socket descriptor.
<i>buf</i>	The pointer to the buffer that receives the data.
<i>len</i>	The length in bytes of the buffer pointed to by the <i>buf</i> parameter.
<i>flags</i>	reserved zero

Returned Value

If successful, the length of the message or datagram in bytes is returned. The value -1 indicates an error. The value 0 indicates the connection is closed. The value of the error code indicates the specific error.

Error Code**Description****EBADF**

socket is not a valid socket descriptor.

ECONNRESET

A connection was forcibly closed by a peer.

EFAULT

Using the *buf* and *len* parameters would result in an attempt to access storage outside the caller's address space.

EINVAL

The request is invalid or not supported. The MSG_OOB flag is set and no out-of-band data is available.

ENOBUFS

Insufficient system resources are available to complete the call.

ENOTCONN

A receive is attempted on a connection-oriented socket that is not connected.

EOPNOTSUPP

The specified flags are not supported for this socket type or protocol.

ETIMEDOUT

The connection timed out during connection establishment, or due to a transmission timeout on active connection.

EWOULDBLOCK

socket is in nonblocking mode and data is not available to read.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and *errno* is set to *EINVAL*. In this case the message EDCV001I or EDCT002I is issued.

Stack characteristics

TCP/IP for VSE/ESA doesn't support the MSG_PEEK and MSG_OOB options.

recvfrom() — Receive Messages on a Socket

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <socket.h>

int recvfrom(int          socket,
             void          *buffer,
             size_t       length,
             int          flags,
             struct sockaddr *name,
             size_t       namelen);
```

General Description

The `recvfrom()` call receives data on a socket named by descriptor *socket* and stores it in a buffer. The `recvfrom()` call applies to any socket, whether connected or unconnected.

Parameter

Description

<i>socket</i>	The socket descriptor.
<i>buffer</i>	The pointer to the buffer that receives the data.
<i>length</i>	The length in bytes of the buffer pointed to by the <i>buffer</i> parameter.
<i>flags</i>	reserved zero
<i>name</i>	A pointer to a socket address structure from which data is received. If <i>name</i> is a nonzero value, the source address is returned.
<i>namelen</i>	The size of <i>name</i> in bytes.

If *name* is nonzero, the source address of the message is filled. *namelen* must first be initialized to the size of the buffer associated with *name*, and is then 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 excess bytes. If data is not available for the socket *socket*, and *socket* is in blocking mode, the `recvfrom()` call blocks the caller until data arrives. If data is not available and *socket* is in nonblocking mode, `recvfrom()` returns a -1 and sets the error code to EWOULDBLOCK. See “`fcntl()` — Control Open Socket Descriptors” on page 104 or “`ioctl()` — Control Socket” on page 156 for a description of how to set nonblocking mode.

For datagram sockets, this call returns the entire datagram that was sent, provided that the datagram fits 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.

Socket address structure for IPv6: For an AF_INET6 socket, the address is returned in a sockaddr_in6 address structure. The sockaddr_in6 structure is defined in the header file **in.h**.

Returned Value

If successful, the length of the message or datagram in bytes is returned. The value 0 indicates the connection is closed, the value -1 indicates an error. The value of the error code indicates the specific error.

Error Code

Description

EBADF

socket is not a valid socket descriptor.

ECONNRESET

The connection was forcibly closed by a peer.

EFAULT

Using the *buffer* and *length* parameters would result in an attempt to access storage outside the caller's address space.

EINVAL

The request is invalid or not supported. The MSG_OOB flag is set and no out-of-band data is available.

ENOBUFS

Insufficient system resources are available to complete the call.

ENOTCONN

A receive is attempted on a connection-oriented socket that is not connected.

EOPNOTSUPP

The specified flags are not supported for this socket type.

ETIMEDOUT

The connection timed out during connection establishment, or due to a transmission timeout on active connection.

EWouldBlock

socket is in nonblocking mode and data is not available to read.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and errno is set to EINVAL. In this case the message EDCV001I or EDCT002I is issued.

recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <socket.h>

ssize_t recvmsg(int socket, struct msghdr *msg, int flags);
```

General Description

The `recvmsg()` call receives messages on a socket with descriptor *socket* and stores them in an array of message headers.

Parameter

Description

- socket* The socket descriptor.
- msg* An array of message headers into which messages are received.
- flags* The *flags* parameter is set by specifying one or more flags. If more than one flag is specified, the logical OR operator (`|`) must be used to separate them.

A message header is defined by a **msg_hdr** structure. A definition of this structure can be found in the **socket.h** include file and contains the following elements:

Element

Description

- msg_iov*
An array of *iovec* buffers into which the message is placed.
- msg_iovlen*
The number of elements in the *msg_iov* array.
- msg_name*
A pointer to a buffer where the sender's address is stored.
- msg_namelen*
The size of the address buffer.
- msg_control*
Ancillary data, see below.
- msg_controllen*
Ancillary data buffer length.
- msg_flags*
Flags on received message.

Ancillary data consists of a sequence of pairs, each consisting of a **cmsghdr** structure followed by a data array. The data array contains the ancillary data message, and the **cmsghdr** structure contains descriptive information that allows an application to correctly parse the data.

Element

Description

- cmsg_len*
Data byte count, including header.
- cmsg_level*
Originating protocol.
- cmsg_type*
Protocol-specific type.

The **socket.h** header file defines the following macros to gain access to the data arrays in the ancillary data associated with a message header:

recvmsg

MSG_DATA(*msg*)

If the argument is a pointer to a **msg_hdr** structure, this macro returns an unsigned character pointer to the data array associated with the **msg_hdr** structure.

MSG_NXTHDR(*mhdr, msg*)

If the first argument is a pointer to a **msg_hdr** structure and the second argument is a pointer to a **msg_hdr** structure in the ancillary data, pointed to by the **msg_control** field of that **msg_hdr** structure, this macro returns a pointer to the next **msg_hdr** structure, or a null pointer if this structure is the last **msg_hdr** in the ancillary data.

MSG_FIRSTHDR(*mhdr*)

If the argument is a pointer to a **msg_hdr** structure, this macro returns a pointer to the first **msg_hdr** structure in the ancillary data associated with this **msg_hdr** structure, or a null pointer if there is no ancillary data associated with the **msg_hdr** structure.

The `recvmsg()` call applies to sockets, regardless of whether they are in the connected state.

This call returns the length of the data received. If data is not available for the socket *socket*, and *socket* is in blocking mode, the `recvmsg()` call blocks the caller until data arrives. If data is not available and *socket* is in nonblocking mode, `recvmsg()` returns a -1 and sets the error code to EWOULDBLOCK.

On successful completion, the **msg_flags** member for the message header is the bitwise-inclusive OR of all flags that indicate conditions detected for the received message.

Socket address structure for IPv6: For an AF_INET6 socket, the address is returned in a `sockaddr_in6` address structure. The `sockaddr_in6` structure is defined in the header file **in.h**.

Returned Value

If successful, the length of the message in bytes is returned. The value -1 indicates an error. The value of `errno` indicates the specific error.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and `errno` is set to EINVAL. In this case the message EDCV001I or EDCT002I is issued.

select() — Monitor Activity on Sockets

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <types.h>
#include <time.h>

int select(int          nmsgsfds,
           fd_set      *readlist,
           fd_set      *writelist,
           fd_set      *exceptlist,
           struct timeval *timeout);
```

General Description

The `select()` call monitors activity on a set of sockets until a timeout occurs, to see if any of the sockets have read, write, or exception processing conditions pending.

Parameter

Description

num The number of socket descriptors to check.

If your application allocates sockets 3, 4, 5, 6, and 7 and you want to check all of your allocations, *num* should be set to 8, the highest descriptor you specified + 1. If your application checks sockets 3 and 4, *num* should be set to 5.

readlist, writelist, exceptlist

Pointers to `fd_set` types, arrays of message queue identifiers, or `sellist` structures to check for reading, writing, and exceptional conditions, respectively. The type of parameter to pass depends on whether you want to monitor socket descriptors, message queue identifiers, or both. To monitor socket descriptors, set the high-order halfword of *nmsgsfds* to 0, the low-order halfword to (highest descriptor number + 1), and use `fd_set` pointers.

timeout

The pointer to the time to wait for the `select()` call to complete.

If *timeout* is not a NULL pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a NULL pointer, the `select()` call blocks until a socket or message becomes ready. To poll the sockets and return immediately, *timeout* should be a non-NULL pointer to a zero-valued `timeval` structure.

To allow you to test more than one socket at a time, the sockets to test are placed into a bit set of type `fd_set`. A bit set is a string of bits such that if *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, bit 33 is set to 1. If socket 33 is not an element of a bit set, 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. If your program needs to allocate a large number of sockets, you may need to increase the size of the bit sets. Increasing the size of the bit sets should be done when you compile the program. To increase the size of the bit sets, define `FD_SETSIZE` before including `time.h`. `FD_SETSIZE` is the largest value of any socket that your program expects to use `select()` on. It is defined to be 2048 in `time.h`. However, TCP/IP for VSE allows for 8000 sockets.

Note:

1. `FD_SETSIZE` may only be defined by the application program if the extended version of `select()` is used (by defining `_OPEN_MSGQ_EXT`). Do NOT define `FD_SETSIZE` in your program if a `sellist` structure is used.
2. If your application program requires a large number of socket descriptors, you should protect your code from possible runtime errors by:
 - Adding a check before your `select()` or `selectex()` calls to see if *num* is larger than `FD_SETSIZE`.
 - Dynamically allocate bit strings large enough to hold the largest descriptor value in your application program, rather than rely on the static bit strings

select

created at compile time. When allocating your own bit strings, use `malloc()` to define an area large enough to represent each bit, rounded up to the next 4-byte multiple. For example, if your largest descriptor value is 31, you need 4 bytes; if your largest descriptor is 32, you need 8 bytes.

- If you dynamically allocate your own bit strings, the `FD_ZERO()` macro will *not* work. The application must zero that storage, by using the `memset` function—that is, `memset(ptr, 0, mallocsize)`. The other macros can be used with the dynamically allocated bit strings, as long as the descriptor you are manipulating is within the bit string. If the descriptor number is larger than the bit string, unpredictable results can occur.

The application program must make sure that the parameters *readlist*, *writelist*, and *exceptlist* point to bit strings that are as large as the bit string size in parameter *num*. TCP/IP services will try to access bits 0 through *num-1-1*, for each of the bit strings. If the bit strings are too short, you will receive unpredictable results when you run your application program.

The following macros are provided to manipulate bit sets.

Macro Description

`FD_ZERO(&fdset)`

Sets all bits in the bit set *fdset* to zero. After this operation, the bit set does not contain sockets as elements. This macro should be called to initialize the bit set before calling `FD_SET()` to set a socket as a member.

Note: If you used `malloc()` to dynamically allocate a new area, the `FD_ZERO()` macro can cause unpredictable results and should *not* be used. You should zero the area using the `memset()` function.

`FD_SET(sock, &fdset)`

Sets the bit for the socket *sock* to a 1, making *sock* a member of the bit set *fdset*.

`FD_CLR(sock, &fdset)`

Clears the bit for the socket *sock* in bit set *fdset*. This operation sets the appropriate bit to a zero.

`FD_ISSET(sock, &fdset)`

Returns nonzero if *sock* is a member of the bit set *fdset*. Returns zero if *sock* is not a member of *fdset*. (This operation returns the 32-bit value which includes the bit representing *sock*.)

The following macros are provided to manipulate the *nmsgsfds* parameter and the return value from `select()`:

Macro Description

`_SET_FDS_MSGS(nmsgsfds, nmsgs, nfds)`

Sets the high-order halfword of *nmsgsfds* to *nmsgs*, and sets the low-order halfword of *nmsgsfds* to *nfds*.

`_NFDS(n)`

If the return value *n* from `select()` is non-negative, returns the number of descriptors that meet the read, write, and exception criteria. A descriptor may be counted multiple times if it meets more than one given criterion.

`_NMSGs(n)`

If the return value *n* from `select()` is non-negative, returns the number of message queues that meet the read, write, and exception criteria.

A socket is ready for reading when incoming data is buffered for it or when a connection request is pending. To test whether any sockets are ready for reading, use either `FD_ZERO()` or `memset()`, if the function was dynamically allocated, to initialize the `fdset` bit set in *readlist* and invoke `FD_SET()` for each socket to test.

A socket is ready for writing if there is buffer space for outgoing data. A nonblocking stream socket in the process of connecting (`connect()` returned `EINPROGRESS`) is selected for write when the `connect()` completes. A call to `write()`, `send()`, or `sendto()` does not block provided that the amount of data is less than the amount of buffer space. To test whether any sockets are ready for writing, initialize the `fdset` bit set in *writelst* with either `FD_ZERO()` or `memset()`, if dynamically allocated, and use `FD_SET()` for each socket to test.

The programmer can pass `NULL` for any of the *readlist*, *writelst*, and *exceptlist* parameters. However, when they are not `NULL`, they must all point to the same type of structures.

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 its status. The `select()` call tests only sockets in the range 0 to *num*-1.

Returned Value

The value -1 indicates the error code should be checked for an error. The value zero indicates an expired time limit.

When the return value is greater than 0, it is similar to *nmsgsfds* in that the high-order 16 bits give the number of message queues, and the low-order 16 bits give the number of descriptors. These values indicate the sum total that meet each of the read, write, and exception criteria. If the return value for socket descriptors is greater than 65,535, only 65,535 is reported.

If the return value is greater than zero, 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 zero. Use the macro `FD_ISSET()` with each socket to test its status.

Error Code

Description

EBADF

One of the bit sets specified an invalid socket or a message queue identifier is invalid. `FD_ZERO()` was probably not called to clear the bit set before the sockets were set.

EFAULT

One of the parameters contained an invalid address.

EINVAL

One of the fields in the **timeval** structure is invalid, or there was an invalid *nmsgsfds* value.

EIO

One of the sockets being selected has become inoperative due to a network problem. This can occur for a socket if TCP/IP is shutdown. To find out which descriptor is bad, you can code a loop to individually `select()` on each descriptor, without waiting, until you get a failure.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C

select

Run-Time always returns the value -1, and errno is set to EIO. In this case the message EDCV001I or EDCT002I is issued.

Example

The following are examples of the select() call.

```
#define _OPEN_MSGQ_EX      /* needed for _SET_FDS_MSGS macro */
#include <time.h>
#include <types.h>
#include <stdio.h>

/* This function returns
   -1 if an error occurred
   0 if aSocket is NOT ready for read
   1 if aSocket is ready for read.
*/
int testSocketReadyForRead(int aSocket)
{
    fd_set socketSet;
    struct timeval timeout;
    int rc, number;

    /* Initialize timeout structure. */
    timeout.tv_sec=1;    /* seconds */

    /* Initialize socket set bits and add sockets to be examined. */
    FD_ZERO(&socketSet)
    FD_SET(aSocket, &socketSet);

    /* Set the number parameter. */
    _SET_FDS_MSGS(number,
                  0, /* don't monitor message queues */
                  aSocket+1);

    /* check for READ availability on this socket */
    rc=select(number,
              &socketSet, /* set of sockets to check for readability */
              NULL, /* set of sockets to check whether ready to write */
              NULL, /* set of sockets to check for pending exceptions */
              &timeout);

    if (rc<0)
    { perror("select");
      return rc;
    }
    else return (FD_ISSET(aSocket,&socketSet) != 0);
}
```

selectex() — Monitor Activity on Sockets

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#define _ALL_SOURCE
#include <types.h>
#include <time.h>

int selectex( int          nmsgsfds,
              fd_set      *readlist,
              fd_set      *writelist,
              fd_set      *exceptlist,
              struct timeval *timeout,
              int          *ecbptr);
```


General Description

The `selectex()` call provides an extension to the `select()` call by allowing you to use an ECB that defines an event not described by *readlist*, *writelist*, or *exceptlist*.

The `selectex()` call monitors activity on a set of sockets until a timeout occurs, or until the ECB is posted, to see if any of the sockets have read, write, or exception processing conditions pending.

See `select()` for more information.

Parameter

Description

num The number of socket descriptors to check. (Refer to `select()` for a full description of this and other parameters below.)

readlist A pointer to an `fd_set` type to check for reading.

writelist

A pointer to an `fd_set` type to check for writing.

exceptlist

A pointer to an `fd_set` type to be checked for exceptional pending conditions.

timeout

The pointer to the time to wait for the `selectex()` call to complete.

ecbptr This variable can contain one of the following values:

1. A pointer to a user event control block. To specify this usage of *ecbptr*, the high-order bit must be set to '0'B.
2. A pointer to a list of ECB pointers. To specify this usage of *ecbptr*, the high order bit must be set to '1'B.

The list can contain the pointers for up to 254 ECBs. The high-order bit of the last pointer in the list must be set to '1'B.

3. A NULL pointer. This indicates no ECBs are specified.

Returned Value

The value -1 indicates the error code should be checked for an error. The value 0 indicates an expired time limit or that the ECB is posted.

When the return value is greater than 0, this value indicates the sum total that meet each of the read, write, and exception criteria. Note that a descriptor may be counted multiple times if it meets more than one requested criterion. If the return value for socket descriptors is greater than 65,535, only 65,535 is reported.

If the return value is greater than zero, 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 zero. Use the macro `FD_ISSET()` with each socket to test its status.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and `errno` is set to `EIO`. In this case the message `EDCV001I` or `EDCT002I` is issued.

send() — Send Data on a Socket

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <socket.h>

ssize_t send(int socket, const void *msg, size_t length, int flags);
```

General Description

The `send()` call sends data on the socket with descriptor `socket`. The `send()` call applies to all connected sockets.

Parameter

	Description
<code>socket</code>	The socket descriptor.
<code>msg</code>	The pointer to the buffer containing the message to transmit.
<code>length</code>	The length of the message pointed to by the <code>msg</code> parameter. Unless the PTF for APAR PQ55591 is installed, the maximum number of bytes to be specified is 64K.
<code>flags</code>	reserved zero

If there is not enough available buffer space to hold the socket data to be transmitted, and the socket is in blocking mode, `send()` blocks the caller until additional buffer space becomes available. If the socket is in nonblocking mode, `send()` returns a -1 and sets the error code to `EWOULDBLOCK`. See “`fcntl() — Control Open Socket Descriptors`” on page 104 or “`ioctl() — Control Socket`” on page 156 for a description of how to set nonblocking mode.

For datagram sockets, this call sends the entire datagram, provided that the datagram fits 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.

Returned Value

The value -1 indicates locally detected errors. The value of the error code indicates the specific error. No indication of failure to deliver is implicit in a `send()` routine.

A value of 0 or greater indicates the number of bytes sent, however, this does not assure that data delivery was complete.

Error Code

	Description
EBADF	<code>socket</code> is not a valid socket descriptor.
ECONNRESET	A connection was forcibly closed by a peer.
EDESTADDRREQ	The socket is not connection-oriented and no peer address is set.

EFAULT

Using the *msg* and *length* parameters would result in an attempt to access storage outside the caller's address space.

ENOBUFS

Buffer space is not available to send the message.

ENOTCONN

The socket is not connected.

EOPNOTSUPP

The *socket* argument is associated with a socket that does not support one or more of the values set in *flags*.

EWouldBLOCK

socket is in nonblocking mode and no data buffers are available.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and *errno* is set to EOPNOTSUPP. In this case the message EDCV001I or EDCT002I is issued.

Stack characteristics

TCP/IP for VSE/ESA doesn't support the MSG_OOB and MSG_DONTROUTE options.

sendmsg() — Send Messages on a Socket**Format**

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <socket.h>

ssize_t sendmsg(int socket, struct msghdr *msg, int flags);
```

General Description

The `sendmsg()` call sends messages on a socket with descriptor *socket* passed in an array of message headers.

Parameter**Description**

socket The socket descriptor.

msg An array of message headers from which messages are sent.

flags The *flags* parameter is set by specifying 0, or one or more flags. If more than one flag is specified, the logical OR operator (|) must be used to separate them.

A message header is defined by the **msghdr** structure, which can be found in the **socket.h** include file and contains the following elements:

Element**Description**

msg_iov An array of *iovec* buffers containing the message.

msg_iovlen The number of elements in the *msg_iov* array.

sendmsg

msg_name
The pointer to the buffer containing the recipient's address.

msg_namelen
The size of the address buffer.

msg_control
Ancillary data, see below.

msg_controllen
Ancillary data buffer length.

msg_flags
Flags on received message.

Ancillary data consists of a sequence of pairs, each consisting of a **cmsghdr** structure followed by a data array. The data array contains the ancillary data message, and the **cmsghdr** structure contains descriptive information that allows an application to correctly parse the data.

The **socket.h** header file defines the **cmsghdr** structure that includes at least the following members:

Element	Description
---------	-------------

<i>cmsg_len</i>	Data byte count, including header.
-----------------	------------------------------------

<i>cmsg_level</i>	Originating protocol.
-------------------	-----------------------

<i>cmsg_type</i>	Protocol-specific type.
------------------	-------------------------

The **socket.h** header file defines the following macros to gain access to the data arrays in the ancillary data associated with a message header:

CMSG_DATA(*cmsg*)

If the argument is a pointer to a **cmsghdr** structure, this macro returns an unsigned character pointer to the data array associated with the **cmsghdr** structure.

CMSG_NXTHDR(*mhdr, cmsg*)

If the first argument is a pointer to a **msghdr** structure and the second argument is a pointer to a **cmsghdr** structure in the ancillary data, pointed to by the **msg_control** field of that **msghdr** structure, this macro returns a pointer to the next **cmsghdr** structure, or a null pointer if this structure is the last **cmsghdr** in the ancillary data.

CMSG_FIRSTHDR(*mhdr*)

If the argument is a pointer to a **msghdr** structure, this macro returns a pointer to the first **cmsghdr** structure in the ancillary data associated with this **msghdr** structure, or a null pointer if there is no ancillary data associated with the **msghdr** structure.

The **sendmsg()** call applies to sockets regardless of whether they are in the connected state.

This call returns the length of the data sent. If there is not enough available buffer space to hold the socket data to be transmitted, and the socket is in blocking mode,

sendmsg() blocks the caller until additional buffer space becomes available. If the socket is in nonblocking mode, sendmsg() returns a -1 and sets the error code to EWOULDBLOCK.

Socket Address Structure for IPv6 For an AF_INET6 socket, if msg_name is specified the address should be in a sockaddr_in6 address structure. The sockaddr_in6 structure is defined in the header file **in.h**.

Returned Value

If successful, the length of the message in bytes is returned. The value -1 indicates an error. The value of errno indicates the specific error.

A value of 0 or greater indicates the number of bytes sent, however, this does not assure that data delivery was complete.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and errno is set to EOPNOTSUPP. In this case the message EDCV001I or EDCT002I is issued.

sendto() — Send Data on a Socket

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <socket.h>

ssize_t sendto(int          socket,
               const void *msg,
               size_t      length,
               int         flags,
               const struct sockaddr *address,
               size_t      address_length);
```

General Description

The sendto() call sends data on the socket with descriptor *socket*. The sendto() call applies to either connected or unconnected sockets.

Parameter

Description

<i>socket</i>	The socket descriptor.
<i>msg</i>	The pointer to the buffer containing the message to transmit.
<i>length</i>	The length of the message in the buffer pointed to by the <i>msg</i> parameter. Unless the PTF for APAR PQ55591 is installed, the maximum number of bytes to be specified is 64K.
<i>flags</i>	reserved zero
<i>address</i>	The address of the target.
<i>address_length</i>	The size of the address pointed to by <i>address</i> .

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 additional buffer space becomes available. If the socket is in nonblocking mode, sendto() returns a -1 and sets the error code to EWOULDBLOCK. See “fcntl() —

Control Open Socket Descriptors” on page 104 or “ioctl() — Control Socket” on page 156 for a description of how to set nonblocking mode.

For datagram sockets, this call sends the entire datagram, provided that the datagram fits 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.

Socket address structure for IPv6: The `sockaddr_in6` structure is added to the `in.h` header. It is used to pass IPv6 specific addresses between applications and the system.

Returned Value

If successful, the number of characters sent is returned. The value -1 indicates an error. The value of *errno* indicates the specific error. No indication of failure to deliver is implied in the return value of this call when used with datagram sockets.

A value of 0 or greater indicates the number of bytes sent, however, this does not assure that data delivery was complete.

Error Code

Description

EAFNOSUPPORT

The address family is not supported (it is not AF_INET or AF_INET6).

EBADF

socket is not a valid socket descriptor.

ECONNRESET

A connection was forcibly closed by a peer.

EFAULT

Using the *msg* and *length* parameters would result in an attempt to access storage outside the caller's address space.

EINVAL

address_length is not the size of a valid address for the specified address family.

ENOBUFS

Buffer space is not available to send the message.

ENOTCONN

The socket is not connected.

EOPNOTSUPP

The *socket* argument is associated with a socket that does not support one or more of the values set in *flags*.

EWouldBlock

socket is in nonblocking mode and no data buffers are available.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1, and *errno* is set to EOPNOTSUPP. In this case the message EDCV001I or EDCT002I is issued.

sethostent() — Open the Host Information Data Set

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

void sethostent(int stayopen);
```

General Description

The `sethostent()` call opens and rewinds the data set which contains information about known hosts. If the `stayopen` flag is nonzero, the data set remains open after each call.

Returned Value

`sethostent()` returns no values.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the message EDCV001I or EDCT002I is issued.

setibmopt() — Set IBM TCP/IP Image

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <socket.h>

int setibmopt(int cmd, struct ibm_tcpimage *bfrp);
```

General Description

The `setibmopt()` function call is used to set TCP/IP options. Currently, the only supported command is `IBMTCP_IMAGE` which allows the `setibmopt()` to choose the active TCP/IP image stack the application will connect to.

To reset `ibm_tcpimage` to nothing chosen, set the `name` to all blanks.

Parameter

Description

cmd The value in *cmd* must be set to the command to be performed. Currently, only `IBMTCP_IMAGE` is supported and must be paired with the *bfrp* parameter as described.

bfrp The pointer to an `ibm_tcpimage` structure.

To set the TCP/IP image for a socket, the application should set values in the `ibm_tcpimage` structure as follows:

Element

Description

status 0 means is not known and need not be checked. Currently, this is the only value with meaning.

version

0 means the version is to be set on return if known.

name The name must be left justified, uppercase, padded with blanks, and be the name of an active TCP stack.

Returned Value

If successful, setibmopt() returns 0. If unsuccessful, setibmopt() returns -1 and sets errno to one of the following values:

Error Code

Description

EFAULT

Using the *bfrp* supplied would result in access of a storage location that is inaccessible.

EIBMBADTCPNAME

A name of a PFS was specified that either is not configured or is not a Sockets PFS.

EOPNOTSUPP

The *cmd* is a function that is not supported.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the message EDCV001I or EDCT002I is issued.

setnetent() — Open the Network Information Data Set

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

void setnetent(int stayopen);
```

General Description

The setnetent() call opens and rewinds the data set, which contains information about known networks. If the *stayopen* flag is nonzero, the data set remains open after each call to setnetent().

Returned Value

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the message EDCV001I or EDCT002I is issued.

setprotoent() — Open the Protocol Information Data Set

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

void setprotoent(int stayopen);
```

General Description

The setprotoent() call opens and rewinds the data set which contains information about known protocols. If the *stayopen* flag is nonzero, the data set remains open after each call.

Returned Value

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the message EDCV001I or EDCT002I is issued.

setservent() — Open the Network Services Information Data Set

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

void setservent(int stayopen);
```

General Description

The `setservent()` call opens and rewinds the data set which contains information about known services. If the *stayopen* flag is nonzero, the data set remains open after each call to `setservent()`.

Returned Value

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the message EDCV001I or EDCT002I is issued.

setsockopt() — Set Options Associated with a Socket

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <socket.h>

int setsockopt(int socket,
               int level,
               int option_name,
               const void *option_value,
               size_t option_length);
```

IPv6: To include support for IPv6 socket options, add the following code:

```
#define _OPEN_SYS_SOCKET_IPV6 1
#include <in.h>
```

General Description

The `setsockopt()` call sets options associated with a socket. Options can exist at multiple protocol levels; they are always present at the highest socket level.

Parameter

Description

socket The socket descriptor.

level The level for which the option is being set.

option_name
The name of a specified socket option.

option_value
The pointer to option data.

option_length
The length of the option data.

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

setsockopt

options at the IPv4 or IPv6 level, the level parameter must be set to IPPROTO_IP as defined in `socket.h` or IPPROTO_IPV6 as defined in `in.h`.

The *option_value* and *option_length* parameters are used to pass data used by the particular set command. The *option_value* parameter points to a buffer containing the data needed by the set command. The *option_value* parameter is optional and can be set to the NULL pointer, if data is not needed by the command. The *option_length* parameter must be set to the size of the data pointed to by *option_value*.

All of the socket-level options except SO_LINGER, SO_RCVTIMEO and SO_SNDTIMEO expect *option_value* to point to an integer and *option_length* to be set to the size of an integer. When the integer is nonzero, the option is enabled. When it is zero, the option is disabled. The SO_LINGER option expects *option_value* to point to a **linger** structure, as defined in `socket.h`. This structure is defined in the following example:

```
struct linger
{
    int    l_onoff;           /* option on/off */
    int    l_linger;        /* linger time */
};
```

The *l_onoff* field is set to 0, if the SO_LINGER option is disabled. A nonzero value enables the option. The *l_linger* field specifies the amount of time to linger on close. The units of *l_linger* are seconds.

The following options are recognized at the socket level:

Option Description

SO_LINGER

Lingers on close if data is present. If this option is enabled and there is unsent data present when `close()` is called, the calling application program is blocked during the `close()` call, until the data is transmitted or the connection has timed out. If this option is disabled, the TCP/IP address space 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 only a finite amount of time trying to send the data. The `close()` call returns without blocking the caller. This option has meaning only for stream sockets.

SO_KEEPALIVE

This option is provided for source compatibility reasons only. It will not perform any action, but the user should instead use the common TCP/IP setting : SET PULSE_TIME=nnn. This TCP/IP option has the same effect on the entire TCP/IP partition as SO_KEEPALIVE is supposed to have for a single TCP connection.

SO_REUSEADDR

This option is provided for source compatibility reasons only. It will not perform any action, but TCP/IP implicitly allows for immediate address reuse.

The following options are recognized at the IPv4 level:

Option Description

IP_ADD_MEMBERSHIP

This option is used 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. Applications that only transmit do not need to do so.

The multicast IP address and the interface IP address are passed in the following structure available in **in.h**:

```
struct ip_mreq
{
    struct in_addr imr_multiaddr; /* IP multicast addr of group */
    struct in_addr imr_interface; /* local IP addr of interface */
};
```

If **INADDR_ANY** is specified on the interface address of the **mreq** structure passed, a default interface will be chosen as follows:

- If the group address specified in the **mreq** structure was specified on a **GATEWAY** statement, use that interface.
- If 224.0.0.0 was specified on **GATEWAY** statement, use that interface.
- If **DEFAULTNET** was specified and is multicast capable, use that interface.

IP_ADD_SOURCE_MEMBERSHIP

This option is used to join a source-specific multicast group specified by the **ip_mreq_source** structure. The **ip_mreq_source** structure is defined in **in.h**.

IP_BLOCK_SOURCE

This option is used to block from a given source to a given multicast group (e.g., if the user "mutes" that source). The source multicast group is specified by the **ip_mreq_source** structure which is defined in **in.h**.

IP_DROP_MEMBERSHIP

This option is used to leave a multicast group.

The multicast IP address and the interface IP address are passed in the following structure available in **in.h**:

```
struct ip_mreq
{
    struct in_addr imr_multiaddr; /* IP multicast addr of group */
    struct in_addr imr_interface; /* local IP addr of interface */
};
```

If **INADDR_ANY** is specified on the interface address of the **mreq** structure passed, the system will drop the first group that matches the group (class D) address without regard to the interface.

IP_DROP_SOURCE_MEMBERSHIP

This option is used to leave a source-specific multicast group specified by the **ip_mreq_source** structure. The **ip_mreq_source** structure is defined in **in.h**.

IP_MULTICAST_IF

Sets the interface for sending outbound multicast datagrams from this socket application. Multicast datagrams are transmitted only on one interface at a time. An IP address is passed using **struct in_addr**.

If **INADDR_ANY** is specified for the interface address passed, a default interface is chosen as follows:

- If 224.0.0.0 was specified on **GATEWAY** statement, use that interface.
- If **DEFAULTNET** was specified and is multicast capable, use that interface.

IP_MULTICAST_LOOP

Enables/disables loopback of outgoing multicast datagrams. Default is enable. If it is enabled, multicast applications that have joined the outgoing multicast group can receive a copy of the multicast datagrams destined for that address/port pair. The loopback indicator is passed in as `u_char`. 0 is specified to disable loopback. 1 is specified to enable loopback.

IP_MULTICAST_TTL

Sets the IP time-to-live of outgoing multicast datagrams. The default value is 1 (that is, multicast only to the local subnet). The TTL value is passed in as `u_char`.

IP_UNBLOCK_SOURCE

This option is used to undo the operation performed with the `IP_BLOCK_SOURCE` option (e.g., if the user "mutes" that source). The source group is specified by the `ip_mreq_source` structure which is defined in `in.h`.

MCAST_BLOCK_SOURCE

This option is used to block data from a given source to a given group (for example, if the user "mutes" that source). The source is specified by the `group_source_req` structure which is defined in `in.h`.

MCAST_JOIN_GROUP

This option is used to join an any-source group. The group is specified by the `group_req` structure. The `group_req` structure is defined in `in.h`.

MCAST_JOIN_SOURCE_GROUP

This option is used to join a source-specific group. The source is specified by the `group_source_req` structure which is defined in `in.h`.

MCAST_LEAVE_GROUP

This option is used to leave an any-source group. The group is specified by the `group_req` structure. The `group_req` structure is defined in `in.h`.

MCAST_LEAVE_SOURCE_GROUP

This option is used to leave a source-specific group. The source is specified by the `group_source_req` structure which is defined in `in.h`.

MCAST_UNBLOCK_SOURCE

This option is used to undo the operation performed with the `MCAST_BLOCK_SOURCE` option (for example, if the user then "unmutes" the source). The source is specified by the `group_source_req` structure which is defined in `in.h`.

The following options are recognized at IPv6 level:

Option Description**IPV6_JOIN_GROUP**

Controls the receipt of multicast packets by joining the multicast group specified by the `ipv6_mreq` structure that is passed. The `ipv6_mreq` structure is defined in `in.h`.

IPV6_LEAVE_GROUP

Controls the receipt of multicast packets by leaving the multicast group specified by the `ipv6_mreq` structure that is passed. The `ipv6_mreq` structure is defined in `in.h`.

IPV6_MULTICAST_HOPS

Sets the hop limit for outgoing multicast packets. The hop limit value is passed in as an `int`.

IPV6_MULTICAST_IF

Sets the interface for outgoing multicast packets. An interface index is used to specify the interface. It is passed in as a `u_int`.

IPV6_MULTICAST_LOOP

If a multicast datagram is sent to a group to which the sending host itself belongs (on the outgoing interface), a copy of the datagram is looped back by the IP layer for local delivery, if this option is set to one. If this option is set to zero, a copy is not looped back. Other option values return an error of `EINVAL`. The default is one (loopback). The option value is passed in as an `int`.

IPV6_UNICAST_HOPS

Controls the hop limit in outgoing unicast IPv6 packets. The hop limit value is passed in as an `int`.

IPV6_V6ONLY

Determines whether a socket is restricted to IPv6 communications only. The default setting is off. The option value is passed in as an `int`. A nonzero value means the option is enabled (socket can only be used for IPv6 communications). 0 means the option is disabled.

Note: To use these options, you must use the feature test macro `#define _OPEN_SYS_SOCK_IPV6`.

Returned Value

The value 0 indicates success; the value -1 indicates an error. The value of the error code indicates the specific error.

Error Code**Description****EBADF**

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

EFAULT

Using *option_value* and *option_length* parameters would result in an attempt to access storage outside the caller's address space.

EINVAL

The specified option is invalid at the specified socket level or the socket has been shut down.

ENOBUFS

Insufficient system resources are available to complete the call.

ENOPROTOOPT

The *option_name* parameter is unrecognized, or the *level* parameter is not `SOL_SOCKET`.

ENOSYS

The function is not implemented. You attempted to use a function that is not yet available.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and `errno` is set to `ENOSYS`. In this case the message `EDCV001I` or `EDCT002I` is issued.

Stack characteristics

TCP/IP for VSE/ESA supports option `SO_LINGER` only. Emulation support for `SO_KEEPALIVE` and `SO_REUSEADDR` is granted too.

- `SO_KEEPALIVE`

Support for this option is provided for source code compatibility reasons only. Indeed, setting a keep alive value has no effect on the TCP connection. Instead the user should use the `SET PULSE_TIME` TCP/IP setting which manages the keep-alive mechanism for the owning TCP/IP partition, rather than for a single connection only.

- `SO_REUSEADDR`

This option is used to allow for immediate local address reuse. TCP/IP always allows for immediate reuse, therefore this socket is provided for compatibility reasons only. There is no way to disable socket reuse.

shutdown()

The `shutdown()` options `SHUT_RD` and `SHUT_WR` to shut down a particular end of a duplex connection are not supported by TCP/IP for VSE/ESA. Only `SHUT_RDWR` is supported to shut down both ends. Further, while on other platforms after a call to `shutdown()` the socket descriptor remains valid, TCP/IP for VSE/ESA acts as if a call to `close()` has also been issued. Calling `close()` after `shutdown()` by the application therefore would cause error `EBADF`. For compatibility reasons the TCP/IP support for the LE socket API remembers the pending close request after the call to `shutdown()` and doesn't raise the `EBADF` error code. However, if a new call to `socket()` was issued between calling `shutdown()` and `close()` the socket descriptor may have been reused by the TCP/IP stack already. This is true for the CICS runtime environment especially, where another transaction outside the program's control may have allocated a socket already. For compatibility reasons and to allow for portability it is therefore not recommended to close a socket by using `shutdown()`, but `close()` should be used instead. The call to `shutdown()` should be avoided all together.

socket()

TCP/IP for VSE/ESA supports TCP and UDP connections in the `AF_INET` domain, i.e. only the `IPPROTO_TCP` and `IPPROTO_UDP` protocol options are supported. `IPPROTO_IP` (numeric value 0) causes special processing. According to the socket type, the matching protocol is automatically chosen.

- `SOCK_DGRAM` causes protocol `IPPROTO_UDP` to be chosen.
- `SOCK_STREAM` causes protocol `IPPROTO_TCP` to be chosen.

Sockets of type `SOCK_RAW` are not supported by TCP/IP for VSE/ESA.

Example

The following are examples of the `setsockopt()` call. See “`getsockopt()` — Get the Options Associated with a Socket” on page 128 for examples of how the `getsockopt()` options set are queried.

```
#include <socket.h>

int rc;
int s;
int option_value;
struct linger l;
:
/* I want to linger on close */
```

```

l.l_onoff = 1;
l.l_linger = 100;
rc = setsockopt(s, SOL_SOCKET, SO_LINGER, &l, sizeof(l));

```

shutdown() — Shut Down a Connection

Format

```

#define _XOPEN_SOURCE_EXTENDED 1
#include <socket.h>

```

```

long shutdown(int socket, int how);

```

General Description

The `shutdown()` call shuts down a connection.

Parameter

	Description
<i>socket</i>	The socket descriptor.
<i>how</i>	The condition of the shutdown. <i>how</i> can have a value of : <ul style="list-style-type: none"> • SHUT_RD which ends communication from the socket indicated by <i>socket</i>. • SHUT_WR which ends communication to the socket indicated by <i>socket</i>. • SHUT_RDWR which ends communication both to and from socket indicated by <i>socket</i>.

Returned Value

The value 0 indicates success; the value -1 indicates an error. The value of the error code indicates the specific error.

Error Code

	Description
EBADF	<i>socket</i> is not a valid socket descriptor.
EINVAL	The <i>how</i> parameter was not set to one of the valid values.
ENOBUFS	Insufficient system resources are available to complete the call.
ENOTCONN	The socket is not connected.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and `errno` is set to `ENOTCONN`. In this case the message `EDCV001I` or `EDCT002I` is issued.

Stack characteristics

TCP/IP for VSE/ESA does not support the `shutdown()` options `SHUT_RD` and `SHUT_WR` that are used to shut down a particular end of a duplex connection. Only `SHUT_RDWR` is supported to shut down both ends. Further, while on other platforms after a call to `shutdown()` the socket descriptor remains valid, TCP/IP for VSE/ESA acts as if a call to `close()` has also been issued. Calling `close()` after

shutdown

shutdown() by the application therefore would cause error EBADF. For compatibility reasons the TCP/IP support for the LE socket API remembers the pending close request after the call to shutdown() and doesn't raise the EBADF error code. However, if a new call to socket() was issued between calling shutdown() and close() the socket descriptor may have been reused by the TCP/IP stack already. This is true for the CICS runtime environment especially, where another transaction outside the program's control may have allocated a socket already. For compatibility reasons and to allow for portability it is therefore not recommended to close a socket by using shutdown(), but close() should be used instead. The call to shutdown() should be avoided all together.

socket()

TCP/IP for VSE/ESA supports TCP and UDP connections in the AF_INET domain, i.e. only the IPPROTO_TCP and IPPROTO_UDP protocol options are supported. IPPROTO_IP (numeric value 0) causes special processing. According to the socket type, the matching protocol is automatically chosen.

- SOCK_DGRAM causes protocol IPPROTO_UDP to be chosen.
- SOCK_STREAM causes protocol IPPROTO_TCP to be chosen.

Sockets of type SOCK_RAW are not supported by TCP/IP for VSE/ESA.

socket() — Create a Socket

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <socket.h>
#include <in.h>

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

General Description

The socket() call creates an endpoint for communication and returns a socket descriptor representing the endpoint. Different types of sockets provide different communication services.

Parameter

Description

domain The address domain requested, either AF_INET or AF_INET6.

type The type of socket created, either SOCK_STREAM or SOCK_DGRAM.

protocol

The protocol requested. Some possible values are 0, IPPROTO_UDP, or IPPROTO_TCP.

The *domain* parameter specifies a communication domain within which communication is to take place. This parameter selects the address family (format of addresses within a domain) that is used. The families supported are AF_INET or AF_INET6, which is the Internet domain. This constant is defined in the **socket.h** include file.

The *type* parameter specifies the type of socket created. The type is analogous with the semantics of the communication requested. These socket type constants are defined in the **socket.h** include file. The types supported are:

Socket Type

Description

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 multiple times. This type is supported in the AF_INET or AF_INET6 domains.

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 the AF_INET or AF_INET6 domains.

Note: RAW sockets are not supported.

Understanding the socket() Parameters

The *protocol* parameter specifies a particular protocol to be used with the socket. In most cases, a single protocol exists to support a particular type of socket in a particular address family. If the *protocol* parameter is set to 0, the system selects the default protocol number for the domain and socket type requested. The `getprotobyname()` call can be used to get the protocol number for a protocol with a known name.

SOCK_STREAM sockets model duplex-byte streams. They provide reliable, flow-controlled connections between peer application programs. Stream sockets are either active or passive. Active sockets are used by clients who start connection requests with `connect()`. By default, `socket()` creates active sockets. Passive sockets are used by servers to accept connection requests with the `connect()` call. You can transform an active socket into a passive socket by binding a name to the socket with the `bind()` call and by indicating a willingness to accept connections with the `listen()` call. After a socket is passive, it cannot be used to start connection requests.

In the AF_INET and or AF_INET6 domains, the `bind()` call applied to a stream socket lets the application program specify the networks from which it is willing to accept connection requests. The application program 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 program can use a *wildcard* to specify that it wants to receive connection requests from any network. For AF_INET sockets, this is done by setting the *Internet address* field in the **address** structure to the constant INADDR_ANY, as defined in **in.h**. For AF_INET6 sockets, this is done by setting the *Internet address* field in the **address** structure to IN6ADDR_ANY as defined in **in.h**.

After a connection has been established between stream sockets, any of the data transfer calls can be used: (`read()`, `readv()`, `recv()`, `recvfrom()`, `send()`, `sendto()`, `write()`, and `writv()`). Usually, the `read()`-`write()` or `send()`-`recv()` pairs are used for sending 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 guarantees of reliability. Messages sent have a maximum size.

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 from which network interfaces it wishes to receive packets. Wildcard addressing, as described for stream

socket()

sockets, applies for 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 program has received a datagram socket, it can exchange datagrams using the `sendto()` and `recvfrom()`, or `sendmsg()` and `recvmsg()`, calls. If the application program goes one step further by calling `connect()` and fully specifying the name of the peer with which all messages will be exchanged, the other data transfer calls `read()`, `write()`, `readv()`, `writew()`, `send()`, and `recv()` can also be used. For more information on placing a socket into the connected state, see “`connect()` — Connect a Socket” on page 100.

Datagram sockets allow messages to be broadcast to multiple recipients. Setting the destination address to be a broadcast address is network-interface-dependent (it depends on the class of address and whether *subnets*—logical networks divided into smaller physical networks to simplify routing—are used).

Sockets are deallocated with the `close()` call.

Returned Value

A nonnegative socket descriptor indicates success. The value -1 indicates an error. The value of the error code indicates the specific error.

Error Code

Description

EAFNOSUPPORT

The address family is not supported (it is not `AF_INET` or `AF_INET6`).

EINVAL

The request is invalid or not supported.

ENOBUFS

Insufficient system resources are available to complete the call.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and `errno` is set to `EINVAL`. In this case the message `EDCV001I` or `EDCT002I` is issued.

Stack characteristics

TCP/IP for VSE/ESA supports TCP and UDP connections in the `AF_INET` domain, only the `IPPROTO_TCP` and `IPPROTO_UDP` protocol options are supported. `IPPROTO_IP` (numeric value 0) causes special processing. According to the socket type, the matching protocol is automatically chosen.

- `SOCK_DGRAM` causes protocol `IPPROTO_UDP` to be chosen.
- `SOCK_STREAM` causes protocol `IPPROTO_TCP` to be chosen.

Sockets of type `SOCK_RAW` are not supported by TCP/IP for VSE/ESA.

Example

The following is an example of the `socket()` call.

```

int s;
char *name;
:
:
/* Get stream socket in Internet domain with default protocol */
s = socket (AF_INET, SOCK_STREAM, 0);
:
:

```

socketpair() — Create a Pair of Sockets

Format

```

#define _XOPEN_SOURCE_EXTENDED 1
#include <socket.h>

int socketpair(int *domain, int type, int protocol, int sv[2]);

```

General Description

The `socketpair()` call acquires a pair of sockets of the type specified that are unnamed and connected in the specified domain and using the specified protocol.

Parameter

Description

domain The domain in which to open the socket.

type The type of socket created.

protocol

The protocol requested.

sv The descriptors used to refer to the obtained sockets.

Returned Value

A nonnegative socket descriptor indicates success. The value -1 indicates an error. The value of `errno` indicates the specific error.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and `errno` is set to `EINVAL`. In this case the message `EDCV001I` or `EDCT002I` is issued.

takesocket() — Acquire a Socket from Another Program

Format

```

#define _OPEN_SYS_SOCK_EXT
#include <types.h>
#include <socket.h>

int takesocket(struct clientid *clientid, int sdesc);

```

General Description

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's startup parameter list.

Parameter

Description

clientid A pointer to the *clientid* of the application from which you are taking a socket.

takesocket

sdesc The descriptor of the socket to be taken.

If the *c_reserved.type* field of the *clientid* structure was set to *SO_CLOSE* on the *givesocket()* call, *c_close.SockToken* of *clientid* structure should be used as input to *takesocket()*, instead of the normal socket descriptor. See “*givesocket() — Make the Specified Socket Available*” on page 133 for a description of the *clientid* structure.

Returned Value

The value -1 indicates an error. The value of *errno* indicates the specific error. If not -1, the return value is the new socket descriptor.

Error Code

Description

EBADF

The *sdesc* parameter does not specify a valid socket descriptor owned by the other application, or the socket has already been taken.

EFAULT

Using the *clientid* parameter as specified would result in an attempt to access storage outside the caller's partition.

EINVAL

The *clientid* parameter does not specify a valid client identifier. Either the client process cannot be found, or the client exists but has no outstanding givesockets.

ENFILE

The socket descriptor table is already full.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and *errno* is set to *EINVAL*. In this case the message *EDCV001I* or *EDCT002I* is issued.

termapi() — Terminate the Socket API for a Subtask

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <socket.h>
```

```
void termapi(void);
```

General Description

The *termapi()* function terminates the socket API for a subtask.

Returned Value

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the message *EDCV001I* or *EDCT002I* is issued.

write() — Write Data on a Socket

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

ssize_t write(int fs, const void *buf, ssize_t N);
```

General Description

The `write()` call writes data from a buffer on a socket with descriptor `fs`. The `write()` call can only be used with connected sockets. This call writes up to `N` bytes of data.

`write()` is equivalent to `send()` with no flags set.

Parameter

Description

<i>socket</i>	The socket descriptor.
<i>buf</i>	The pointer to the buffer holding the data to be written.
<i>N</i>	The length in bytes of the buffer pointed to by the <i>buf</i> parameter. Unless the PTF for APAR PQ55591 is installed, the maximum number of bytes to be specified is 64K.

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 additional buffer space becomes available. If the socket is in nonblocking mode, `write()` returns a -1 and sets the error code to EWOULDBLOCK. See “`fcntl()` — Control Open Socket Descriptors” on page 104 or “`ioctl()` — Control Socket” on page 156 for a description of how to set the nonblocking mode.

If the socket is not ready to accept data and the process is trying to write data to the socket:

- Unless `O_NDELAY` is set, `write()` blocks until the socket is ready to accept data.
- If `O_NDELAY` is set, `write()` returns a 0.

For datagram sockets, this call sends the entire datagram, provided that the datagram fits into the TCP/IP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application program wishes to send 1000 bytes, each call to this function can send 1 byte or 10 bytes or the entire 1000 bytes. Therefore, application programs using stream sockets should place this call in a loop, calling this function until all data has been sent.

Returned Value

If successful, `write()` returns the number of bytes actually written, less than or equal to `N`. If unsuccessful, it returns the value -1 and sets `errno` to one of the following:

A value of 0 or greater indicates the number of bytes sent. However, this does not assure that data delivery was complete.

EBADF

fs is not a valid socket descriptor.

write

ECONNRESET

A connection was forcibly closed by a peer.

EDESTADDRREQ

The socket is not connection-oriented and no peer address is set.

EFAULT

Using the *buf* and *N* parameters would result in an attempt to access storage outside the caller's address space.

EINVAL

The request is invalid or not supported.

EIO

An I/O error occurred.

ENOBUFS

Buffer space is not available to send the message.

ENOTCONN

The socket is not connected.

EWouldBlock

The socket is in nonblocking mode and data is not available to write.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and *errno* is set to *EINVAL*. In this case the message *EDCV001I* or *EDCT002I* is issued.

Example

The following are examples of the `write()` call.

```
#include <stdio.h>
#include <string.h>

/*Write the zero terminated string aString to the socket aSocket and
print number of bytes written. Return number of bytes written or -1
for no success.
*/
int writeToSocket(int aSocket, char* aString)
{ int numberOfBytesWritten;

  numberOfBytesWritten=
    write(aSocket, aString, strlen(aString));
  if (numberOfBytesWritten < 0)
  { perror("write"); return -1; }
  else
  { printf("number of bytes written is %d.\n", numberOfBytesWritten);
    return numberOfBytesWritten;
  }
}
```

writev() — Write Data on a Socket from an Array

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <uio.h>

ssize_t writev(int fs, const struct iovec *iov, int iovcnt);
```

General Description

The `writev()` call writes data to a socket with descriptor *fs* from a set of buffers. The data is gathered from the buffers specified by *iov[0]...iov[iovcnt-1]*. The descriptor must refer to a connected socket.

Parameter

	Description
<i>fs</i>	The socket descriptor.
<i>iov</i>	A pointer to an array of iovec buffers.
<i>iovcnt</i>	The number of buffers pointed to by the <i>iov</i> parameter.

The **iovec** structure is defined in **uio.h** and contains the following fields:

Element

	Description
<i>iov_base</i>	Pointer to the buffer.
<i>iov_length</i>	Length of the buffer.

This call writes the sum of the *iov_length* 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, `writev()` blocks the caller until additional buffer space becomes available. If the socket is in a nonblocking mode, `writev()` returns a -1 and sets the error code to `EWOULDBLOCK`.

Returned Value

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

A value of 0 or greater indicates the number of bytes sent, however, this does not assure that data delivery was complete.

If there is no TCP/IP product installed or if the TCP/IP product has not implemented this specific function, the corresponding dummy routine in C Run-Time always returns the value -1 and `errno` is set to `EINVAL`. In this case the message `EDCV001I` or `EDCT002I` is issued.

writev

Chapter 11. Using the CALL Instruction Application Programming Interface (EZASOKET API)

This chapter describes the CALL Instruction API for TCP/IP Application programs and includes the following topics:

- Environmental Restrictions and Programming Requirements
- CALL instruction API
- Understanding COBOL, Assembler, and PL/I call formats

Environmental Restrictions and Programming Requirements

The following restrictions apply to the Callable Socket API:

- The EZASOKET API cannot be used with programs running in an ICCF Pseudo Partition.
- 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.
- Addressability mode (AMODE) considerations
The EZASOKET Call API must be invoked while the caller is in 31-bit AMODE.
- When using the EZASOKET CALL API in CICS transactions while CICS operates with storage protection, all programs using the CALL API need to be defined with EXECKEY(CICS). This is also true for those programs that link to these programs. TASKDATAKEY(CICS) for the transaction definition is NOT required.
- When using the CALL API in CICS transactions, the EZA "task-related-user-exit" (TRUE) has to be activated before these transactions can be run. For details on how to activate this TRUE, please refer to "CICS Considerations for the EZA Interfaces" on page 83.

CALL Instruction Application Programming Interface (API)

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

Note:

1. Reentrant code is supported by this interface.
2. For a PL/I program, include the following statement before your first call instruction.

```
DCL EZASOKET ENTRY OPTIONS(RETCODE,ASM,INTER) EXT;
```

3. Register conventions:

Register 0, 1, 14, and 15 are used by the interface and must be, if necessary, saved prior to invocation.

Register 13 must point to a 72-byte save area provided by the caller.

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

```
▶▶—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.

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—▶▶
```

You can use the following call format for reentrant programming.

```
▶▶—CALL EZASOKET,(SOC-FUNCTION,—parm1, parm2, ...—ERRNO, RETCODE),VL,MF=(E,list-addr)—▶▶
```

PL/I Language Call Format

```
▶▶—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.

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 COBOL VSE language syntax and conventions, but you should use the syntax and conventions that are appropriate for the language you want to use.

Figure 16 shows examples of storage definition statements for COBOL, PL/I, and assembler language programs.

```

COBOL PIC

PIC S9(4) COMP           HALFWORD BINARY VALUE
PIC S9(8) COMP           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 16. Storage Definition Statement Examples

Error Messages and Return Codes

For information about error messages, refer to *z/VSE Messages and Codes, Volume 1* and *TCP/IP for VSE 1.5 Messages and Codes*.

For information about error codes that are returned by TCP/IP, refer to “ERRNO Values” on page 74.

Debugging

See Appendix B, “Debugging Facility for EZASMI and EZASOKET Interfaces (EZAAPI Trace),” on page 525.

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

ACCEPT

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.

Note:

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. If 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. If 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. If 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. 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.

Important: IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply, if this program is used.

Example in COBOL

```
WORKING STORAGE
 01 SOC-FUNCTION    PIC X(16) VALUE IS 'ACCEPT'.
 01 S              PIC 9(4) BINARY.
 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).
 01 ERRNO         PIC 9(8) BINARY.
 01 RETCODE       PIC S9(8) BINARY.

PROCEDURE
  CALL 'EZASOKET' USING SOC-FUNCTION S NAME ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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

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 `PRD1.MACLIB(EZBREHST)` 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 TCP/IP the value is a decimal 2, indicating `AF_INET`.

PORT A halfword binary field specifying the client's port number.

IPv4-ADDRESS

A fullword binary field specifying the 32-bit IPv4 Internet address.

RESERVED

Specifies eight bytes of binary zeros. 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 that specifies the length of the IPv6 socket address structure. Any value specified by the use of this field is ignored, if processed as input. The field is set to 0, if processed as output.

FAMILY

A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating `AF_INET6`.

PORT A halfword binary 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 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 "ERRNO Values" on page 74 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.

BIND

In a typical server program, the BIND call follows a SOCKET call and completes the process of creating a new socket.

The BIND call can either specify the required port or let the system choose the port. A listener program should always bind to the same well-known port, so that clients know what socket address to use when attempting to connect.

In the AF_INET domain, the BIND call for a stream socket can specify the networks from which it is willing to accept connection requests. The application can fully specify the network interface by setting the ADDRESS field 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 interface. This is done by setting the ADDRESS field to a fullword of zeros.

Important: IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply, if this program is used.

Example in COBOL

```

WORKING STORAGE
  01 SOC-FUNCTION    PIC X(16) VALUE IS 'BIND'.
  01 S               PIC 9(4) BINARY.
  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).
  01 ERRNO          PIC 9(8) BINARY.
  01 RETCODE        PIC S9(8) BINARY.

PROCEDURE
  CALL 'EZASOKET' USING SOC-FUNCTION S NAME ERRNO RETCODE.

```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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

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 PRD1.MACLIB(EZBREHST) 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 TCP/IP the value is a decimal 2, indicating AF_INET.

PORT A halfword binary field specifying the client's port number. If you set the port number to zero, TCP/IP assigns the port. The application can call the GETSOCKNAME macro after the BIND macro to discover the assigned port.

IPv4-ADDRESS

A fullword binary field specifying the 32-bit IPv4 Internet address.

RESERVED

Specifies eight bytes of binary zeros. 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 that specifies the length of the IPv6 socket address structure. Any value specified by the use of this field is ignored, if processed as input. The field is set to 0, if processed as output.

FAMILY

A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary field that is set to the client port number. If you set the port number to zero, TCP/IP assigns the port. The application can call the GETSOCKNAME macro after the BIND macro to discover the assigned port.

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 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.

Parameter Values Returned to the Application

ERRNO

A fullword binary field. If RETCODE is negative, this field contains an error number. See "ERRNO Values" on page 74, 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.

Note:

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 270.
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. If a child server is created, the concurrent server obtains a new socket, passes the new socket to the child server, and 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.

Example in COBOL

```

WORKING STORAGE
  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.
  
```

```
CALL 'EZASOKET' USING SOC-FUNCTION S ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see "Converting Parameter Descriptions" on page 201.

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 "ERRNO Values" on page 74 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:

1. It completes the binding process for a stream socket if a BIND call has not been previously issued.
2. 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 zero RETCODE indicates that the connection was completed.

CONNECT

- A nonzero RETCODE with an ERRNO 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 261.

Important: IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply, if this program is used.

Example in COBOL

```
WORKING STORAGE
01 SOC-FUNCTION    PIC X(16) VALUE IS 'CONNECT'.
01 S              PIC 9(4) BINARY.
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).
01 ERRNO         PIC 9(8) BINARY.
01 RETCODE       PIC S9(8) BINARY.
```

```
CALL 'EZASOKET' USING SOC-FUNCTION S NAME ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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

Input parameter. The NAME parameter for CONNECT specifies the IPv4 or IPv6 socket address of the target to which the local, client socket is to be connected. Include the PRD1.MACLIB(EZBREHST) 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 TCP/IP the value is a decimal 2, indicating AF_INET.

PORT A halfword binary field specifying the client's port number.

IPv4-ADDRESS

A fullword binary field specifying the 32-bit IPv4 Internet address.

RESERVED

Specifies eight bytes of binary zeros. 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 that specifies the length of the IPv6 socket address structure. Any value specified by the use of this field is ignored, if processed as input. The field is set to 0, if processed as output.

FAMILY

A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary 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 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.

Parameter Values Returned to the Application**ERRNO**

A fullword binary field. If RETCODE is negative, this field contains an error number. See "ERRNO Values" on page 74 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 249 for another way to control a socket's blocking mode.

The following example shows an FCNTL call instruction:

Example in COBOL

```

WORKING STORAGE
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
CALL 'EZASOKET' USING SOC-FUNCTION S COMMAND REQARG
ERRNO RETCODE.

```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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 “ERRNO Values” on page 74 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.

Important: This function call is not available with TCP/IP for VSE/ESA.

The following example shows a FREEADDRINFO call instruction:

Example in COBOL

```
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.
```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

Parameter Values Set by the Application

SOC-FUNCTION

A 16-byte character field containing FREEADDRINFO. The field is left justified and padded on the right with blanks.

ADDRINFO

The address of a set of address information structures returned by the GETADDRINFO RES argument.

Parameter Values Returned to the Application

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, ERRNO contains a valid error number. Otherwise, ignore the ERRNO field. See “ERRNO Values” on page 74 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.

GETADDRINFO

The GETADDRINFO call translates the name of a service location (for example, a host name), a service name, or both, into a set of socket addresses and other associated information.

This information can be used to create a socket and connect to, or to send a datagram to, the specified service.

GETADDRINFO

Important: This function call is not available with TCP/IP for VSE/ESA.

Example in COBOL

```
WORKING-STORAGE SECTION.  
    01 SOC-FUNCTION PIC X(16) VALUE IS 'GETADDRINFO'.  
    01 NODE PIC X(255).  
    01 NODELEN PIC 9(8) BINARY.  
    01 SERVIC 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 X(4).  
        03 FILLER PIC X(4).  
        03 FILLER PIC 9(8) BINARY.  
        03 FILLER PIC X(4).  
        03 FILLER PIC 9(8) BINARY.  
        03 FILLER PIC X(4).  
        03 FILLER PIC 9(8) BINARY.  
    01 RES-ADDRINFO.  
        03 FLAGS PIC 9(8) BINARY.  
        03 AF PIC 9(8) BINARY.  
        03 SOCTYPE PIC 9(8) BINARY.  
        03 PROTO PIC 9(8) BINARY.  
        03 NAMELEN PIC 9(8) BINARY.  
        03 FILLER PIC X(4).  
        03 FILLER PIC X(4).  
        03 CANONNAME USAGE IS POINTER.  
        03 FILLER PIC X(4).  
        03 NAME USAGE IS POINTER.  
        03 FILLER PIC X(4).  
        03 NEXTP USAGE IS POINTER.  
  
PROCEDURE DIVISION.  
    MOVE 'www.hostname.com' TO NODE.  
    MOVE 16 TO NODELEN.  
    MOVE 'TELNET' TO SERVIC.  
    MOVE 6 TO SERVLN.  
    SET HINTS TO ADDRESS OF HINTS-ADDRINFO.  
    CALL 'EZASOKET' USING SOC-FUNCTION NODE NODELEN SERVIC SERVLN HINTS RES CANNLEN  
    ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

Parameter Values Set by the Application

SOC-FUNCTION

A 16-byte character field containing GETADDRINFO. The field is left justified and padded on the right with blanks.

NODE

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, NODE should contain the queried host IP address in network byte order presentation form. This is an optional field, but if specified you must also code NODELEN. The NODE name being queried consists of up to NODELEN or up to the first binary zero. You can append scope information to the host name by using the format *node%scope information*. The combined information must be 255 bytes or less.

NODELEN

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.

SERVIC

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, SERVIC should contain the queried port number in presentation form. This is an optional field, but if specified you must also code SERVLLEN. The SERVIC name being queried consists of up to SERVLLEN or up to the first binary zero.

SERVLLEN

A fullword binary field set to the length of the service name specified in the SERVIC field and should not include extraneous blanks. This is an optional field but if specified you must also code SERVIC.

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, 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. This is an optional field.

The address information structure has the following fields:

Value 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') or a decimal value of 1.

Specifies how to fill in the NAME pointed to by the returned RES.

If this flag is specified, the returned address information is 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, the IP address portion of the socket address structure pointed to by the returned RES will be set to

GETADDRINFO

INADDR_ANY for an IPv4 address or to the IPv6 unspecified address (in6addr_any) for an IPv6 address.

If this flag is not set, the returned address information will be suitable for the CONNECT call (for a connection-mode protocol) or for a CONNECT, SENDTO, or SENDMSG call (for a connectionless protocol). In this case, if the NODE argument is not specified, 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 or the default loopback address for an IPv6 address.

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, 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, 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 SERVIC 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. If the AI-ALL flag is not also specified and 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.

If the AF field has a value of AF_INET6 and AI-ALL is set, the AI-V4MAPPED flag must also be set to indicate that the caller will accept all addresses (IPv6 and IPv4-mapped IPv6 addresses). If 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, 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, 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.

Note: To perform the binary ORing 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 ORing 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-ADDRCONFIG 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), 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_SOCTYPE. Note that although SOCK_RAW will be accepted, it is only valid if SERVIC is numeric (for example, SERVIC=23). A lookup for a SERVIC name will never occur in the appropriate services file using any protocol value other than SOCK_STREAM or SOCK_DGRAM.

If PROTO is not 0 and SOCTYPE is 0, 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.

GETADDRINFO

If SOCTYPE and PROTO are both specified as 0, GETADDRINFO will proceed as follows:

- If SERVIC is null, or if SERVIC is numeric, any returned address information will default to a specification of SOCK_STREAM as SOCK_STREAM.
- If SERVIC is specified as a service name (for example, SERVIC=FTP), the GETADDRINFO call will search the appropriate services file 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, they should be compatible, regardless of the value specified by SERVIC. 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, GETADDRINFO will proceed as follows:

- If SERVIC is null, or if SERVIC is numeric, any returned address information will default to a specification of SOCK_STREAM as SOCK_STREAM.
- If SERVIC is specified as a service name (for example, SERVIC=FTP), the GETADDRINFO will search the appropriate services file 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 SERVIC. 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 `SERVIC` fails [for example, the service name does not appear in an appropriate service file using the input protocol], the `GETADDRINFO` call will fail with return code of `EAI_SERVICE`.

NAMELEN

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

CANNONNAME

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.

Note:

- `FLAGS` can be specified with their corresponding decimal value.
- To perform the binary ORing of the `FLAGS` 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 `X'00000050'`, which is the sum of ORing `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-ADDRCONFIG TO FLAGS.
```

RES Initially a fullword binary field. On a successful return, this field contains a pointer to a chain of one or more address information structures. Use the `EZBREHST` (from `PRD1.MACLIB`) macro to establish address information mapping. The structures are allocated in the key of the calling application. Do not use or reference these structures between tasks. When you are finished using the structures, explicitly free their storage by specifying the returned pointer on a `TYPE=FREEADDRINFO` call; storage that is not explicitly freed is released when the task is ended.

CANNLEN

Initially an input parameter. A fullword binary field used to contain the length of the canonical name returned by the `RES` `CANNONNAME` field. This is an optional field.

Parameter Values Returned to the Application

ERRNO

A fullword binary field. If `RETCODE` is negative, `ERRNO` contains a valid error number. Otherwise, ignore the `ERRNO` field. See “`ERRNO` Values” on page 74 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.

The `ADDRINFO` structure uses indirect addressing to return a variable number of `NAMES`. If you are coding in PL/I or assembler language, this structure can be processed in a relatively straight-forward manner. If you are coding in COBOL,

GETADDRINFO

this structure might be difficult to interpret. You can use the subroutine "EZACIC09" on page 286 to simplify interpretation of the information returned by the GETADDRINFO calls.

GETCLIENTID

The 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" on page 234 and "TAKESOCKET" on page 276 calls.

If GETCLIENTID is called by a server, the identifier of the *caller* (not necessarily the *client*) is returned.

Important: IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply, if this program is used.

Example in COBOL

```
WORKING STORAGE
  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
  CALL 'EZASOKET' USING SOC-FUNCTION CLIENT ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see "Converting Parameter Descriptions" on page 201.

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

A fullword binary number specifying the domain of the client. On input, this is an optional parameter for AF_INET, and a required parameter for AF_INET6 to specify the domain of the client. For TCP/IP, the value is a decimal 2 indicating AF_INET, or decimal 19 indicating AF_INET6. On output, this is the returned domain of the client.

NAME

An 8-byte character field. It is built with the partition's partition ID, which is left adjusted and padded with blanks.

TASK An 8-byte character field. This task identifier can be specified by the user with the INITAPI call or defaulted by the system (see the description of the INITAPI call for details).

RESERVED

Specifies 20-byte character reserved field. This field is required and internally used by TCP/IP.

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See "ERRNO Values" on page 74 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 internet address is specified in the call.

A given TCP/IP host can have multiple alias names and multiple host internet addresses.

Example in COBOL

```

WORKING STORAGE
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
CALL 'EZASOKET' USING SOC-FUNCTION HOSTADDR HOSTENT RETCODE.

```

For equivalent PL/I and assembler language declarations, see "Converting Parameter Descriptions" on page 201.

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 "ERRNO Values" on page 74 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	An error occurred

GETHOSTBYADDR

GETHOSTBYADDR returns the HOSTENT structure shown in Figure 17.

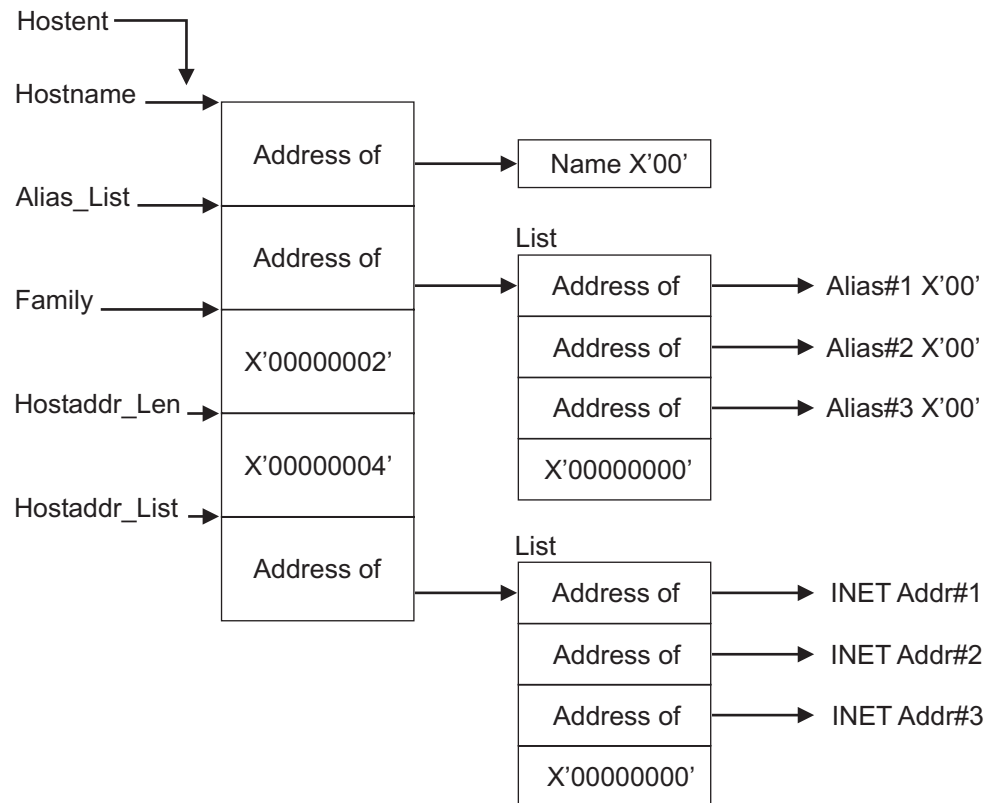


Figure 17. HOSTENT Structure Returned by the GETHOSTBYADDR Call

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'.

Important: ALIAS names are not supported with TCP/IP for VSE/ESA.

- 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'

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 284.

GETHOSTBYNAME

The GETHOSTBYNAME call returns the alias name and the 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 internet addresses.

TCP/IP tries to resolve the host name through a name server, if one is present. If a call is made to convert a symbolic name to an IP address, TCP/IP for VSE/ESA searches the local names table (created by DEFINE NAME) first. If this search fails, the name is passed to the specified DNSs (set with SET DNSx). TCP/IP for VSE/ESA will try each DNS, beginning with DNS1, until a response is received or all servers have been polled. The first server to respond determines if the request succeeds or fails. If the search within a DNS fails, the default domain string (as specified with SET DEFAULT_DOMAIN) is appended to the name (following a period) and the DNS is consulted the last time for the name resolution.

Example in COBOL

```

WORKING STORAGE
  01 SOC-FUNCTION    PIC X(16) VALUE IS 'GETHOSTBYNAME'.
  01 NAMELEN        PIC 9(8)  BINARY.
  01 NAME           PIC X(24).
  01 HOSTENT        PIC 9(8)  BINARY.
  01 RETCODE        PIC S9(8) BINARY.

PROCEDURE
  CALL 'EZASOKET' USING SOC-FUNCTION NAMELEN NAME
                    HOSTENT RETCODE.

```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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.

NAME

A character string, up to 24 characters, set to a host name. 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

GETHOSTBYNAME

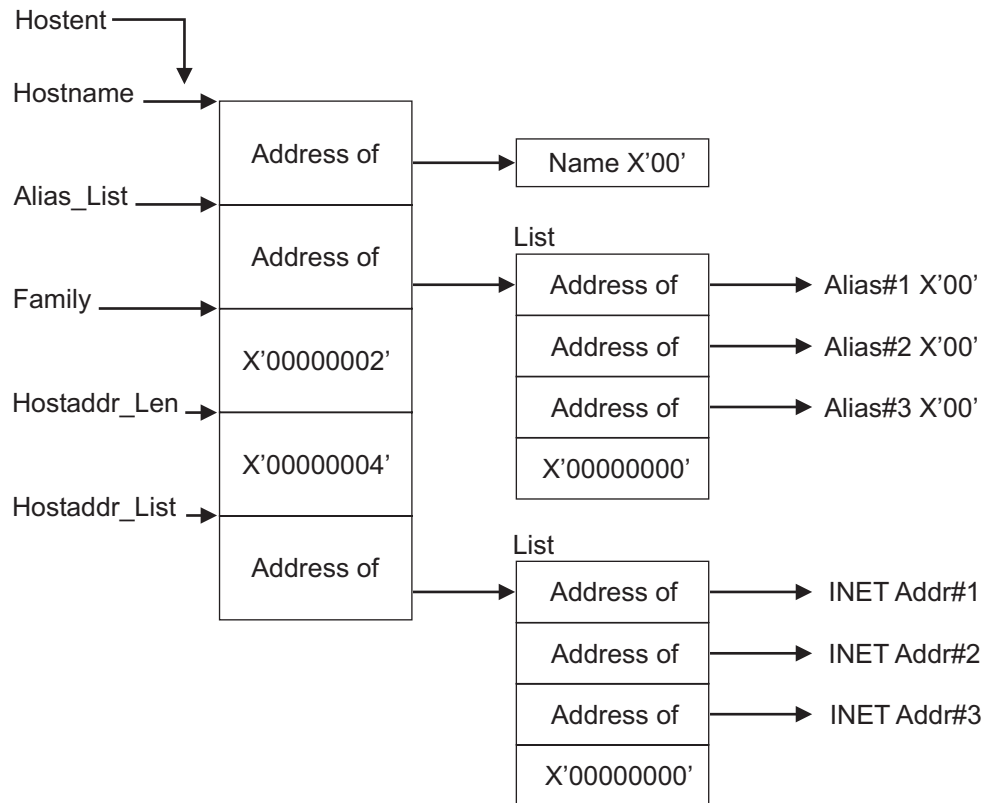


Figure 18. HOSTENT Structure Returned by the GETHOSTBYNAME Call

GETHOSTBYNAME returns the HOSTENT structure shown in Figure 18. 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'.

Important: Alias names are not supported with TCP/IP for VSE/ESA.

- 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'.

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 284.

GETHOSTID

The GETHOSTID call returns the 32-bit internet address for the current host.

Example in COBOL

```

WORKING STORAGE
  01 SOC-FUNCTION    PIC X(16) VALUE IS 'GETHOSTID'.
  01 RETCODE        PIC S9(8) BINARY.

PROCEDURE
  CALL 'EZASOKET' USING SOC-FUNCTION RETCODE.

```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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. A *-1* in RETCODE indicates an error. A possible reason can be that TCP/IP has not been started. There is no ERRNO parameter for this call.

GETHOSTNAME

The GETHOSTNAME call returns the domain name of the local host.

Example in COBOL

```

WORKING STORAGE
  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
  CALL 'EZASOKET' USING SOC-FUNCTION NAMELEN NAME
                    ERRNO RETCODE.

```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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**NAMELEN**

A fullword binary field set to the length of the host name.

NAME

Indicates the receiving field for the host name. TCP/IP for VSE/ESA allows a maximum length of 64-characters. The internet standard is a maximum name length of 255 characters. The actual length of the NAME field is found in NAMELEN.

GETHOSTNAME

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See “ERRNO Values” on page 74 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 z/VSE system and their status, versions, and names.

With this information, the caller can dynamically choose the TCP/IP image with which to connect by using the INITAPI call.

Important: This function call is not available with TCP/IP for VSE/ESA.

Example in COBOL

```
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.
```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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.

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.

GETNAMEINFO

The GETNAMEINFO call 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.

Important: This function call is not available with TCP/IP for VSE/ESA.

Example in COBOL

```
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 SERVIC PIC X(32).
  01 SERVLN PIC 9(8) BINARY.
  01 FLAGS PIC 9(8) BINARY VALUE 0.
  01 NI-NOFQDN PIC 9(8) BINARY VALUE 1.
  01 NI-NUMERICHOST PIC 9(8) BINARY VALUE 2.
  01 NI-NAMEREQD PIC 9(8) BINARY VALUE 4.
  01 NI-NUMERICSERVER PIC 9(8) BINARY VALUE 8.
  01 NI-DGRAM PIC 9(8) BINARY VALUE 16.
  01 NI-NUMERICSCOPE PIC 9(8) BINARY VALUE 32.
```

```
* IPv4 socket structure.
  01 NAME.
    03 FAMILY PIC 9(4) BINARY.
    03 PORT PIC 9(4) BINARY.
    03 IP-ADDRESS PIC 9(8) BINARY.
    03 RESERVED PIC X(8).
```

```
* IPv6 socket structure.
  01 NAME.
    03 FAMILY PIC 9(4) BINARY.
    03 PORT PIC 9(4) BINARY.
    03 FLOWINFO PIC 9(8) BINARY.
    03 IP-ADDRESS.
      10 FILLER PIC 9(16) BINARY.
      10 FILLER PIC 9(16) BINARY.
    03 SCOPE-ID PIC 9(8) BINARY.
  01 ERRNO PIC 9(8) BINARY.
  01 RETCODE PIC S9(8) BINARY.
```

```
PROCEDURE DIVISION.
```

GETNAMEINFO

```
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 SERVIC SERVLN FLAGS ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

Parameter Values Set by the Application

SOC-FUNCTION

A 16-byte character field containing GETNAMEINFO. The field is left justified and padded on the right with blanks.

NAME

An IPv4 or IPv6 socket address structure to be translated. Include the PRD1.MACLIB(EZBREHST) macro to get the assembler mappings for the socket address structure. The socket address structure mappings start at the SOCKADDR label.

The AF_INET socket address structure fields start at the SOCK_SIN label. The AF_INET6 socket address structure fields start at the SOCK_SIN6 label. The IPv4 socket address structure must specify the following fields:

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

FAMILY

A halfword binary number specifying the IPv4 addressing family. For TCP/IP the value is a decimal 2, indicating AF_INET.

PORT A halfword binary number specifying the port number.

IPv4-ADDRESS

A fullword binary number specifying the 32-bit IPv4 Internet address.

RESERVED

An 8-byte reserved field. This field is required, but is not used.

The IPv6 socket address structure specifies the following fields:

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

NAMELEN

A 1-byte binary field that specifies the length of the IPv6 socket address structure. Any value specified by the use of this field is ignored, if processed as input. The field is set to 0, if processed as output.

FAMILY

A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary number that specifies the port number.

FLOW-INFO

This field is ignored by the TYPE=GETNAMEINFO macro.

IPv6-ADDRESS

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

SCOPE-ID

A fullword binary field that specifies the scope for an IPv6 address as an interface index. The resolver ignores the SCOPE_ID field, unless the address in IPv6-ADDRESS is a link-local address and the HOST parameter is also specified.

NAMELEN

A fullword binary field. The length of the socket address structure pointed to by the NAME argument.

HOST On input, storage capable of holding the returned resolved host name, which can be up to 255 bytes long, for the input socket address. If inadequate storage is specified to contain the resolved host name, the resolver returns the host name up to the storage specified and truncation might occur. If the host's name cannot be located, the numeric form of the host's address is returned instead of its name. However, if the NI_NAMEREQD option is specified and no host name is located, an error is returned. One or both of the following groups of parameters are required:

- The HOST and HOSTLEN parameters
- The SERVIC and SERVLLEN parameters

Otherwise, an error occurs. The HOST name being queried consists of up to HOSTLEN or up to the first binary 0.

If the IPv6-ADDRESS value is a link-local address, and the SCOPE_ID interface index is nonzero, scope information is appended to the resolved host name using the format *host%scope information*. The scope information can be the numeric form of the SCOPE_ID interface index or the interface name that is associated with the SCOPE_ID interface index. Use the NI_NUMERICSCOPE option to select which form is returned. The combined host name and scope information is 255 bytes or less.

HOSTLEN

A fullword binary field that contains the length of the host storage that is used to contain the returned resolved host name. If HOSTLEN is 0 on input, the resolved host name is not returned. The HOSTLEN value must be equal to or greater than the length of the longest host name, or hostname and scope information combination, to be returned. The TYPE=GETNAMEINFO returns the host name, or host name and scope information combination, up to the length specified by the HOSTLEN value. On output, HOSTLEN contains the length of the returned resolved host name, or host name and scope information combination. This is an optional field, but if you specify this field, you also must code the HOST value. One or both of the following groups of parameters are required:

- The HOST and HOSTLEN parameters
- The SERVIC and SERVLLEN parameters

Otherwise, an error occurs.

SERVIC

On input, storage capable of holding the returned resolved service name, which can be up to 32 bytes long, for the input socket address. If inadequate storage is specified to contain the resolved service name, the resolver returns the service name up to the storage specified and truncation might occur. If the service name cannot be located, or if NI_NUMERICSERV was specified in the FLAGS operand, the presentation form of the service address is returned instead of its name. This is an

GETNAMEINFO

optional field, but if you specify this field, you also must code the `SERVLEN` parameter. The `SERVIC` name being queried consists of up to `SERVLEN` or up to the first binary zero. One or both of the following groups of parameters are required:

- The `HOST` and `HOSTLEN` parameters
- The `SERVIC` and `SERVLEN` parameters

Otherwise, an error occurs.

SERVLEN

Initially an input parameter. A fullword binary field that contains the length of the `SERVIC` storage used to contain the returned resolved service name. If `SERVLEN` is 0 on input, the service name information is not returned. `SERVLEN` must be equal to or greater than the length of the longest service name to be returned. The `TYPE=GETNAMEINFO` returns the service name up to the length specified by `SERVLEN`. On output, `SERVLEN` contains the length of the returned resolved service name. This is an optional field, but if you specify it, you must also code the `SERVIC` parameter. One or both of the following groups of parameters are required:

- The `HOST` and `HOSTLEN` parameters
- The `SERVIC` and `SERVLEN` parameters

Otherwise, an error occurs.

FLAGS

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.
'NI_NUMERICSCOPE'	X'00000020'	32	Only return the numeric form of the SCOPE-ID interface index, if applicable.

Parameter Values Returned to the Application

ERRNO

A fullword binary field. If `RETCODE` is negative, the field contains an error number. See "ERRNO Values" on page 74 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

GETPEERNAME

The GETPEERNAME call returns the name of the remote socket to which the local socket is connected.

Important: IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply, if this program is used.

Example in COBOL

```

WORKING STORAGE
01 SOC-FUNCTION    PIC X(16) VALUE IS 'GETPEERNAME'.
01 S              PIC 9(4) BINARY.
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).
01 ERRNO         PIC 9(8) BINARY.
01 RETCODE       PIC S9(8) BINARY.

PROCEDURE
CALL 'EZASOKET' USING SOC-FUNCTION S NAME ERRNO RETCODE.

```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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

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 PRD1.MACLIB(EZBREHST) 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 TCP/IP the value is a decimal 2, indicating AF_INET.

PORT A halfword binary field specifying the client's port number.

IPv4-ADDRESS

A fullword binary field specifying the 32-bit IPv4 Internet address.

GETPEERNAME

RESERVED

Specifies eight bytes of binary zeros. 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 that specifies the length of the IPv6 socket address structure. Any value specified by the use of this field is ignored, if processed as input. The field is set to 0, if processed as output.

FAMILY

A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary 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 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 "ERRNO Values" on page 74 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.

Important: IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply, if this program is used.

A stream socket is not assigned a name until after a successful call to either BIND, CONNECT, or ACCEPT, therefore the GETSOCKNAME call can be used after an implicit bind to discover which port was assigned to the socket.

Example in COBOL

```

WORKING STORAGE
01 SOC-FUNCTION    PIC X(16) VALUE IS 'GETSOCKNAME'.
01 S               PIC 9(4) BINARY.
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).
01 ERRNO          PIC 9(8) BINARY.
01 RETCODE        PIC S9(8) BINARY.

```

```

PROCEDURE
  CALL 'EZASOKET' USING SOC-FUNCTION S NAME ERRNO RETCODE.

```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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 local socket whose address is required.

Parameter Values Returned to the Application**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 PRD1.MACLIB(EZBREHST) 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 TCP/IP the value is a decimal 2, indicating AF_INET.

PORT A halfword binary field specifying the port number bound to this socket. If the socket is not bound, a zero is returned.

IPv4-ADDRESS

A fullword binary field specifying the 32-bit IPv4 Internet address.

RESERVED

Specifies eight bytes of binary zeros. 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 that specifies the length of the IPv6 socket

GETSOCKNAME

address structure. Any value specified by the use of this field is ignored, if processed as input. The field is set to 0, if processed as output.

FAMILY

A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary field specifying the port number bound to this socket. If the socket is not bound, a zero is returned.

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 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 "ERRNO Values" on page 74 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" on page 270 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.

Example in COBOL

```
WORKING STORAGE
  01 SOC-FUNCTION    PIC X(16) VALUE IS 'GETSOCKOPT'.
  01 S              PIC 9(4) BINARY.
  01 OPTNAME        PIC 9(8) BINARY.
  88 SO-LINGER      VALUE 128.
  01 OPTVAL         PIC X(16) BINARY.
  01 OPTLEN         PIC 9(8) BINARY.
  01 ERRNO          PIC 9(8) BINARY.
  01 RETCODE        PIC S9(8) BINARY.
```

PROCEDURE

```
CALL 'EZASOKET' USING SOC-FUNCTION S OPTNAME
                        OPTVAL OPTLEN ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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. The options are as follows:

IP_MULTICAST_IF

Use this option to 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.

IP_MULTICAST_LOOP

Use this option to 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.

IP_MULTICAST_TTL

Use this option to 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.

IPV6_MULTICAST_HOPS

Use this option to obtain the hop limit used for outgoing multicast packets. This is an IPv6-only socket option.

IPV6_MULTICAST_IF

Use this option to obtain the index of the IPv6 interface used for sending outbound multicast datagrams from the socket application. This is an IPv6-only socket option.

IPV6_MULTICAST_LOOP

Use this option to determine whether a multicast datagram is looped back on the outgoing interface by the IP layer for local delivery, if 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.

IPV6_UNICAST_HOPS

Use this option to obtain the hop limit used for outgoing unicast IPv6 packets. This is an IPv6-only socket option.

IPV6_V6ONLY

Use this option to 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.

SO-LINGER

Requests the status of LINGER.

- If the LINGER option has been enabled, and data transmission has not been completed, a CLOSE call blocks the calling program until the data is transmitted or until the connection has timed out.
- If LINGER is not enabled, a CLOSE call returns without blocking the caller. TCP/IP attempts to send the data; although the data transfer is usually successful, it cannot be guaranteed, because TCP/IP only attempts to send the data for a specified amount of time.

Parameter Values Returned to the Application

OPTVAL

- If SO-LINGER is specified in OPTNAME, the following structure is returned:

ONOFF	PIC X(8)
LINGER	PIC 9(8)

- A nonzero value returned in ONOFF indicates that the option is enabled; a zero value indicates that it is disabled.
- The LINGER value indicates the amount of time (in seconds) TCP/IP will continue to attempt to send the data after the CLOSE call is issued. To *set* the Linger time, see “SETSOCKOPT” on page 270.

OPTLEN

A fullword binary field containing the length of the data returned in OPTVAL.

- For OPTNAME of SO-LINGER, OPTVAL contains two fullwords, so OPTLEN will be set to 8 (two fullwords).

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See “ERRNO Values” on page 74 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 |

GIVESOCKET

The GIVESOCKET call is used to pass a socket from one process to another.

TCP/IP normally uses GETCLIENTID, GIVESOCKET, and TAKESOCKET calls in the following sequence:

1. A process issues a GETCLIENTID call to get the jobname of its region and its VSE 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 which 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.

Important: IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply, if this program is used.

Example in COBOL

```

WORKING STORAGE
  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
  CALL 'EZASOKET' USING SOC-FUNCTION S CLIENT ERRNO RETCODE.

```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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 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

Specifies an 8-character field, left-justified, padded to the right with blanks set to the address space name of the application (partition ID) going to take the socket. If this field is left blank, any z/VSE partition can take the socket.

GIVESOCKET

TASK Specifies an eight-character field that can be set to blanks, or to the identifier of the socket-taking VSE subtask. If this field is set to blanks, any subtask in the partition specified in the NAME field can take the socket.

RESERVED

A 20-byte reserved field. This field is required, but only used internally.

Parameter Values Returned to the Application

ERRNO

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

See "ERRNO Values" on page 74, 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
----	-------------------------------

GSKFREEMEM

The GSKFREEMEM call frees memory passed to the application on a previous call to an SSL function.

Note: The distinguished name returned in the null-terminated string by the GSKGETDNBYLAB call must be freed using GSKFREEMEM.

Example in COBOL

```
WORKING-STORAGE SECTION.  
  01 SOC-FUNCTION PIC X(16) VALUE 'GSKFREEMEM    '.  
  01 AREA          PIC 9(8) BINARY.  
  01 ERRNO         PIC 9(8) BINARY.  
  01 RETCODE       PIC S9(8) BINARY.  
  
PROCEDURE DIVISION.  
  CALL 'EZASOKET' USING SOC-FUNCTION AREA  
                        ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see "Converting Parameter Descriptions" on page 201.

Parameter Values Set by the Application

SOC-FUNCTION

A 16-byte character field containing 'GSKFREEMEM'. The field is left-justified and padded on the right with blanks.

AREA

Parameter Values Returned to the Application

ERRNO

A fullword binary field. May show detailed error information.

RETCODE

A fullword binary field that returns one of the following

0 Successful call.

less than 0
An error occurred.

GSKGETCIPHINF

The GSKGETCIPHINF call requests cipher related information for SSL for VSE.

This information determines the encryption level that the system can support and returns a list of cipher specifications that SSL can use. This allows an application to determine, at run time, the level of SSL encryption that the installed application can request.

Example in COBOL

```

WORKING-STORAGE SECTION.
    01 SOC-FUNCTION PIC X(16) VALUE 'GSKGETCIPHINF  '.
    01 CIPHLEVEL   PIC 9(8) BINARY.
    01 SECLEVEL    PIC X(104).
    01 ERRNO       PIC 9(8) BINARY.
    01 RETCODE     PIC S9(8) BINARY.

PROCEDURE DIVISION.
    CALL 'EZASOKET' USING SOC-FUNCTION CIPHLEVEL SECLEVEL
                        ERRNO RETCODE.

```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

Parameter Values Set by the Application**SOC-FUNCTION**

A 16-byte character field containing 'GSKGETCIPHINF'. The field is left-justified and padded on the right with blanks.

CIPHLEVEL

A fullword binary field with a number that determines the type of cipher information to be returned. Valid values are

- 1** only exportable cipher information is to be returned (GSK_LOW_SECURITY)
- 2** exportable and domestic cipher information is to be returned (GSK_HIGH_SECURITY)

Parameter Values Returned to the Application**SECLEVEL**

A 104 byte area where the system returns the following information:

4 bytes

System SSL version (always 3 for GSK_VERSION3)

64 bytes

A character string (terminated with x00) with the SSL Version 3 cipher specs allowed for use on the system (these are passable on the V3CIPHER parameter on the GSKSSOCINIT call).

GSKGETCIPHINF

32 bytes

This field will always be filled with binary zeros because SSL for VSE does not support SSL Version 2 cipher specs.

4 bytes

One of the following

- 1 GSK_SEC_LEVEL_US
- 2 GSK_SEC_LEVEL_EXPORT
- 3 GSK_SEC_LEVEL_EXPORT_FR

ERRNO

A fullword binary field. May show detailed error information.

RETCODE

A fullword binary field that returns one of the following

- 0 Successful call.

less than 0

An error occurred. Please refer to VSE library member SSLVSE.A or to the *TCP/IP for VSE 1.5 Optional Features* for a detailed description of error return codes.

GSKGETDNBYLAB

The GSKGETDNBYLAB call returns the complete distinguished name for a key based on the label the key has in the key database file.

This value can be used for the DNAME field in the GSKSSOCINIT call.

Note: The distinguished name returned in the null-terminated string must be freed using the GSKFREEMEM call.

Example in COBOL

```
WORKING-STORAGE SECTION.  
  01 SOC-FUNCTION PIC X(16) VALUE 'GSKGETDNBYLAB  '.  
  01 KEYLABEL     PIC X(Length of key label).  
  01 ERRNO       PIC 9(8) BINARY.  
  01 RETCODE     PIC S9(8) BINARY.  
  
PROCEDURE DIVISION.  
  CALL 'EZASOKET' USING SOC-FUNCTION KEYLABEL  
                        ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

Parameter Values Set by the Application

SOC-FUNCTION

A 16-byte character field containing 'GSKGETDNBYLAB'. The field is left-justified and padded on the right with blanks.

KEYLABEL

Parameter Values Returned to the Application

ERRNO

A fullword binary field. May show detailed error information.

RETCODE

A fullword binary field that returns one of the following

greater 0

Successful call. RETCODE denotes a pointer to character string with the distinguished name.

0 or less than 0

Unsuccessful call.

GSKINIT

The GSKINIT call sets the overall SSL for VSE environment for the current partition.

After the function completes successfully, the application is ready to call SSL for VSE interfaces and to create and use secure socket connections.

Example in COBOL

```

WORKING-STORAGE SECTION.
  01 SOC-FUNCTION PIC X(16) VALUE 'GSKINIT      '.
  01 SECTYPE.
    05 SECTYPE1 PIC X(5) VALUE IS 'SSL30'.
    05 SECTPYE2 PIC 9(1) BINARY VALUE 0.
  01 KEYRING.
    05 KEYRING1 PIC X(11) VALUE IS 'PRIMARY.GSK'.
    05 KEYRING2 PIC 9(1) COMP VALUE 0.
  01 V3TIMEOUT PIC 9(8) COMP VALUE 86400.
  01 CAROOTS PIC 9(8) COMP VALUE 0.
  01 AUTHTYPE PIC 9(8) COMP VALUE 0.
  01 ERRNO PIC 9(8) BINARY.
  01 RETCODE PIC S9(8) BINARY.

PROCEDURE DIVISION.
  CALL 'EZASOKET' USING SOC-FUNCTION SECTYPE KEYRING
                      V3TIMEOUT CAROOTS AUTHTYPE
                      ERRNO RETCODE.

```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

Parameter Values Set by the Application**SOC-FUNCTION**

A 16-byte character field containing 'GSKINIT'. The field is left-justified and padded on the right with blanks.

SECTYPE

A character string that identifies the minimum acceptable security protocol. The value must be entered in upper case characters and terminated with a X00. Valid values are (without double-quotes):

- "SSL30" for SSL Version 3.0
- "TLS31" for TLS Version 1.0 (not supported for client applications)

KEYRING

A character string specifying the "lib.sublib" where the private key and certificates are stored. The string must be terminated with x00. Provide a string of 8 blanks, if you want to use the default "SSL for VSE" files as defined in procedure \$SSL4VSE.PROC. Refer to *TCP/IP for VSE 1.5 Optional Features* for details on this procedure.

V3TIMEOUT

The number of seconds for the SSL V3 session Identifier to expire. The valid range is 0 to 86400 (1 day). If this parameter is not specified, a default value of 86400 is applied.

CAROOTS

A value that specifies which CA (Certificate Authority) root to use for certificate verification. The supported values are:

- 0 Use the CA roots from the local key database file for certificate verification.
- 1 Allow client authentication with certificates issued by the same certificate authority as VSE.

AUTHTYPE

A value that specifies the method to use for verifying the client's certificate. This field is only used, if CAROOTS is set to 1. The supported values are:

- 0 the client's certificate is verified using the local key database file.
- 1 the same meaning as with value 0
- 2 the same meaning as with value 0
- 3 the client's certificate is not verified.

Parameter Values Returned to the Application**ERRNO**

A fullword binary field. May show detailed error information.

RETCODE

A fullword binary field that returns one of the following

- 0 Successful call.

not equal 0

An error occurred. Please refer to VSE library member SSLVSE.A or to the *TCP/IP for VSE 1.5 Optional Features* for a detailed description of error return codes.

GSKSSOCCLOSE

The GSKSSOCCLOSE call ends a secure socket connection and frees all SSL for VSE resources for that connection.

Example in COBOL

```

WORKING-STORAGE SECTION.
  01 SOC-FUNCTION PIC X(16) VALUE 'GSKSSOCCLOSE  '.
  01 SSOCDATA     PIC 9(8) BINARY.
  01 ERRNO        PIC 9(8) BINARY.
  01 RETCODE      PIC S9(8) BINARY.

PROCEDURE DIVISION.
  CALL 'EZASOKET' USING SOC-FUNCTION SSOCDATA
                        ERRNO RETCODE.

```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

Parameter Values Set by the Application

SOC-FUNCTION

A 16-byte character field containing 'GSKSSOCCLOSE'. The field is left-justified and padded on the right with blanks.

SSOCDATA

Address of GSKSOCDATA structure as returned in RETCODE by the GSKSSOCINIT call.

Parameter Values Returned to the Application

ERRNO

A fullword binary field. May show detailed error information.

RETCODE

A fullword binary field that returns one of the following

0 Successful call.

less than 0.

An error occurred. Please refer to VSE library member SSLVSE.A or to the *TCP/IP for VSE 1.5 Optional Features* for a detailed description of error return codes.

GSKSSOCINIT

The GSKSSOCINIT call initializes the data areas for SSL for VSE to initiate or accept a secure socket connection.

After the function is completed successfully, a pointer to a secured socket control block (in the following referred to as GSKSOCDATA) is returned to the application. Other calls using this secure socket connection must refer to this pointer.

During the call a complete handshake is performed based on the input specified with the GSKSSOCINIT call. While SSL for VSE performs the mechanics of the SSL handshake, "normal" RECV and SEND routines (provided by the EZAAPI processing environment) will be used to transport the SSL data during the SSL handshake, as well as for all subsequent read/write operations.

Example in COBOL

```

WORKING-STORAGE SECTION.
  01 SOC-FUNCTION PIC X(16) VALUE 'GSKSSOCINIT' .
  01 S PIC 9(4) BINARY.
  01 HANDSHAKE PIC 9(8) BINARY.
  01 DNAME.
    05 DNAME1 PIC X(n) VALUE IS '.....'.
    05 DNAME2 PIC 9(1) BINARY VALUE 0.
  01 V3CIPHER.
    05 V3CIPHER1 PIC X(6) VALUE IS '0A0908'.
    05 V3CIPHER2 PIC 9(1) COMP VALUE 0.
  01 SECTYPE USAGE IS POINTER.
  01 V3CIPHERSEL PIC X(2).
  01 CERTINFO USAGE IS POINTER.
  01 REASCODE PIC 9(8) BINARY.
  01 ERRNO PIC 9(8) BINARY.
  01 RETCODE PIC S9(8) BINARY.

PROCEDURE DIVISION.
  CALL 'EZASOKET' USING SOC-FUNCTION S HANDSHAKE DNAME
    SECTYPE V3CIPHER V3CIPHERSEL CERTINFO REASCODE
    ERRNO RETCODE.

```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

Parameter Values Set by the Application

SOC-FUNCTION

A 16-byte character field containing 'GSKSSOCINIT'. The field is left-justified and padded on the right with blanks.

S A halfword binary field with the descriptor of the socket that is going to be used for a secure socket connection.

HANDSHAKE

A halfword binary number that specifies how the handshake is performed:

- 0** Perform the SSL handshake as a client (GSK_AS_CLIENT).
- 1** Perform the SSL handshake as a server (GSK_AS_SERVER).
- 2** Perform the SSL handshake as a server that requires client authentication (GSK_AS_SERVER_WITH_CLIENT_AUTH).
- 3** Perform the SSL handshake as a client without authentication (GSK_AS_CLIENT_NO_AUTH).

DNAME

A character that is the Distinguished name or label of the desired entry (certificate) in the key database file. This character string must be terminated with x00. To use the default key database file entry, point to a string of 8 blanks. The distinguished name for a key database file entry may be determined via the EZASOCKET GETDNBYLAB function call.

V3CIPHER

A character string that contains the list of SSL Version 3.0 ciphers in order of usage preference. Valid values as supported by TCP/IP for VSE are:

- 01 for RSA512_NULL_MD5
- 02 for RSA512_NULL_SHA
- 08 for RSA512_DES40CBC_SHA
- 09 for RSA1024_DESCBC_SHA
- 0A for RSA1024_3DESCBC_SHA
- 62 for RSA1024_EXPORT_DESCBC_SHA

You can use any combination of these values in any order. The list of values must be terminated with x00. The exportable cipher suites 01,02,08,62 can only be used with SSL30, and will not work with TLS1.0. To use the default SSL V3 cipher specs (which is 0A0908) specify a string of 8 blanks.

Parameter Values Returned to the Application

SECTYPE

A fullword binary field where the address of a character string is stored that identifies the minimum acceptable security protocol. The character string is terminated with x00. Valid values are (without double-quotes):

- "SSL30" for SSL Version 3.0
- "TLS31" for TLS Version 1.0

V3CIPHSEL

A 2-byte area (provided by the application) where the architected SSL Version 3.0 cipher spec value selected for this session is stored (for example: x0009).

CERTINFO

A fullword binary field where the address of the Distinguished Name components from the client's certificate is stored. This parameter is only valid, if client authentication is requested for a server using SSL. The layout of this area is as follows:

- 4 bytes**
Pointer to base64 certificate body
- 4 bytes**
Length of base64 certificate body
- 4 bytes**
Pointer to session ID for this connection
- 4 bytes**
Flag to indicate if new session
- 4 bytes**
Pointer to certificate serial number
- 4 bytes**
Pointer to common name of client
- 4 bytes**
Pointer to locality
- 4 bytes**
Pointer to state or province
- 4 bytes**
Pointer to country
- 4 bytes**
Pointer to organization
- 4 bytes**
Pointer to organizational unit
- 4 bytes**
Pointer to issuer's common name
- 4 bytes**
Pointer to issuer's locality
- 4 bytes**
Pointer to issuer's state or province
- 4 bytes**
Pointer to issuer's country
- 4 bytes**
Pointer to issuer's organization
- 4 bytes**
Pointer to issuer's organizational unit

GSKSSOCINIT

REASCODE

A fullword binary field where the failure reason code for the GSKSSOCINIT call is stored. A value of 0 indicates the successful completion of the function.

ERRNO

A fullword binary field. May show detailed error information.

RETCODE

If REASCODE is 0, the RETCODE parameter contains the pointer to a GSKSOCDATA structure which needs to be used in subsequent SSL for VSE operations.

GSKSSOCREAD

The GSKSSOCREAD call receives data on a secure socket connection.

Example in COBOL

```
WORKING-STORAGE SECTION.  
  01 SOC-FUNCTION PIC X(16) VALUE 'GSKSSOCREAD  '.  
  01 SSOCDATA PIC 9(8) 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 SSOCDATA NBYTE BUF  
  ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

Parameter Values Set by the Application

SOC-FUNCTION

A 16-byte character field containing 'GSKSSOCREAD'. The field is left-justified and padded on the right with blanks.

SSOCDATA

Address of GSKSOCDATA structure as returned in RETCODE by the GSKSSOCINIT call.

NBYTE

A fullword binary number set to the size of BUF. GSKSSOCREAD will not return more than the number of bytes specified in NBYTE even if more data is available. The length of the data buffer must be either 64 Kb or at least 32 bytes larger than the largest send buffer that is to be received.

BUF 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.

Parameter Values Returned to the Application

ERRNO

A fullword binary field. May show detailed error information.

RETCODE

A fullword binary field that returns one of the following

0 or greater 0.

Successful call. RETCODE denotes the number of bytes which have been received.

less than 0.

An error occurred. Please refer to VSE library member SSLVSE.A or to the *TCP/IP for VSE 1.5 Optional Features* for a detailed description of error return codes.

GSKSSOCRESET

The GSKSSOCRESET call refreshes the security parameters, such as encryption keys, for a session.

Example in COBOL

```
WORKING-STORAGE SECTION.
  01 SOC-FUNCTION PIC X(16) VALUE 'GSKSSOCRESET  '.
  01 SSOCDATA    PIC 9(8) BINARY.
  01 ERRNO      PIC 9(8) BINARY.
  01 RETCODE    PIC S9(8) BINARY.

PROCEDURE DIVISION.
  CALL 'EZASOKET' USING SOC-FUNCTION SSOCDATA
                    ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

Parameter Values Set by the Application**SOC-FUNCTION**

A 16-byte character field containing 'GSKSSOCRESET'. The field is left-justified and padded on the right with blanks.

SSOCDATA

Address of GSKSOCDATA structure as returned in RETCODE by the GSKSSOCINIT call.

Parameter Values Returned to the Application**ERRNO**

A fullword binary field. May show detailed error information.

RETCODE

A fullword binary field that returns one of the following

0 Successful call.

less than 0.

An error occurred. Please refer to VSE library member SSLVSE.A or to the *TCP/IP for VSE 1.5 Optional Features* for a detailed description of error return codes.

GSKSSOCWRITE

The GSKSSOCWRITE call sends data on a secure socket connection.

Example in COBOL

```
WORKING-STORAGE SECTION.
  01 SOC-FUNCTION PIC X(16) VALUE 'GSKSSOCWRITE  '.
  01 SSOCDATA    PIC 9(8) BINARY.
  01 NBYTE      PIC 9(8) BINARY.
```

GSKSSOCWRITE

```
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 SSOCDATA NBYTE BUF
        ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

Parameter Values Set by the Application

SOC-FUNCTION

A 16-byte character field containing 'GSKSSOCWRITE'. The field is left-justified and padded on the right with blanks.

SSOCDATA

Address of GSKSOCDATA structure as returned in RETCODE by the GSKSSOCINIT call.

NBYTE

A fullword binary number set to the number of bytes to transmit. The maximum supported number of bytes is 64K.

BUF Specifies the buffer containing the data to be transmitted. BUF should have the size specified in NBYTE.

Parameter Values Returned to the Application

ERRNO

A fullword binary field. May show detailed error information.

RETCODE

A fullword binary field that returns one of the following

0 or greater 0.

Successful call. RETCODE denotes the number of bytes which have been sent.

less than 0.

An error occurred. Please refer to VSE library member SSLVSE.A or to the *TCP/IP for VSE 1.5 Optional Features* for a detailed description of error return codes.

GSKUNINIT

The GSKUNINIT call removes the current overall settings for the SSL environment.

It removes fields such as session timeout values and SSL protocols.

Example in COBOL

```
WORKING-STORAGE SECTION.
01 SOC-FUNCTION PIC X(16) VALUE 'GSKUNINIT' .
01 ERRNO        PIC 9(8) BINARY.
01 RETCODE      PIC S9(8) BINARY.

PROCEDURE DIVISION.
    CALL 'EZASOKET' USING SOC-FUNCTION
        ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

Parameter Values Set by the Application

SOC-FUNCTION

A 16-byte character field containing 'GSKUNINIT'. The field is left-justified and padded on the right with blanks.

Parameter Values Returned to the Application

ERRNO

A fullword binary field. May show detailed error information.

RETCODE

A fullword binary field that returns one of the following

0 Successful call.

not equal 0

An error occurred. Please refer to VSE library member SSLVSE.A or to the *TCP/IP for VSE 1.5 Optional Features* for a detailed description of error return codes.

INITAPI

The INITAPI call connects an application to the TCP/IP interface.

Almost all sockets programs that are written in COBOL, PL/I, or assembler language must issue the INITAPI 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
- SELECT
- SELECTEX
- SOCKET
- TAKESOCKET

Example in COBOL

```

WORKING STORAGE
  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
  CALL 'EZASOKET' USING SOC-FUNCTION MAXSOC IDENT SUBTASK
  MAXSNO ERRNO RETCODE.

```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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

Input parameter. A halfword binary field specifying the maximum number of sockets supported for this application. Currently, TCP/IP for VSE/ESA ignores this input and defaults the maximum number of sockets supported to 8192. Socket descriptor numbers are in the range 0 – 8191.

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

Starting with z/VSE 4.2, this parameter can be used to select the local TCP/IP stack used with this application. This 8-byte parameter can be set to "SOCKETnn" or just to "nn" (left- or right-adjusted, padded with 6 blanks). The value "nn" determines the ID of the selected TCP/IP stack as it is specified with the ID parameter in the TCP/IP startup JCL.

ADSNAME

The parameter can be used to specify the name of the TCP/IP Interface Routine used by the EZA processing environment. If nothing is specified here, the IBM-supplied TCP/IP Interface Routine EZASOH99 is used. Note that this specification can be overwritten with the following JCL statement: // SETPARAM [SYSTEM,] EZA\$PHA='routine-name'.

SUBTASK

Indicates an eight-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. If not specified or specified as 8 blanks, a default subtask name is used. In a batch environment we have

byte 0-2

first 3 characters of the JOBNAME

byte 3

hex F0

byte 4-7

the VSE Task Identifier

In a CICS transaction environment we have

byte 0-2

the CICS EIBTRNID (transaction identifier)

byte 3

hex F1

byte 4-7

the CICS EIBTASKN (task number)

Parameter Values Returned to the Application

MAXSNO

Output parameter. A fullword binary field containing the greatest descriptor number that may get assigned to this application. The socket descriptor assigned to the application will not be in consecutive order. Currently, TCP/IP for VSE/ESA always returns 8191.

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See "ERRNO Values" on page 74 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.

Example in COBOL

```

WORKING-STORAGE SECTION.
01  SOKET-FUNCTION          PIC X(16) VALUE 'IOCTL'.
01  S                       PIC 9(4)  BINARY.
01  COMMAND                 PIC 9(4)  BINARY.

01  IFREQ,
   3  NAME                   PIC X(16).
   3  FAMILY                 PIC 9(4)  BINARY.
   3  PORT                   PIC 9(4)  BINARY.
   3  ADDRESS                PIC 9(8)  BINARY.
   3  RESERVED              PIC X(8).

01  IFREQOUT,
   3  NAME                   PIC X(16).
   3  FAMILY                 PIC 9(4)  BINARY.
   3  PORT                   PIC 9(4)  BINARY.
   3  ADDRESS                PIC 9(8)  BINARY.
   3  RESERVED              PIC X(8).

01  GRP_IOCTL_TABLE(100)
02  IOCTL_ENTRY,
   3  NAME                   PIC X(16).
   3  FAMILY                 PIC 9(4)  BINARY.
   3  PORT                   PIC 9(4)  BINARY.
   3  ADDRESS                PIC 9(8)  BINARY.
   3  NULLS                 PIC X(8).

01  IOCTL_REQARG           POINTER ;
01  IOCTL_RETARG          POINTER ;
01  ERRNO                  PIC 9(8)  BINARY.

```

IOCTL

01 RETCODE PIC 9(8) BINARY.

PROCEDURE

```
CALL 'EZASOKET' USING SOC-FUNCTION S COMMAND REQARG  
RETARG ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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 the value shown in Table 9 on page 349.

REQARG and RETARG

REQARG is used to pass arguments to IOCTL and RETARG receives arguments from IOC. For the lengths and meanings of REQARG and RETARG see Table 9 on page 349.

Parameter Values Returned to the Application

RETARG

Returns an array whose size is based on the value in COMMAND.

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See “ERRNO Values” on page 74 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
----	-------------------------------

LISTEN

The LISTEN call creates a connection-request queue of a specified length for incoming connection requests.

Important: 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. If 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 LISTEN call requires a BIND request to be issued previously.

Example in COBOL

```

WORKING STORAGE
  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
  CALL 'EZASOKET' USING SOC-FUNCTION S BACKLOG ERRNO RETCODE.

```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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. This parameter is ignored. A value of 1 is always assumed.

Parameter Values Returned to the Application**ERRNO**

A fullword binary field. If RETCODE is negative, the field contains an error number. See “ERRNO Values” on page 74 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

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.

Important: This function call is not available with TCP/IP for VSE/ESA.

Example in COBOL

```

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.

```

NTOP

```
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 RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

Parameter Values Set by the Application

SOC-NTOP-FUNCTION

A 16-byte character field containing NTOP. The field is left justified and padded on the right with blanks.

NTOP-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

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 the PRESENTABLE-ADDRESS field on input and upon a successful return will contain the length of the converted IP address.

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. Otherwise ignore the ERRNO field. See “ERRNO Values” on page 74 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 from its standard text presentation form to its numeric binary form.

On successful completion, PTON returns the converted IP address in the buffer provided.

Important: This function call is not available with TCP/IP for VSE/ESA.

Example in COBOL

```

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.
```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

Parameter Values Set by the Application

SOC-PTON-FUNCTION

A 16-byte character field containing PTON. The field is left justified and padded on the right with blanks.

AF-INET6

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

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

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

A fullword binary field. If RETCODE is negative, the field contains an error number. Otherwise ignore the ERRNO field. See “ERRNO Values” on page 74 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

READ

The READ call reads the data on sockets.

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 281 for a subroutine that translates ASCII input data to EBCDIC.

Example in COBOL

```

WORKING STORAGE
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
    CALL 'EZASOKET' USING SOC-FUNCTION S NBYTE BUF
                        ERRNO RETCODE.

```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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 “ERRNO Values” on page 74 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

Value Description

- | | |
|----|--|
| 0 | A zero 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 call reads data on a socket and stores it into a set of buffers.

If a datagram socket is too long to fit into the supplied buffers, the extra bytes are discarded.

Example in COBOL

```

WORKING-STORAGE SECTION.
  01 SOC-FUNCTION          PIC X(16) VALUE 'READV'.
  01 S                     PIC 9(4) BINARY.
  01 ERRNO                 PIC 9(8) BINARY.
  01 RETCODE               PIC S9(8) BINARY.
  01 IOVCNT                PIC 9(8) BINARY.
  01 IOV.
      03 BUFFER-ENTRY OCCURS 5 TIMES.
          05 IOV-POINTER  USAGE IS POINTER.
          05 RESERVED     PIC X(4).
          05 IOV-LENGTH   PIC 9(8) BINARY.
  01 HEAPID                PIC S9(9) BINARY VALUE IS 0.

```

PROCEDURE DIVISION.

```

      MOVE 50 TO IOV-LENGTH(1).
      MOVE 50 TO IOV-LENGTH(2).
      MOVE 50 TO IOV-LENGTH(3).
      MOVE 50 TO IOV-LENGTH(4).
      MOVE 50 TO IOV-LENGTH(5).
      CALL 'CEEGETST' USING HEAPID, IOV-LENGTH(1), IOV-POINTER(1),
          FC.
      CALL 'CEEGETST' USING HEAPID, IOV-LENGTH(2), IOV-POINTER(2),
          FC.
      CALL 'CEEGETST' USING HEAPID, IOV-LENGTH(3), IOV-POINTER(3),
          FC.
      CALL 'CEEGETST' USING HEAPID, IOV-LENGTH(4), IOV-POINTER(4),
          FC.
      CALL 'CEEGETST' USING HEAPID, IOV-LENGTH(5), IOV-POINTER(5),
          FC.

      CALL 'EZASOKET' USING SOC-FUNCTION S IOV IOVCNT
          ERRNO RETCODE.

```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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 halfword binary number set to the socket descriptor of the socket that is going to read the data.

IOV An array of three fullword structures with the number of structures equal to the value of IOVCNT.

The format of the structure is as follows:

- Fullword 1: The address of the data buffer. This buffer is filled by the completion of the call.
- Fullword 2: reserved
- Fullword 3: The length of the data buffer referred to by Fullword 1

IOVCNT

A fullword binary field specifying the number of data buffers provided for this call. The maximum is 120.

Parameter Values Returned to the Application

ERRNO

A fullword binary field. If RETCODE is negative, this field contains an error number. See "ERRNO Values" on page 74 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

Value	Description
0	The connection is closed and no data is available.
>0	The number of bytes copied into the buffer.
-1	An error occurred. 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 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 EWOULDBLOCK. See "FCNTL" on page 209 or "IOCTL" on page 249 for a description of how to set nonblocking mode.

Note: See "EZACIC05" on page 281 for a subroutine that translates ASCII input data to EBCDIC.

Example in COBOL

```

WORKING STORAGE
  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
  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
  CALL 'EZASOKET' USING SOC-FUNCTION S FLAGS NBYTE BUF
  ERRNO RETCODE.

```

For equivalent PL/I and assembler language declarations, see "Converting Parameter Descriptions" on page 201.

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 which must be set to NO-FLAG or 0.

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 "ERRNO Values" on page 74 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.

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 EWOULDBLOCK. See “FCNTL” on page 209 or “IOCTL” on page 249 for a description of how to set nonblocking mode.

Important: IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply, if this program is used.

Note: See “EZACIC05” on page 281 for a subroutine that will translate ASCII input data to EBCDIC.

Example in COBOL

```

WORKING STORAGE
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.
01 NBYTE         PIC 9(8) BINARY.
01 BUF           PIC X(length of buffer).
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).
01 ERRNO         PIC 9(8) BINARY.
01 RETCODE       PIC S9(8) BINARY.

PROCEDURE
CALL 'EZASOKET' USING SOC-FUNCTION S FLAGS
                  NBYTE BUF NAME ERRNO RETCODE.

```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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 which must be set to NO-FLAG or 0.

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

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. Include the PRD1.MACLIB(EZBREHST) 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 TCP/IP the value is a decimal 2, indicating AF_INET.

PORT A halfword binary field specifying the port number of the sending socket.

IPv4-ADDRESS

A fullword binary field specifying the 32-bit IPv4 Internet address.

RESERVED

Specifies eight bytes of binary zeros. 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 that specifies the length of the IPv6 socket address structure. Any value specified by the use of this field is ignored, if processed as input. The field is set to 0, if processed as output.

FAMILY

A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary field specifying the port number of the sending socket.

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 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 "ERRNO Values" on page 74 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.

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, if 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.

Defining Which Sockets to Test

The SELECT call monitors for read operations, write operations, and exception operations:

- If 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.
- If 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.
- If 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 zero; the leftmost bit represents socket descriptor 31, and so on. If your process uses 32 or fewer sockets, the bit string is one fullword. If your process uses 33 sockets, the bit string is two full words. The first fullword represents socket descriptors 0 to 31, the second fullword is for socket descriptors 32 to 63. 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 282.

Read Operations

Read operations include ACCEPT, READ, RECV, or RECVFROM calls. A socket is ready to be read if data has been received for it, or if a connection request has occurred.

SELECT

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 ready for reading.

Write Operations

A socket is selected for writing (ready to be written) if:

- 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 EINPROGRESS. This socket will be selected for write, when the CONNECT completes.

A call to WRITE, SEND, or SENDTO blocks, if 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.

To test whether any of several sockets is ready for writing, set the WSNDMSK bits representing those sockets to one before issuing the SELECT call. When the SELECT call returns, the corresponding bits in the WRETMSK indicate sockets 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. If 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 one. 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 zero through the MAXSOC value.

TIMEOUT Parameter

If the time specified in the TIMEOUT parameter elapses before any event is detected, the SELECT call returns, RETCODE is set to 0.

Example in COBOL

```
WORKING STORAGE
  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 RRETSK      PIC X(*).
01 WRETSK      PIC X(*).
01 ERETSK      PIC X(*).
01 ERRNO       PIC 9(8) BINARY.
01 RETCODE     PIC S9(8) BINARY.

```

PROCEDURE

```

CALL 'EZASOKET' USING SOC-FUNCTION MAXSOC TIMEOUT
RSNDSK WSNDSK ESNDSK
RRETSK WRETSK ERETSK
ERRNO RETCODE.

```

The following example shows a SELECT call instruction:

* The bit mask lengths can be determined from the expression:
 $((\text{maximum socket number} + 32) / 32 \text{ (drop the remainder)}) * 4$

Bit masks are 32-bit fullwords with one bit for each socket. Up to 32 sockets fit into one 32-bit mask [PIC X(4)]. If you have 33 sockets, you must allocate two 32-bit masks [PIC X(8)].

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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

Input parameter. A fullword binary field specifying the largest socket descriptor number to be checked plus 1 (remember, TCP/IP for VSE/ESA supports socket descriptor numbers from 0 to 8191).

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 zero.

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 timeout after 3.5 seconds, set TIMEOUT-SECONDS to 3 and TIMEOUT-MICROSEC to 500000.

RSNDSK

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.

SELECT

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 set.
- 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 set.
- 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 "ERRNO Values" on page 74 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 or list of ECBs.

It completes, if 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

For a detailed description on testing sockets, refer to the description of "SELECT" on page 261.

Example in COBOL

```

WORKING STORAGE
  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 RSNDSK         PIC X(*).
  01 WSNDSK         PIC X(*).
  01 ESNDSK         PIC X(*).
  01 RRETSK         PIC X(*).
  01 WRETSK         PIC X(*).
  01 ERETSK         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
  CALL 'EZASOKET' USING SOC-FUNCTION MAXSOC TIMEOUT
                    RSNDSK WSNDSK ESNDSK
                    RRETSK WRETSK ERETSK
                    SELECB ERRNO RETCODE.

```

* The bit mask lengths can be determined from the expression:
 $((\text{maximum socket number} + 32) / 32 \text{ (drop the remainder)}) * 4$

For equivalent PL/I and assembler language declarations, see "Converting Parameter Descriptions" on page 201.

Parameter Values Set by the Application

MAXSOC

Input parameter. A fullword binary field specifying the largest socket descriptor number to be checked plus 1 (remember, TCP/IP for VSE/ESA supports socket descriptor numbers from 0 to 8191).

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:

SELECTEX

- 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 timeout after 3.5 seconds, set TIMEOUT-SECONDS to 3 and TIMEOUT-MICROSEC to 500000.

RSNDMSK

The bit-mask array to control checking for read interrupts. If this parameter is not specified or the specified bit-mask is zeros, the SELECT will not check for read interrupts. The length of this bit-mask array is dependent on the value in MAXSOC.

WSNDMSK

The bit-mask array to control checking for write interrupts. If this parameter is not specified or the specified bit-mask is zeros, the SELECT will not check for write interrupts. The length of this bit-mask array is dependent on the value in MAXSOC.

ESNDMSK

The bit-mask array to control checking for exception interrupts. If this parameter is not specified or the specified bit-mask is zeros, the SELECT will not check for exception interrupts. The length of this bit-mask array is dependent on the value in MAXSOC.

SELECB

An ECB which, if posted, causes completion of the SELECTEX.

If an ECB list is specified, you must set the high-order bit of the last entry in the ECB list to one to signify it is the last entry, and you must add the LIST keyword. The ECBs must reside in the caller primary address space.

Note: The maximum number of ECBs that can be specified in a list is 254.

Parameter Values Returned to the Application

ERRNO

A fullword binary field; if RETCODE is negative, this contains an error number. See “ERRNO Values” on page 74 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 zero) or one of the caller's ECBs has been posted (ECB value will be non-zero 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 | Error; check ERRNO. |

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.

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 281 for a subroutine that will translate EBCDIC input data to ASCII.

Example in COBOL

```

WORKING STORAGE
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
CALL 'EZASOKET' USING SOC-FUNCTION S FLAGS NBYTE
                   BUF ERRNO RETCODE.

```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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 which must be set to 0.

NBYTE

A fullword binary number set to the number of bytes of data to be transferred.

SEND

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 "ERRNO Values" on page 74 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.

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 281 for a subroutine that will translate EBCDIC input data to ASCII.

Important: IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply, if this program is used.

Example in COBOL

```
WORKING STORAGE
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.
01 NBYTE         PIC 9(8) BINARY.
01 BUF           PIC X(length of buffer).
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).
01 ERRNO         PIC 9(8) BINARY.
01 RETCODE       PIC S9(8) BINARY.
```

```

PROCEDURE
  CALL 'EZASOKET' USING SOC-FUNCTION S FLAGS NBYTE
                          BUF NAME ERRNO RETCODE.

```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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 must be set to 0.

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

Input parameter. The address of the IPv4 or IPv6 target. Include the PRD1.MACLIB(EZBREHST) 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 TCP/IP the value is a decimal 2, indicating AF_INET.

PORT A halfword binary field specifying the port number bound to the socket.

IPv4-ADDRESS

A fullword binary field specifying the 32-bit IPv4 Internet address of the socket.

RESERVED

Specifies eight bytes of binary zeros. 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 that specifies the length of the IPv6 socket address structure. Any value specified by the use of this field is ignored, if processed as input. The field is set to 0, if processed as output.

FAMILY

A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary field specifying the port number bound to the socket.

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 of the socket, in network byte order, of the client 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.

Parameter Values Returned to the Application**ERRNO**

A fullword binary field. If RETCODE is negative, the field contains an error number. See "ERRNO Values" on page 74 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 OPTVAL parameter is optional and can be set to 0, if data is not needed by the command. The OPTLEN parameter must be set to the size of the data pointed to by OPTVAL.

Example in COBOL

```

WORKING STORAGE
01 SOC-FUNCTION    PIC X(16) VALUE IS 'SETSOCKOPT'.
01 S              PIC 9(4) BINARY.
01 OPTNAME        PIC 9(8) BINARY.
                   88 SO-REUSEADDR VALUE 4.
                   88 SO-KEEPALIVE VALUE 8.
                   88 SO-LINGER    VALUE 128.
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

```

CALL 'EZASOKET' USING SOC-FUNCTION S OPTNAME
                        OPTVAL OPTLEN ERRNO RETCODE.

```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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

Specify one of the following values.

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.

IP_ADD_SOURCE_MEMBERSHIP

Use this option to enable an application to join a source multicast group on a specific interface and a specific source address. You must specify an interface and a source address with this option. Applications that want to receive multicast datagrams need to join source multicast groups. This is an IPv4-only socket option.

IP_BLOCK_SOURCE

Use this option to enable an application to block multicast packets that have a source address that matches the given IPv4 source address. You must specify an interface and a source address with this option. The specified multicast group must have been joined previously. This is an IPv4-only socket option.

IP_DROP_MEMBERSHIP

Use this option to enable an application to exit a multicast group or to exit all sources for a multicast group. This is an IPv4-only socket option.

IP_DROP_SOURCE_MEMBERSHIP

Use this option to enable an application to exit a source multicast group. This is an IPv4-only socket option.

IP_MULTICAST_IF

Use this option to set 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.

IP_MULTICAST_LOOP

Use this option to control 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.

IP_MULTICAST_TTL

Use this option to set 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.

IP_UNBLOCK_SOURCE

Use this option to enable an application to unblock a previously blocked source for a given IPv4 multicast group. You must specify an interface and a source address with this option. This is an IPv4-only socket option.

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.

IPV6_LEAVE_GROUP

Use this option to control the reception of multicast packets and specify that the socket leave a multicast group. This is an IPv6-only socket option.

IPV6_MULTICAST_HOPS

Use to set the hop limit used for outgoing multicast packets. This is an IPv6-only socket option.

IPV6_MULTICAST_IF

Use this option to set the index of the IPv6 interface used for sending outbound multicast datagrams from the socket application. This is an IPv6-only socket option.

IPV6_MULTICAST_LOOP

Use this option to control whether a multicast datagram is looped back on the outgoing interface by the IP layer for local delivery, if 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.

IPV6_UNICAST_HOPS

Use this option to set the hop limit used for outgoing unicast IPv6 packets. This is an IPv6-only socket option.

IPV6_V6ONLY

Use this option to set 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.

MCAST_BLOCK_SOURCE

Use this option to enable an application to block multicast packets that have a source address that matches the given source address. You must specify an interface index and a source address with this option. The specified multicast group must have been joined previously.

MCAST_JOIN_GROUP

Use this option to enable an application to join a multicast group

on a specific interface. You must specify an interface index. Applications that want to receive multicast datagrams must join multicast groups.

MCAST_JOIN_SOURCE_GROUP

Use this option to enable an application to join a source multicast group on a specific interface and a source address. You must specify an interface index and the source address. Applications that want to receive multicast datagrams only from specific source addresses need to join source multicast groups.

MCAST_LEAVE_GROUP

Use this option to enable an application to exit a multicast group or exit all sources for a given multicast groups.

MCAST_LEAVE_SOURCE_GROUP

Use this option to enable an application to exit a source multicast group.

MCAST_UNBLOCK_SOURCE

Use this option to enable an application to unblock a previously blocked source for a given multicast group. You must specify an interface index and a source address with this option.

SO_REUSEADDR

This option is provided for source compatibility reasons only. It will not perform any action. TCP/IP implicitly allows for immediate address reuse.

SO-KEEPALIVE

This option is provided for source compatibility reasons only. It will not perform any action. Instead the user should use the common TCP/IP setting: SET PULSE_TIME=nnn.

SO-LINGER

Controls how TCP/IP deals with data that it has not been able to transmit when the socket is closed. This option has meaning only for stream sockets.

- If LINGER is enabled and CLOSE is called, the calling program is blocked until the data is successfully transmitted or the connection has timed out.
- If LINGER is disabled, the CLOSE call returns without blocking the caller, and TCP/IP continues to attempt to send the data for a specified period of time. Although this usually provides sufficient time to complete the data transfer, use of the LINGER option does not guarantee successful completion because TCP/IP only waits the amount of time specified in OPTVAL LINGER.

The default is DISABLED.

OPTVAL

Contains data which further defines the option specified in OPTNAME.

- For OPTNAME of SO_REUSEADDR, OPTVAL is a one-word binary integer. Set OPTVAL to a nonzero positive value to enable the option; set OPTVAL to zero to disable the option.
- For SO-LINGER, OPTVAL assumes the following structure:

```
ONOFF      PIC X(4).
LINGER     PIC 9(8) BINARY.
```

SETSOCKOPT

Set `ONOFF` to a nonzero value to enable the option; set it to zero to disable the option. Set the `LINGER` value to the amount of time (in seconds) TCP/IP will linger after the `CLOSE` call.

OPTLEN

A fullword binary number specifying the length of the data returned in `OPTVAL`.

Parameter Values Returned to the Application

ERRNO

A fullword binary field. If `RETCODE` is negative, the field contains an error number. See "ERRNO Values" on page 74 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
----	--

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 `HOW` parameter determines the direction of traffic to shutdown.

If 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 will remain in the system for up to 30 seconds after the `CLOSE` call is issued. In high volume, transaction-based systems this can impact performance severely.

If the `SHUTDOWN` call is issued, when the `CLOSE` call is received, the connection can be closed immediately, instead of waiting for the 30 second delay.

Example in COBOL

```
WORKING STORAGE
  01 SOC-FUNCTION    PIC X(16) VALUE IS 'SHUTDOWN'.
  01 S              PIC 9(4) BINARY.
  01 HOW            PIC 9(8) BINARY.
  88 END-BOTH      VALUE 2.
  01 ERRNO         PIC 9(8) BINARY.
  01 RETCODE       PIC S9(8) BINARY.

PROCEDURE
  CALL 'EZASOKET' USING SOC-FUNCTION S HOW ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see "Converting Parameter Descriptions" on page 201.

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. The following value can be set:

Value Description

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 "ERRNO Values" on page 74 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.

Important: IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply, if this program is used.

Example in COBOL

```
WORKING STORAGE
01 SOC-FUNCTION    PIC X(16) VALUE IS 'SOCKET'.
01 AF              PIC 9(8) COMP VALUE 2.
01 SOCTYPE        PIC 9(8) BINARY.
   88 STREAM      VALUE 1.
   88 DATAGRAM    VALUE 2.
01 PROTO          PIC 9(8) BINARY.
01 ERRNO          PIC 9(8) BINARY.
01 RETCODE        PIC S9(8) BINARY.
```

PROCEDURE

```
CALL 'EZASOKET' USING SOC-FUNCTION AF SOCTYPE
                        PROTO ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see "Converting Parameter Descriptions" on page 201.

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. Specify one of the following:

Value Description

SOCKET

'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.

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. |

Note: RAW sockets are not supported.

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. If this field is set to 17, the UDP Protocol is used. If it is set to 6, the TCP protocol is used.

Parameter Values Returned to the Application

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See "ERRNO Values" on page 74, 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, which it obtained from the concurrent server. See "GIVESOCKET" on page 234 for a discussion of the use of GIVESOCKET and TAKESOCKET calls.

Note: If 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.

Important: IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply, if this program is used.

Example in COBOL

```

WORKING STORAGE
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
  CALL 'EZASOKET' USING SOC-FUNCTION SOCRECV CLIENT
                    ERRNO RETCODE.

```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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, 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.

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: TAKESOCKET can only acquire a socket of the same address family from a GIVESOCKET.

NAME

Specifies an 8-byte character field set to the VSE partition identifier of the program that gave the socket.

TASK Specifies an eight-byte character field set to the task identifier of the task that gave the socket.

RESERVED

A 20-byte reserved field. This field is required, and only used internally.

Parameter Values Returned to the Application**ERRNO**

A fullword binary field. If RETCODE is negative, the field contains an error number. See “ERRNO Values” on page 74 for information about ERRNO return codes.

TAKESOCKET

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

TERMAPI

This call terminates the session created by INITAPI.

Example in COBOL

```
WORKING STORAGE
  01 SOC-FUNCTION    PIC X(16) VALUE IS 'TERMAPI'.

PROCEDURE
  CALL 'EZASOKET' USING SOC-FUNCTION.
```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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 281 for a subroutine that will translate EBCDIC output data to ASCII.

Example in COBOL

```
WORKING STORAGE
  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
  CALL 'EZASOKET' USING SOC-FUNCTION S NBYTE BUF
  ERRNO RETCODE.
```


For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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 “ERRNO Values” on page 74 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 zero indicates the number of bytes of data written.
-1	Check ERRNO for an error code.

WRITEV

The WRITEV call writes data on a socket from a set of buffers.

Example in COBOL

```

WORKING-STORAGE SECTION.
    01 SOC-FUNCTION          PIC X(16) VALUE 'WRITEV          '.
    01 S                    PIC 9(4) BINARY.
    01 ERRNO                PIC 9(8) BINARY.
    01 RETCODE              PIC S9(8) BINARY.
    01 IOVCNT               PIC 9(8) BINARY.
    01 IOV.
        03 BUFFER-ENTRY OCCURS 5 TIMES.
            05 IOV-POINTER  USAGE IS POINTER.
            05 RESERVED     PIC X(4).
            05 IOV-LENGTH   PIC 9(8) BINARY.
    01 HEAPID                PIC S9(9) BINARY VALUE IS 0.

PROCEDURE DIVISION.

    MOVE 50 TO IOV-LENGTH(1).
    MOVE 50 TO IOV-LENGTH(2).
    MOVE 50 TO IOV-LENGTH(3).
    MOVE 50 TO IOV-LENGTH(4).
    MOVE 50 TO IOV-LENGTH(5).
    CALL 'CEEGETST' USING HEAPID, IOV-LENGTH(1), IOV-POINTER(1),
        FC.
    CALL 'CEEGETST' USING HEAPID, IOV-LENGTH(2), IOV-POINTER(2),
        FC.
    CALL 'CEEGETST' USING HEAPID, IOV-LENGTH(3), IOV-POINTER(3),

```

WRITEV

```
FC.  
CALL 'CEEGETST' USING HEAPID, IOV-LENGTH(4), IOV-POINTER(4),  
FC.  
CALL 'CEEGETST' USING HEAPID, IOV-LENGTH(5), IOV-POINTER(5),  
FC.  
  
* Call subroutine to fill the IOV structure  
CALL 'subroutine' USING IOV.  
  
CALL 'EZASOKET' USING SOC-FUNCTION S IOV IOVCNT  
ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

Parameter Values Set by the Application

SOC-FUNCTION

A 16-byte character field containing WRITEV. 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 for which the data is written.

IOV An array of three fullword structures with the number of structures equal to the value of IOVCNT.

The format of the structure is as follows:

- Fullword 1: The address of the data buffer.
- Fullword 2: reserved
- Fullword 3: The length of the data buffer referred to by Fullword 1

IOVCNT

A fullword binary field specifying the number of data buffers provided for this call. The maximum is 120.

Parameter Values Returned to the Application

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See “ERRNO Values” on page 74 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

Value	Description
-------	-------------

- | | |
|----|---|
| ≥0 | The number of bytes written. |
| -1 | An error occurred. Check ERRNO for an error code. |

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; TCP/IP for VSE/ESA 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:

Using Data Translation Programs for Socket Call Interface

- EZACIC04—Translates EBCDIC data to ASCII data
- EZACIC05—Translates ASCII data to EBCDIC data

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:

- 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.

Example in COBOL

```
WORKING STORAGE
  01 OUT-BUFFER  PIC X(length of output).
  01 LENGTH      PIC 9(8) BINARY.

PROCEDURE
  CALL 'EZACIC04' USING OUT-BUFFER LENGTH.
```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

Parameter Values Set by the Application

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.

Example in COBOL

```
WORKING STORAGE
  01 IN-BUFFER  PIC X(length of output)
  01 LENGTH      PIC 9(8) BINARY VALUE

PROCEDURE
  CALL 'EZACIC05' USING IN-BUFFER LENGTH.
```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

Parameter Values Set by the Application

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.

Example in COBOL

```
WORKING STORAGE
  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     PIC 9(16) COMP.

  01 BIT-FUNCTION-CODES.
    05 CTOB               PIC X(4) VALUE 'CTOB'.
    05 BTOC               PIC X(4) VALUE 'BTOC'.

  01 BIT-MASK-LENGTH     PIC 9(8) COMP VALUE 50.
  01 CHAR-STRING-LENGTH PIC 9(8) COMP VALUE 64.
```

```
PROCEDURE CALL (to convert from character to binary)
  CALL 'EZACIC06' USING CTOB
                        BIT-MASK
                        CHAR-MASK
                        CHAR-STRING-LENGTH
                        RETCODE.
```

```
PROCEDURE CALL (to convert from binary to character)
  CALL 'EZACIC06' USING BTOC
                        BIT-MASK
                        CHAR-MASK
                        BIT-MASK-LENGTH
                        RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

Parameter Values Set by the Application

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.

BIT-MASK

Specifies the bit string to be translated for the SELECT call. The bits are ordered right-to-left with the right-most bit representing socket 0. The socket positions in the character array are indexed starting with one making socket zero index number one in the character array. You should keep this in mind when turning character positions on and off.

COMMAND

BTOC—Specifies bit string to character array translation.

CTOB—Specifies character array to bit string translation.

BIT-MASK-LENGTH

Specifies the length of the bit-mask.

CHAR-STRING-LENGTH

Specifies the length of the char-mask.

Parameter Values Returned to the Application**RETCODE**

A binary field that returns one of the following:

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

If you want to use the SELECT call to test sockets zero, five, and nine, and you are using a character array to represent the sockets, you must set the appropriate characters in the character array to one. In this example, index positions one, six and ten in the character array are set to 1. Then you can call EZACIC06 with the COMMAND parameter set to CTOB. When EZACIC06 returns, BIT-MASK contains a fullword with bits zero, five, and nine (numbered from the right) turned on as required by the SELECT call. These instructions process the bit string shown in the following example.

```
MOVE ZEROS TO CHAR-STRING.
MOVE '1' TO CHAR-ENTRY(1), CHAR-ENTRY(6), CHAR-ENTRY(10).
CALL 'EZACIC06' USING CTOB BIT-MASK CHAR-MASK
      CHAR-STRING-LENGTH RETCODE.
MOVE BIT-MASK TO ....
```

When the select call returns and you want to check the bit-mask string for socket activity, enter the following instructions.

```
MOVE ..... TO BIT-MASK.
CALL 'EZACIC06' USING BTOC BIT-MASK CHAR-MASK
      BIT-MASK-LENGTH RETCODE.
PERFORM TEST-SOCKET THRU TEST-SOCKET-EXIT VARYING IDX
      FROM 1 BY 1 UNTIL IDX EQUAL 10.

TEST-SOCKET.
  IF CHAR-ENTRY(IDX) EQUAL '1'
    THEN PERFORM SOCKET-RESPONSE THRU SOCKET-RESPONSE-EXIT
    ELSE NEXT SENTENCE.
TEST-SOCKET-EXIT.
EXIT.
```

EZACIC08

The GETHOSTBYNAME and GETHOSTBYADDR calls were derived from C socket calls that return a structure known as HOSTENT. A given TCP/IP host can have multiple alias names and host internet addresses.

TCP/IP uses indirect addressing to connect the variable number of alias names and internet addresses in the HOSTENT structure that is 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:

- GETHOSTBYADDR or GETHOSTBYNAME returns a HOSTENT structure that indirectly addresses the lists of alias names and internet addresses.
- 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:
 1. The length of host name, if present
 2. The host name
 3. The number of alias names for the host
 4. The alias name sequence number
 5. The length of the alias name
 6. The alias name
 7. The host internet address type, always two for AF_INET
 8. The host internet address length, always 4 for AF_INET
 9. The number of host internet addresses for this host
 10. The host internet address sequence number
 11. The host internet address
- If the GETHOSTBYADDR or GETHOSTBYNAME call returns more than one alias name or host internet address (steps 3 and 9 above), the application program should repeat the call to EZACIC08 until all alias names and host internet addresses have been retrieved.

Example in COBOL

WORKING STORAGE

```

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

```
CALL 'EZASOKET' USING 'GETHOSTBYxxxx'
                    HOSTENT-ADDR
                    RETCODE.
```

Where xxxx is ADDR or NAME.

```
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
```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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 one 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 one 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.

EZACIC08

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	Invalid HOSTENT address

EZACIC09

The GETADDRINFO call was derived from the C socket call that returns 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 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.

Example in COBOL

```

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-canonical-name-len pic 9(8) binary value 2.
02 ai-numeric-host pic 9(8) binary value 4.
02 ai-numeric-serv pic 9(8) binary value 8.
02 ai-v4mapped pic 9(8) binary value 16.
02 ai-all pic 9(8) binary value 32.
02 ai-addrconfig pic 9(8) binary value 64.
*
* Variables used for the EZACIC09 call
*
01 ezacic09-parms.
02 res usage is pointer.
02 res-name-len pic 9(8) binary.
02 res-canonical-name pic x(256).
02 res-name usage is pointer.
02 res-next-addrinfo usage is pointer.
*
* Socket address structure
*
01 server-socket-address.
05 server-family pic 9(4) Binary Value 19.
05 server-port pic 9(4) Binary Value 9997.
05 server-flowinfo pic 9(8) Binary Value 0.
05 server-ipaddr.
10 filler pic 9(16) binary value 0.
10 filler pic 9(16) binary value 0.
05 server-scopeid pic 9(8) Binary Value 0.

LINKAGE SECTION.
01 L1.
03 HINTS-ADDRINFO.
05 HINTS-AI-FLAGS PIC 9(8) BINARY.
05 HINTS-AI-FAMILY PIC 9(8) BINARY.
05 HINTS-AI-SOCKTYPE PIC 9(8) BINARY.
05 HINTS-AI-PROTOCOL PIC 9(8) BINARY.
05 FILLER PIC 9(8) BINARY.
05 FILLER PIC 9(8) BINARY.
05 FILLER PIC 9(8) BINARY.
05 FILLER PIC 9(8) BINARY.
03 HINTS-ADDRINFO-PTR USAGE IS POINTER.
03 RES-ADDRINFO-PTR USAGE IS POINTER.
*
* RESULTS ADDRESS INFO
*
01 RESULTS-ADDRINFO.
05 RESULTS-AI-FLAGS PIC 9(8) BINARY.
05 RESULTS-AI-FAMILY PIC 9(8) BINARY.
05 RESULTS-AI-SOCKTYPE PIC 9(8) BINARY.
05 RESULTS-AI-PROTOCOL PIC 9(8) BINARY.
05 RESULTS-AI-ADDR-LEN PIC 9(8) BINARY.
05 RESULTS-AI-CANONICAL-NAME USAGE IS POINTER.
05 RESULTS-AI-ADDR-PTR USAGE IS POINTER.
05 RESULTS-AI-NEXT-PTR USAGE IS POINTER.
*
* SOCKET ADDRESS STRUCTURE FROM EZACIC09.

```

```

*
01 OUTPUT-NAME-PTR USAGE IS POINTER.
01 OUTPUT-IP-NAME.
   03 OUTPUT-IP-FAMILY PIC 9(4) BINARY.
   03 OUTPUT-IP-PORT PIC 9(4) BINARY.
   03 OUTPUT-IP-SOCK-DATA PIC X(24).
   03 OUTPUT-IPV4-SOCK-DATA REDEFINES OUTPUT-IP-SOCK-DATA.
       05 OUTPUT-IPV4-IPADDR PIC 9(8) BINARY.
       05 FILLER PIC X(20).
   03 OUTPUT-IPV6-SOCK-DATA REDEFINES OUTPUT-IP-SOCK-DATA.
       05 OUTPUT-IPV6-FLOWINFO PIC 9(8) BINARY.
       05 OUTPUT-IPV6-IPADDR.
           10 FILLER PIC 9(16) BINARY.
           10 FILLER PIC 9(16) BINARY.
       05 OUTPUT-IPV6-SCOPEID PIC 9(8) BINARY.
PROCEDURE DIVISION USING L1.
*
* Get and address from the resolver.
*
   move 'yournodename' to node-name.
   move 12 to node-name-len.
   move spaces to service-name.
   move 0 to service-name-len.
   move af-inet6 to hints-ai-family.
   move 49 to hints-ai-flags move 0 to hints-ai-socktype.
   move 0 to hints-ai-protocol.
   set address of results-addrinfo to res-addrinfo-ptr.
   set hints-addrinfo-ptr to address of hints-addrinfo.
   call 'EZASOCKET' using soket-getaddrinfo node-name node-name-len
mm      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.

```

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 201.

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

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. RETCODE

RETCODE

Output parameter. This fullword binary field contains the EZACIC09 return code:

Value	Description
≥0	Successful call
-1	Invalid RES address

Chapter 12. Using the Macro Application Programming Interface (EZASMI API)

This chapter describes the macro API for TCP/IP application programs written in System/390 assembler language.

The macro interface can be used to produce reentrant modules.

The following topics are included:

- Environmental restrictions and programming requirements
- Defining storage for the API macro
- Understanding common parameter descriptions
- Characteristics of stream sockets
- Task management and asynchronous function processing
- Using an unsolicited event exit routine
- Error messages and return codes
- Macros for assembler programs

Environmental Restrictions and Programming Requirements

The following restrictions apply to the Macro Socket API:

- The EZASMI API cannot be used with programs running in an ICCF Pseudo Partition.
- 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.
- When using the EZASMI macro API in CICS transactions while CICS operates with storage protection, all programs using the macro API need to be defined with EXECKEY(CICS). This is also true for those programs that link to these programs. TASKDATAKEY(CICS) for the transaction definition is NOT required.
- Addressability mode (AMODE) considerations
The EZASMI macro API must be invoked while the caller is in 31-bit AMODE.
- When using the macro API in CICS transactions, the EZA "task-related-user-exit" (TRUE) has to be activated before these transactions can be run. For details on how to activate this TRUE, refer to "CICS Considerations for the EZA Interfaces" on page 83.

EZASMI Macro Application Programming Interface (API)

This section describes the EZASMI Macro API for TCP/IP application programs written in the High Level Assembler language. The format and parameters are described for each socket call.

Note:

1. Reentrant code is supported by this interface.
2. Register conventions: Register 0, 1, 14, 15 are used by the interface and must be, if necessary, saved prior to invocation. Register 13 must point to a 72-byte save area provided by the caller.

Defining Storage for the API Macro

The macro API requires a task storage area.

The task storage area must be known to and addressable by all socket users communicating across a specified connection. A connection runs between the application and TCP/IP. The most common way to organize storage is to assign one connection to each VSE 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 two alternatives 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, TIELENTH, which is equal to the length of the storage area. This can be used to issue a VSE GETVIS request to allocate the required storage. Please make sure that this storage area is cleared to binary 0's before using it.

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. A program can use the following format to build unique parameter list storage areas for each function call:

```

BINDPRML  EZASMI    MF=L          This will generate the storage used for
                                                building the parm list in the following BIND call
          EZASMI    TYPE=BIND,      *
          S=SOCKDESC,              *
          NAME=NAMEID,              *
          ERRNO=ERRNO,              *
          RETCODE=RETCODE,          *
          ECB=ECB1,                  *
          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(TIELENTH)          Length of TIE

EZASMI  TYPE=INITAPI,          *
        MAXSOC=MAX75,         *
        ERRNO=ERRNO,          *
        RETCODE=RETCODE,      *
        APITYPE=2,            *
        MAXSNO=MAXS,          *
        TASK=TIE1             *

EZASMI  TYPE=SOCKET,          *
        AF='INET',            *
        SOCTYPE='STREAM',     *
        ERRNO=ERRNO,          *
        RETCODE=RETCODE,      *
        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

address The name of the field that contains the value of the parameter. The following example illustrates a BIND macro where SOCKNO is set to 2.

```
MVC  SOCKNO,=H'2'
EZASMI TYPE=BIND,S=SOCKNO
```

**indaddr*

The name of the address field that contains the address of the field containing the parameter. The following example produces the same result as the example above.

```
MVC  SOCKNO,=H'2'
LA   0,SOCKNO
ST   0,SOCKADD
EZASMI TYPE=BIND,S=*SOCKADD
```

(reg) The name (equated to a number) or the number of a general purpose register. Do not use a register 0, 1, 14, or 15. The following example produces the same result as the previous examples.

```
MVC  SOCKNO,=H'2'
LA   3,SOCKNO
EZASMI TYPE=BIND,SOCKNO=(3)
```

'value' A literal value for the parameter; for example, AF='INET'

Characteristics of Stream 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 one byte, ten 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.

Task Management and Asynchronous Function Processing

The EZASMI socket interface allows asynchronous operation, although by default the task issuing a macro request is put into a WAIT state until the requested function is completed. 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 VSE event control block (ECB) on each socket call. The socket call 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 call drives the specified exit routine when the socket call is complete.

Restriction: This method is not supported with TCP/IP for VSE/ESA.

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.

Using the EZASMI macro you can specify an APITYPE parameter. APITYPE=2 is the only supported (and default) type. It allows to have more than one outstanding asynchronous socket call per socket descriptor (for example, a RECV and a SEND call).It requires the ECB method, if asynchronous macro calls are used.

The ECB input parameter for asynchronous calls must point to a 160- byte storage area:



Figure 19. ECB Input Parameter

The 156-byte storage area following the ECB is used during asynchronous function processing and must not be changed by the application program until the asynchronous function call completes (that is, until the ECB is posted).

Asynchronous functions are processed in the following sequence:

1. The application must issue the EZASMI TYPE=INITAPI with ASYNC='ECB'. The ASYNC parameter notifies the API that asynchronous processing is eventually used for this connection.
2. When an asynchronous function request with an ECB is issued by the application, the request is queued for processing and the API returns control to the application immediately. A successful function queuing returns with RETCODE=0 and ERRNO set to EINPROGRESS. If an error condition is encountered during function queuing, the API returns with RETCODE=-1 and ERRNO showing the error status of the asynchronous operation. The ECB is posted as well.
3. When the function completes (this may even occur before the function call returns to the application), the ECB is posted and function specific return (RETCODE) and error (ERRNO) information is returned.

The following example shows how to code an asynchronous macro function:

```

EZASMI TYPE=READ,      READ A BUFFER OF DATA FROM THE      *
      S=SOCKNO,        CONNECTION PEER.  I MAY NEED TO      *
      NBYTES=COUNT,  WAIT SO GIVE CONTROL BACK TO ME      *
      BUF=DATABUF,    AND LET ME ISSUE MY OWN WAIT.        *
      ERRNO=ERROR,    IT COULD BE PART OF A WAIT WHICH      *
      RETCODE=RCODE,  WOULD INCLUDE OTHER EVENTS.          *
      ECB=MYECB,      SPECIFY ECB/STORAGE AREA FOR INTERFACE *
      ERROR=ERRORRTN

```

WAIT MYECB TELL VSE TO WAIT UNTIL READ IS DONE

Error Messages and Return Codes

For information about error messages, refer to *toz/VSE Messages and Codes* and *TCP/IP for VSE 1.5 Messages and Codes*.

For information about error codes that are returned by TCP/IP, refer to "ERRNO Values" on page 74.

Debugging

Refer to Appendix B, "Debugging Facility for EZASMI and EZASOCKET Interfaces (EZAAPI Trace)," on page 525.

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, if the server receives a connection request from a client.

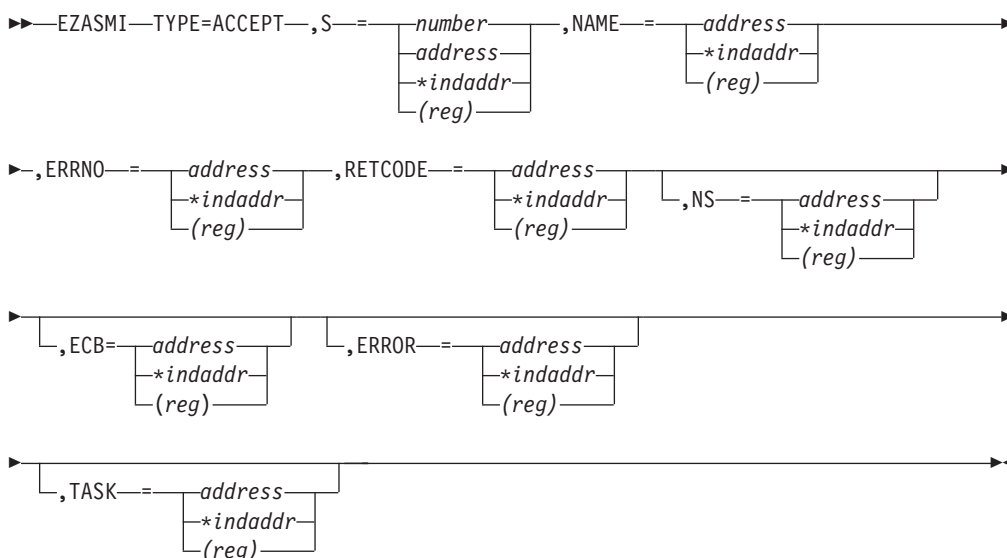
ACCEPT points to a socket that was created with a SOCKET macro and marked by a LISTEN macro. If a process waits for the completion of connection requests from several peer processes, a later ACCEPT macro can block until one of the CONNECT macros completes. To avoid this, issue a SELECT macro between the CONNECT and the ACCEPT macros. Concurrent server programs use the ACCEPT macro to pass connection requests to subtasks.

Important: IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply, if this program is used.

When issued, the ACCEPT macro:

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.

Format



Parameters

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

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 PRD1.MACLIB(EZBREHST) 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:

FAMILY

A halfword binary field specifying the IPv4 addressing family. For TCP/IP the value is a decimal 2, indicating AF_INET.

PORT A halfword binary field specifying the client's port number. 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'.

IPv4-ADDRESS

A fullword binary field specifying the 32-bit IPv4 Internet address.

RESERVED

Specifies eight bytes of binary zeros. This field is required, but is not used.

The IPv6 socket address structure contains the following fields:

NAMELEN

A 1-byte binary field that specifies the length of the IPv6 socket address structure. Any value specified by the use of this field is ignored, if processed as input. The field is set to 0, if processed as output.

FAMILY

A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary 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 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 "ERRNO Values" on page 74 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.

ACCEPT

NS Not supported for TCP/IP for VSE/ESA.

ECB Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.
- A 156-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.

ERROR

Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

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

BIND

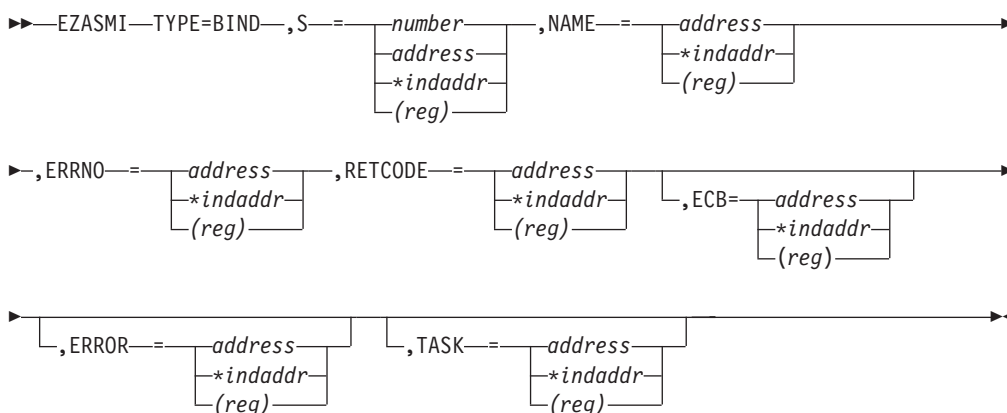
In a server program, the BIND macro normally follows a SOCKET macro to complete the new socket creation process.

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 macro.

In the AF_INET domain, the BIND macro for a stream socket can specify the networks from which it is willing to accept connection requests. Your application can select the network interface by setting **ADDRESS** to the internet address of the network from which you want to accept connection requests. Alternatively, your application can accept connection requests from any network if you set the address field to a fullword of zeros.

Important: IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply, if this program is used.

Format



Parameters

S Input parameter. A value or the address of a halfword binary number specifying the socket descriptor.

NAME

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 PRD1.MACLIB(EZBREHST) 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:

FAMILY

A halfword binary field specifying the IPv4 addressing family. For TCP/IP the value is a decimal 2, indicating AF_INET.

PORT A halfword binary field specifying the client's port number. If you set the port number to zero, TCP/IP assigns the port. The application can call the GETSOCKNAME macro after the BIND macro to discover the assigned port.

IPv4-ADDRESS

A fullword binary field specifying the 32-bit IPv4 Internet address.

RESERVED

Specifies eight bytes of binary zeros. This field is required, but is not used.

The IPv6 socket address structure contains the following fields:

NAMELEN

A 1-byte binary field that specifies the length of the IPv6 socket address structure. Any value specified by the use of this field is ignored, if processed as input. The field is set to 0, if processed as output.

FAMILY

A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary field that is set to the client port number. If you set the port number to zero, TCP/IP assigns the port. The application can call the GETSOCKNAME macro after the BIND macro to discover the assigned port.

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 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.

BIND

ERRNO

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

See "ERRNO Values" on page 74, 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
----	-------------------------------

ECB Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.
- A 156-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.

ERROR

Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

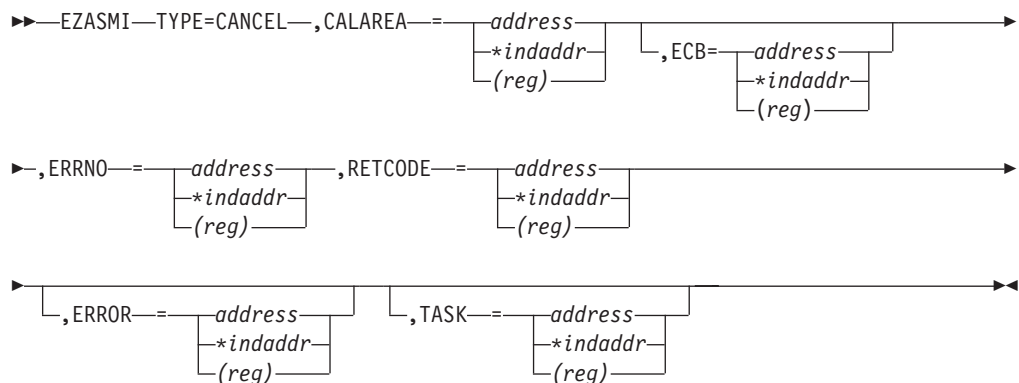
TASK Input parameter. The location of the task storage area in your program.

CANCEL

The CANCEL function terminates a call in progress.

The call being cancelled must have specified ECB .

Format



Parameters

CALAREA

Input parameter. The ECB specified in the call being cancelled.

ECB Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.

- A 156-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.

ERRNO

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

See "ERRNO Values" on page 74, for information about ERRNO return codes.

RETCODE

Output parameter. A fullword binary field. If RETCODE is 0, the CANCEL was successful. The error status (ERRNO) of the cancelled call is sent to ECANCELED. If RETCODE is -1, the CANCEL failed. Check ERRNO for an error code. For example, ERRNO is set to EINPROGRESS if the selected request cannot be cancelled because it is in progress, or set to EINVAL if the selected request cannot be cancelled because it has already been completed.

ERROR

Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

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

CLOSE

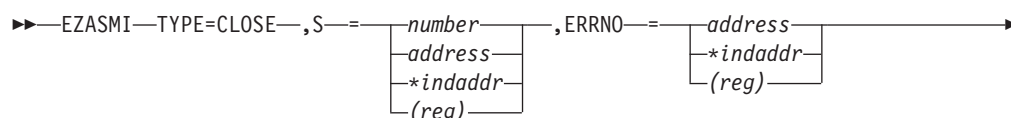
The CLOSE macro shuts down the socket and frees the resources that are allocated to the socket.

Issue the SHUTDOWN macro before you issue the CLOSE macro.

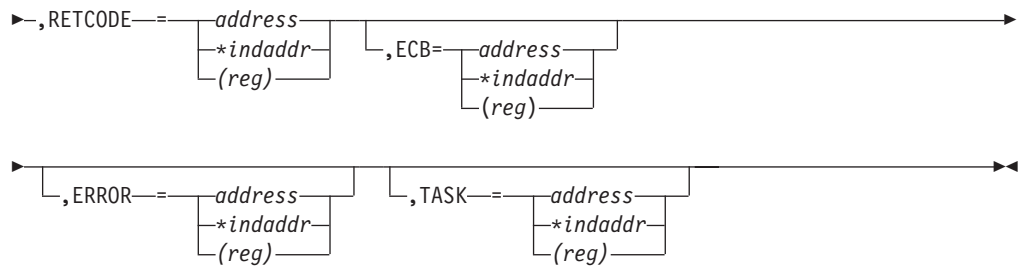
CLOSE can also be issued by a concurrent server after it gives a socket to a subtask program. After issuing GIVESOCKET and receiving notification that the client child has successfully issued TAKESOCKET, the concurrent server issues the CLOSE macro to complete the transfer of ownership.

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. SETSOCKET 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 371.

Format



CLOSE



Parameters

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.

See “ERRNO Values” on page 74, 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
----	-------------------------------

ECB Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.
- A 156-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.

ERROR

Input parameter. The location in your program to receive control, if 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.

Important: IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply, if this program is used.

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.

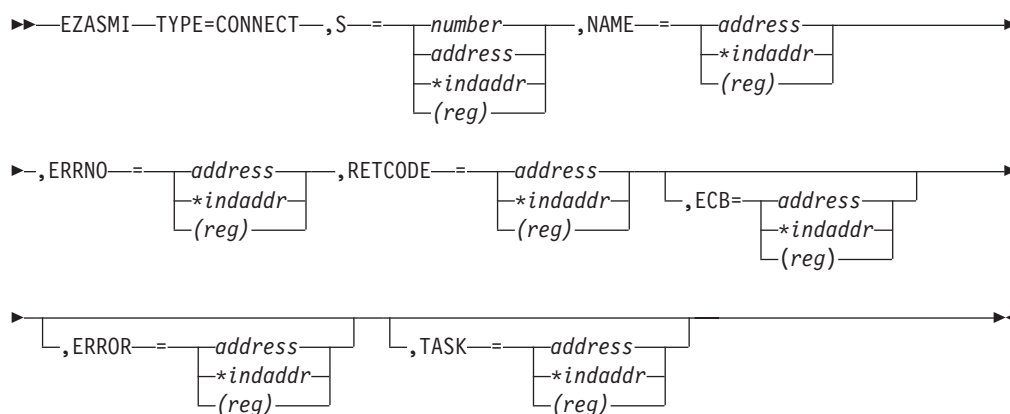
If the socket is in blocking mode, CONNECT blocks the calling program until the connection is established, or until an error is received.

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

- A zero RETCODE indicates that the connection was completed.
- A nonzero RETCODE with an ERRNO 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.

Format



Parameters

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 target to which the local, client socket is to be connected. Include the PRD1.MACLIB(EZBREHST) 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:

CONNECT

FAMILY

A halfword binary field specifying the IPv4 addressing family. For TCP/IP the value is a decimal 2, indicating AF_INET.

PORT A halfword binary field specifying the client's port number.

IPv4-ADDRESS

A fullword binary field specifying the 32-bit IPv4 Internet address.

RESERVED

Specifies eight bytes of binary zeros. This field is required, but is not used.

The IPv6 socket address structure contains the following fields:

NAMELEN

A 1-byte binary field that specifies the length of the IPv6 socket address structure. Any value specified by the use of this field is ignored, if processed as input. The field is set to 0, if processed as output.

FAMILY

A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary 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 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 "ERRNO Values" on page 74 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 Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.
- A 156-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.

ERROR

Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

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

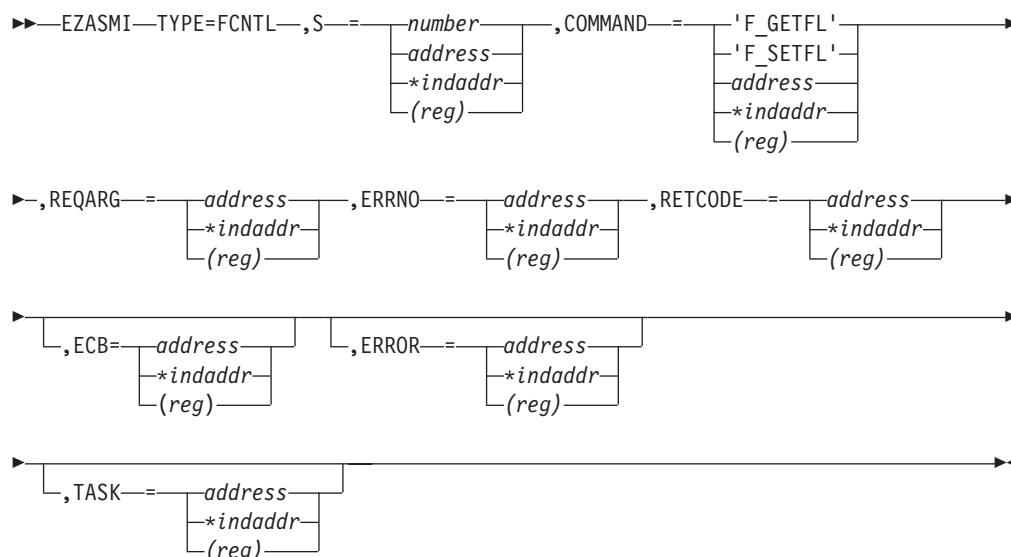
FCNTL

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 348 for another way to control socket blocking.

Format



Parameters

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:

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 EWOULDBLOCK.

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 “ERRNO Values” on page 74 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 zero in RETCODE. For either COMMAND, a RETCODE of -1 indicates an error. Check ERRNO for the error number.

ECB Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.
- A 156-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.

ERROR

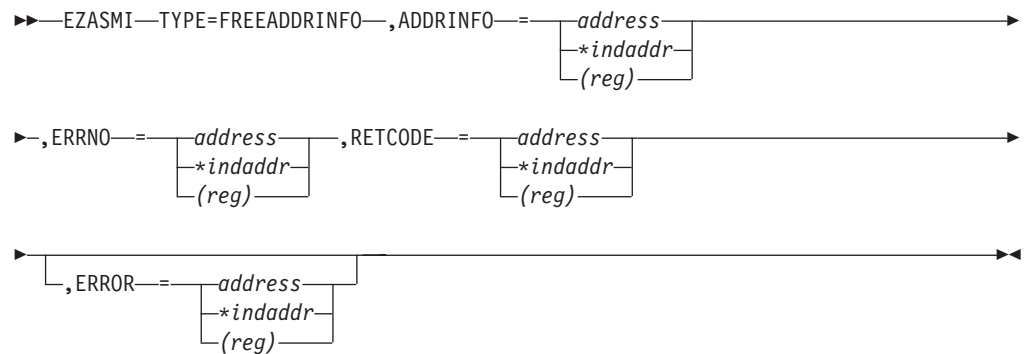
Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

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.

Important: This function call is not available with TCP/IP for VSE/ESA.

Format**Parameters****ADDRINFO**

Input parameter. The address of a set of address information structures returned by a 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 “ERRNO Values” on page 74 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

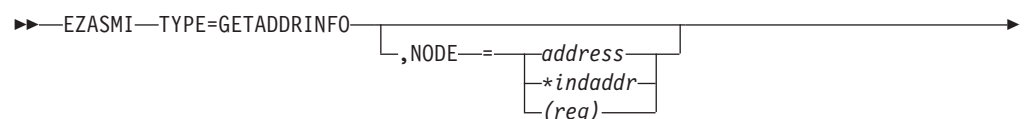
Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

GETADDRINFO

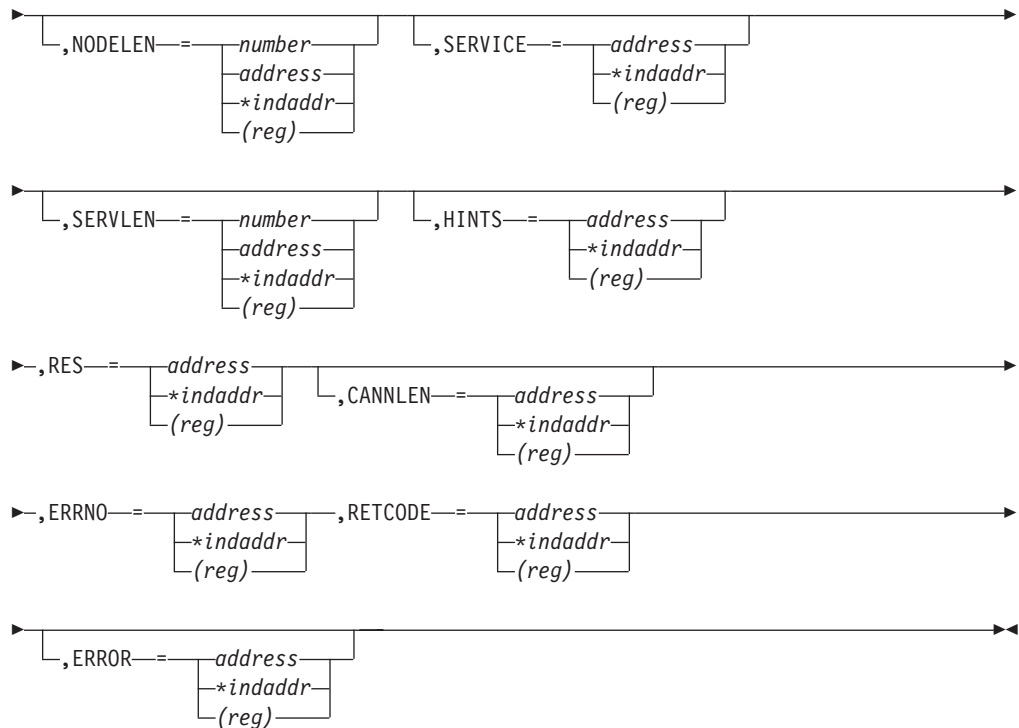
The GETADDRINFO macro translates the name of a service location (for example, a host name), a service name, or both, into a set of socket addresses and other associated information.

This information can be used to create a socket and connect to, or to send a datagram to, the specified service.

Important: This function call is not available with TCP/IP for VSE/ESA.

Format

GETADDRINFO



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 in the storage pointed to by the `HINTS` operand, `NODE` should contain the queried host's IP address in network byte order presentation form. This is an optional field, but if specified you must also code `NODELEN`. The `NODE` name being queried consists of up to `NODELEN` or up to the first binary zero.

You can append scope information to the host name by using the format `node%scope information`. The combined information must be 255 bytes or less.

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, `SERVICE` should contain the queried port number in presentation form. This is an optional field, but if specified you must also code `SERVLEN`. The `SERVICE` name being queried consists of up to `SERVLEN` or up to the first binary zero.

SERVLEN

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, 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.

The address information structure has the following fields:

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') or a decimal value of 1.

Specifies how to fill in the NAME pointed to by the returned RES.

If this flag is specified, 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, the IP address portion of the socket address structure pointed to by the returned RES will be set to INADDR_ANY for an IPv4 address or to the IPv6 unspecified address (in6addr_any) for an IPv6 address.

If this flag is not set, the returned address information will be suitable for the CONNECT call (for a connection-mode protocol) or for a CONNECT, SENDTO, or SENDMSG call (for a connectionless protocol). In this case, if the NODE argument is not specified, 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, 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, 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, if IPv6 is supported, the caller will accept IPv4-mapped IPv6 addresses. If 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.

If the AF field has a value of AF_INET6 and AI-ALL is set, the AI-V4MAPPED flag must also be set to indicate that the caller will accept all addresses (IPv6 and IPv4-mapped IPv6 addresses). If the AF field has a value of AF_UNSPEC, if 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, if the system supports IPv6, this flag is ignored.

AI-ADDRCONFIG (X'00000040') or a decimal value of 64.

If this flag is specified, 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, 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.

Note: To perform the binary ORing 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 ORing 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-ADDRCONFIG 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), 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 is only valid, if SERVICE is numeric (for example, SERVICE=23). A lookup for a SERVICE name will never occur in the appropriate services file using any protocol value other than SOCK_STREAM or SOCK_DGRAM.

If PROTO is not 0 and SOCTYPE is 0, 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, GETADDRINFO will proceed as follows:

- If SERVICE is null, or if SERVICE is numeric, 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 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, 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

GETADDRINFO

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, `GETADDRINFO` will proceed as follows:

- If `SERVICE` is null, or if `SERVICE` is numeric, 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 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 using the input protocol], the `GETADDRINFO` call will be failed with return code of `EAI_SERVICE`.

NAMELEN

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

CANNONNAME

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 contains a pointer to a chain of one or more address information structures. The structures are allocated in the key of the calling application. Do not use or reference these structures between MVS tasks. When you are finished using the structures, explicitly free their storage by specifying the returned pointer on a `FREEADDRINFO` call. Storage that is not explicitly freed is released when the task is ended.

The address information structure contains the following fields:

FLAGS

A fullword binary field that is not used as output.

AF A fullword binary field. The value returned in this field can be used as the `AF=` argument on the `TYPE=SOCKET` macro to create a socket suitable for use with the returned address `NAME`.

SOCTYPE

A fullword binary field. The value returned in this field can be used as the `SOCTYPE=` argument on the `TYPE=SOCKET` macro to create a socket suitable for use with the returned address `NAME`.

PROTO

A fullword binary field. The value returned in this field can be used as the PROTO= argument on the TYPE=SOCKET macro to create a socket suitable for use with the returned address NAME.

NAMELEN

A fullword binary field. The length of the NAME socket address structure. The value returned in this field can be used as the arguments for the TYPE=CONNECT or TYPE=BIND macros with such a socket, according to the AI_PASSIVE flag.

CANNONNAME

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, the CANNONNAME field in the first returned address information structure contains the address of storage containing the canonical name corresponding to the input NODE argument. If the canonical name is not available, the CANNONNAME field refers to the NODE argument or a string with the same contents. The CANNLEN field contains the length of the returned canonical name.

NAME

A fullword binary field. The address of the returned socket address structure. The value returned in this field can be used as the arguments for the TYPE=CONNECT or TYPE=BIND macros with such a socket, according to the AI_PASSIVE flag.

NEXT A fullword binary field. Contains the address of the next address information structure on the list, or 0's if it is the last structure on the list.

CANNLEN

Initially an input parameter. A fullword binary field used to contain the length of the canonical name returned by the RES CANNONNAME 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 "ERRNO Values" on page 74 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, if the application programming interface (API) processing module cannot be loaded.

GETCLIENTID

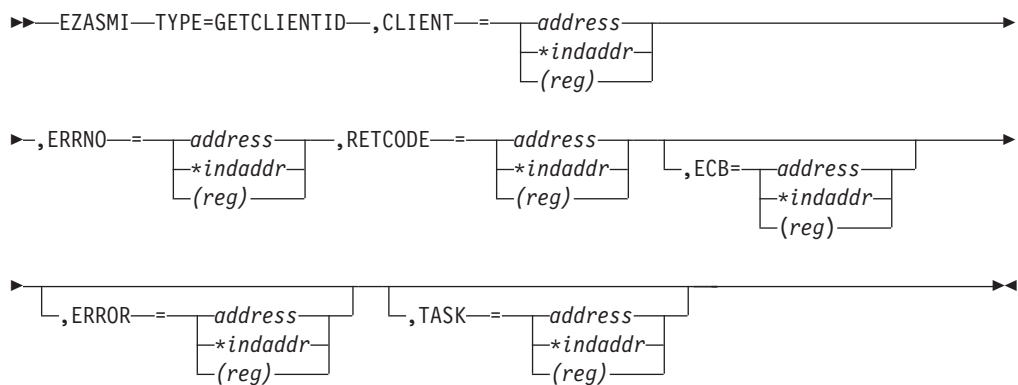
The GETCLIENTID macro returns the identifier by which the calling application is known to the TCP/IP address space.

The client ID structure returned is used by the GIVESOCKET and TAKESOCKET macros.

Important: IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply, if this program is used.

If GETCLIENTID is called by a server or client, the identifier of the calling application is returned.

Format



Parameters

CLIENT

A client-ID structure that describes the application that issued the call.

DOMAIN

A fullword binary number specifying the domain of the client. On input, this is an optional parameter for AF_INET, and a required parameter for AF_INET6 to specify the domain of the client. For TCP/IP, the value is a decimal 2 indicating AF_INET, or decimal 19 indicating AF_INET6. On output, this is the returned domain of the client.

NAME

An 8-byte character field. It is built with the partition's partition ID, which is left adjusted and padded with blanks.

TASK An 8-byte character field. This task identifier can be specified by the user with the INITAPI call or defaulted by the system (see the description of the INITAPI call for details).

RESERVED

Specifies 20-byte character reserved field. This field is required and internally used by TCP/IP.

ERRNO

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

See "ERRNO Values" on page 74 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

Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.
- A 156-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 .

ERROR

Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

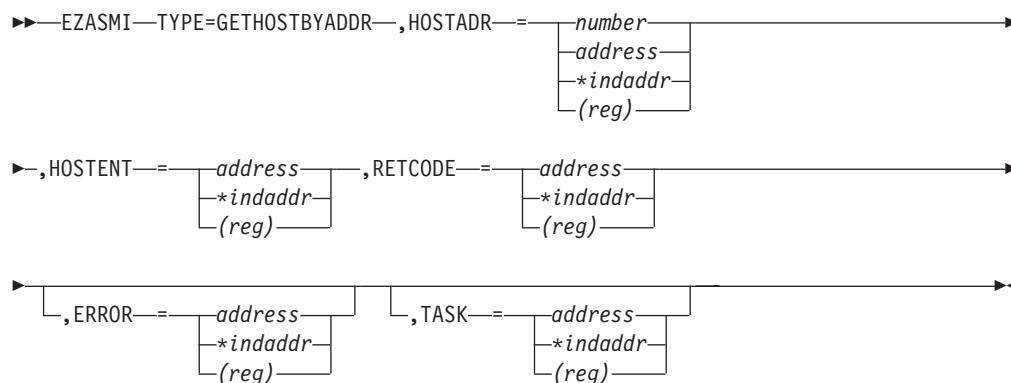
TASK Input parameter. The location of the task storage area in your program.

GETHOSTBYADDR

The GETHOSTBYADDR macro returns domain and alias names of the host whose internet address is specified by the macro.

A TCP/IP host can have multiple alias names and host internet addresses.

Format



Note: GETHOSTBYADDR and GETHOSTBYNAME all use the same static area to return the HOSTENT structure. This static area is only valid until the next one of these functions is called on the same thread or till TERMAPI.

Parameters

HOSTADR

Input parameter. A fullword unsigned binary field set to the internet address of the host whose name you want to find.

GETHOSTBYADDR

HOSTENT

Input parameter. A fullword word containing the address of the HOSTENT structure returned by the macro. For information about the HOSTENT structure, see Figure 20.

RETCODE

Output parameter. 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, if the application programming interface (API) processing module cannot be loaded.

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

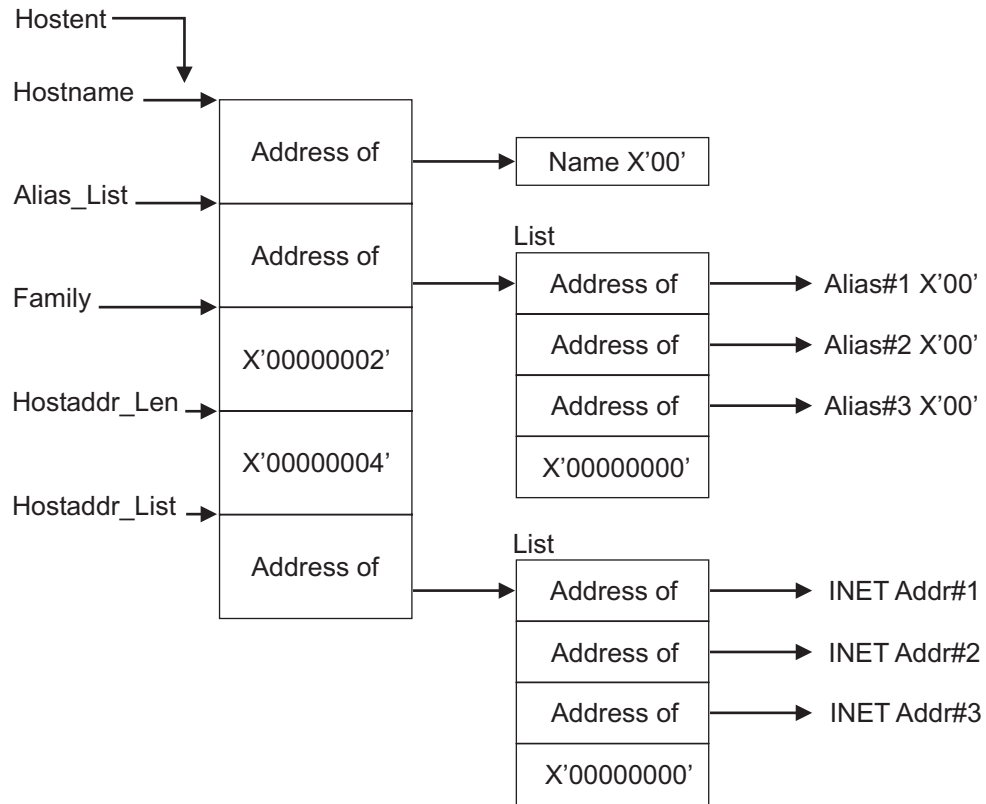


Figure 20. HOSTENT Structure Returned by the GETHOSTBYADDR Macro

GETHOSTBYADDR returns the HOSTENT structure shown in Figure 20. 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'

Note: Alias names are not supported.

- 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

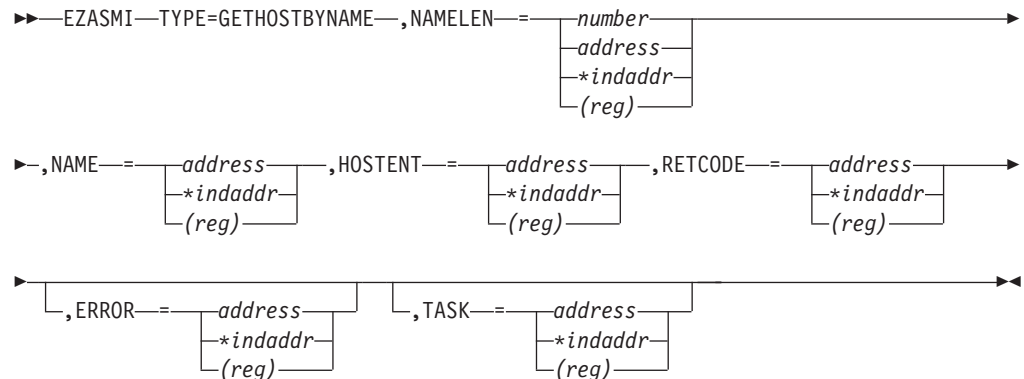
The GETHOSTBYNAME macro returns the alias names and the internet addresses of a host whose domain name is specified in the macro.

TCP/IP tries to resolve the host name through a name server, if one is present.

If a call is made to convert a symbolic name to an IP address, TCP/IP for VSE/ESA searches the local names table (created by DEFINE NAME) first. If this search fails, the name is passed to the specified DNS (set with SET DNSx). TCP/IP for VSE/ESA will try each DNS, beginning with DNS1, until a response is received or all servers have been polled. The first server to respond determines if the request succeeds or fails. If the search within a DNS fails, the default domain string (as specified with SET DEFAULT_DOMAIN) is appended to the name (following a period) and the DNS is consulted the last time for the name resolution.

If the host name is not found, the return code is -1.

Format



Note: GETHOSTBYADDR and GETHOSTBYNAME all use the same static area to return the **hostent** structure. This static area is only valid until the next one of these functions is called on the same thread or till TERMAPI.

Parameters

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.

GETHOSTBYNAME

NAME

A character string, up to 24 characters, set to a host name. This call returns the address of HOSTENT for this name.

HOSTENT

Output parameter. A fullword word containing the address of HOSTENT returned by the macro. For information about the HOSTENT structure, see Figure 21.

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, if the application programming interface (API) processing module cannot be loaded.

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

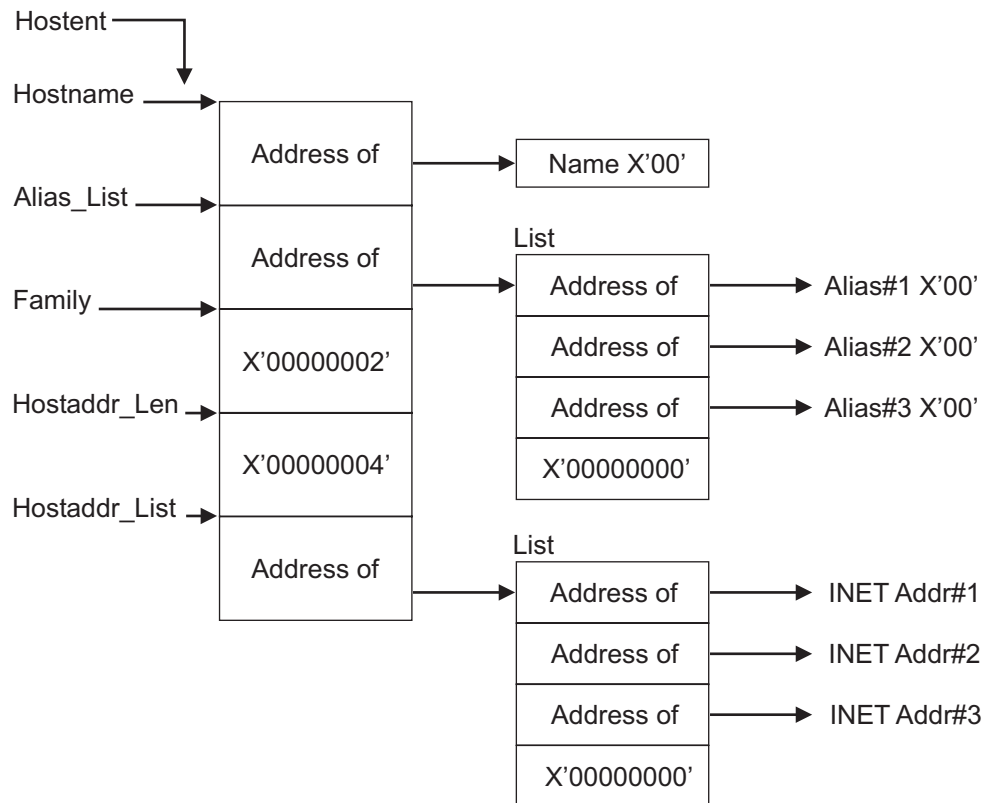


Figure 21. HOSTENT Structure Returned by the GETHOSTBYNAME Macro

GETHOSTBYNAME returns the HOSTENT structure shown in Figure 21. 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'.

Note: Alias names are not supported.

- 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'.

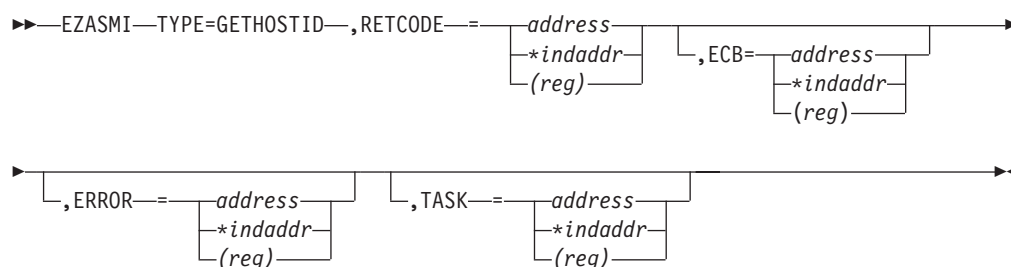
The HOSTENT structure uses indirect addressing to return a variable number of alias names and internet addresses.

GETHOSTID

The GETHOSTID macro returns the 32-bit identifier for the current host.

This value is the default home internet address.

Format



Parameters

RETCODE

Output parameter. Returns 32-bit internet address of the host. A -1 in RETCODE indicates an error. There is no ERRNO parameter for this macro.

ECB

Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.
- A 156-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.

ERROR

Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

TASK

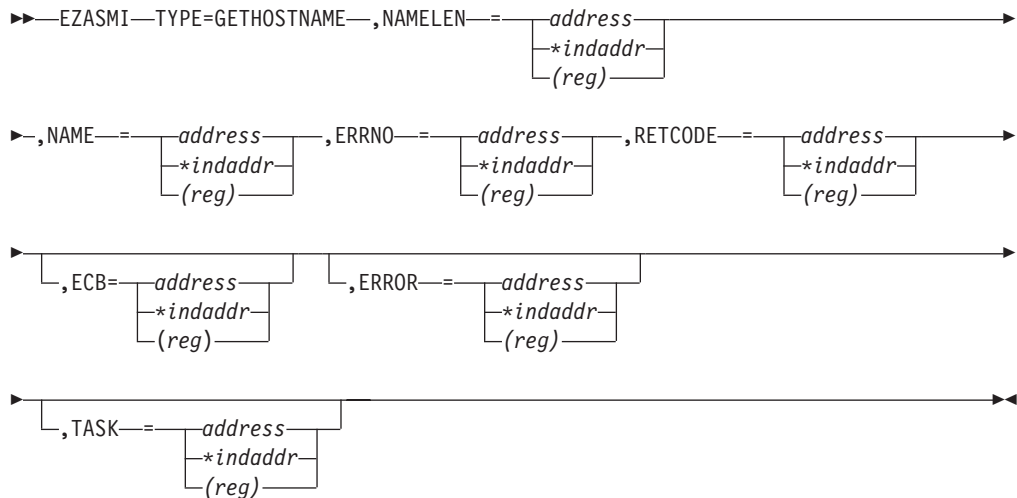
Input parameter. The location of the task storage area in your program.

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.

Format



Parameters

NAMELEN

Input and output 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 for VSE allows a maximum length of 64 characters. This field is filled with a host name the length returned in NAMELEN when the call completes.

ERRNO

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

See "ERRNO Values" on page 74 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

Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.
- A 156-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 .

ERROR

Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

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

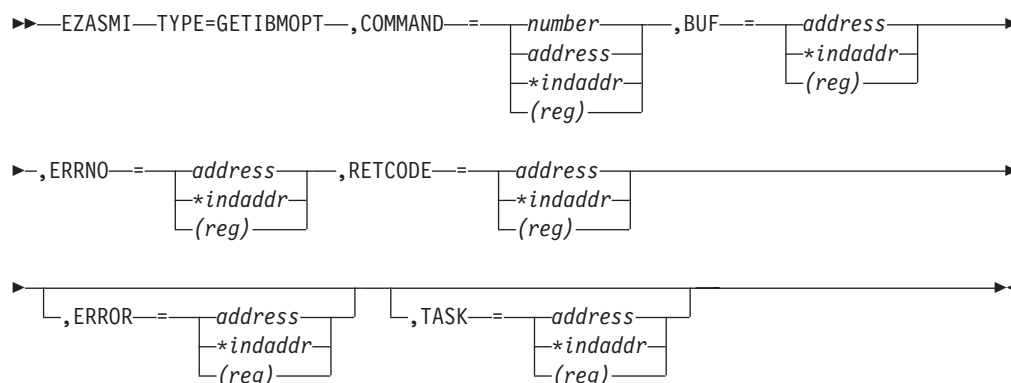
GETIBMOPT

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

With this information, the caller can dynamically choose the TCP/IP image with which to connect, using the INITAPI macro.

Important: This function call is not available with TCP/IP for VSE/ESA.

Format



Parameters

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.

Table 8. NUM_IMAGES Field Settings

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)

Table 8. NUM_IMAGES Field Settings (continued)

NUM_IMAGES (4 bytes)		
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)

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

ERRNO

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

See "ERRNO Values" on page 74 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, if the application programming interface (API) processing module cannot be loaded.

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

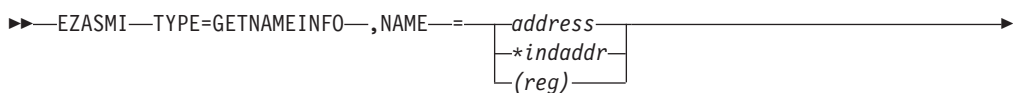
GETNAMEINFO

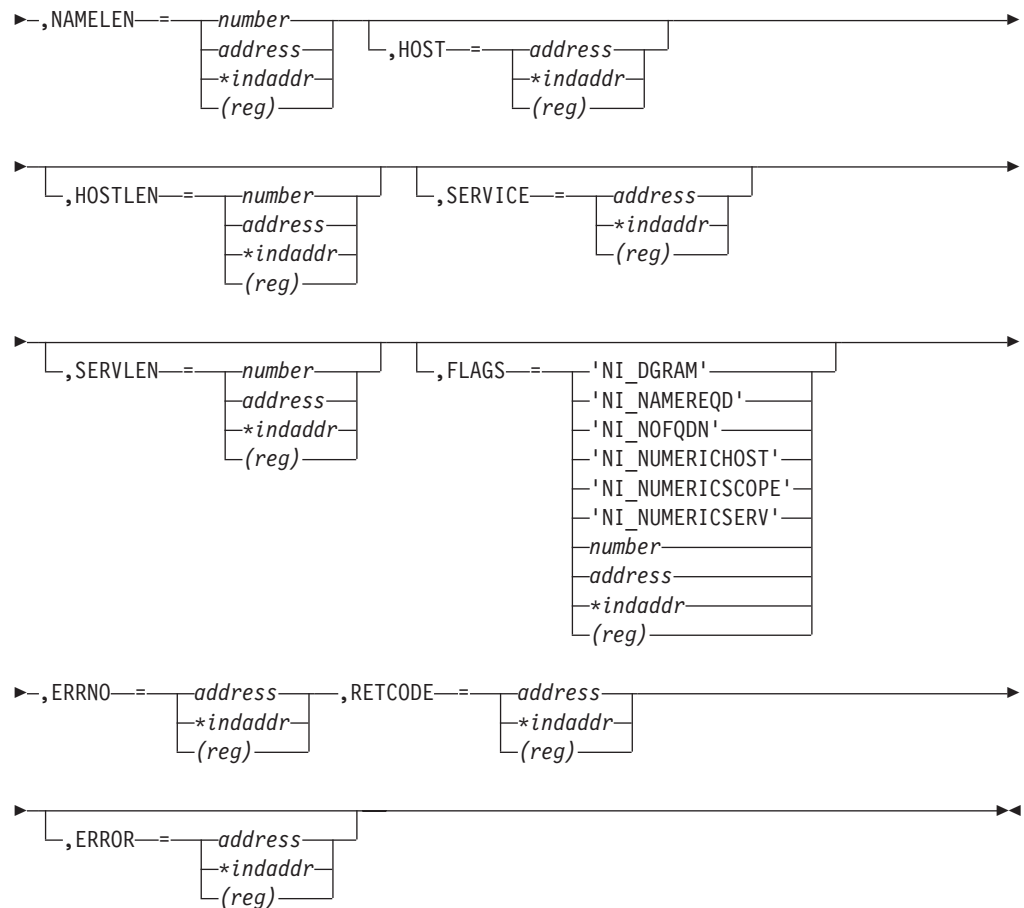
The GETNAMEINFO macro returns the node name and service location of a socket address that is specified in the macro.

On successful completion, GETNAMEINFO returns the node and service named, if requested, in the buffers provided.

Important: This function call is not available with TCP/IP for VSE/ESA.

Format





Parameters

NAME

An IPv4 or IPv6 socket address structure to be translated. Include the PRD1.MACLIB(EZBREHST) macro to get the assembler mappings for the socket address structure. The socket address structure mappings start 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:

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:

NAMELEN

A 1-byte binary field that specifies the length of the IPv6 socket

GETNAMEINFO

address structure. Any value specified by the use of this field is ignored, if processed as input. The field is set to 0, if processed as output.

FAMILY

A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary number that specifies the port number.

FLOW-INFO

This field is ignored by the TYPE=GETNAMEINFO macro.

IPv6-ADDRESS

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

SCOPE-ID

A fullword binary field that specifies the scope for an IPv6 address as an interface index. The resolver ignores the SCOPE_ID field, unless the address in IPv6-ADDRESS is a link-local address and the HOST parameter is also specified.

NAMELEN

An input parameter. A fullword binary field. The length of the socket address structure pointed to by the NAME argument.

HOST On input, storage capable of holding the returned resolved host name, which can be up to 255 bytes long, for the input socket address. If inadequate storage is specified to contain the resolved host name, the resolver returns the host name up to the storage specified and truncation might occur. If the host's name cannot be located, the numeric form of the host's address is returned instead of its name. However, if the NI_NAMEREQD option is specified and no host name is located, an error is returned. One or both of the following groups of parameters are required:

- The HOST and HOSTLEN parameters
- The SERVICE and SERVLN parameters

Otherwise, an error occurs. The HOST name being queried consists of up to HOSTLEN or up to the first binary 0.

If the IPv6-ADDRESS value is a link-local address, and the SCOPE_ID interface index is nonzero, scope information is appended to the resolved host name using the format *host%scope information*. The scope information can be the numeric form of the SCOPE_ID interface index or the interface name that is associated with the SCOPE_ID interface index. Use the NI_NUMERICSCOPE option to select which form is returned. The combined host name and scope information is 255 bytes or less.

HOSTLEN

A fullword binary field that contains the length of the host storage that is used to contain the returned resolved host name. If HOSTLEN is 0 on input, the resolved host name is not returned. The HOSTLEN value must be equal to or greater than the length of the longest host name, or hostname and scope information combination, to be returned. The TYPE=GETNAMEINFO returns the host name, or host name and scope information combination, up to the length specified by the HOSTLEN value. On output, HOSTLEN contains the length of the returned resolved host name, or host name and scope information combination. This is an

optional field, but if you specify this field, you also must code the HOST value. One or both of the following groups of parameters are required:

- The HOST and HOSTLEN parameters
- The SERVICE and SERVLLEN parameters

Otherwise, an error occurs.

SERVICE

On input, storage capable of holding the returned resolved service name, which can be up to 32 bytes long, for the input socket address. If inadequate storage is specified to contain the resolved service name, the resolver returns the service name up to the storage specified and truncation might occur. If the service name cannot be located, or if NI_NUMERICSERV was specified in the FLAGS operand, the presentation form of the service address is returned instead of its name. This is an optional field, but if you specify this field, you also must code the SERVLLEN parameter. The SERVICE name being queried consists of up to SERVLLEN or up to the first binary zero. One or both of the following groups of parameters are required:

- The HOST and HOSTLEN parameters
- The SERVICE and SERVLLEN parameters

Otherwise, an error occurs.

SERVLLEN

Initially an input parameter. A fullword binary field that contains the length of the SERVICE storage used to contain the returned resolved service name. If SERVLLEN is 0 on input, the service name information is not returned. SERVLLEN must be equal to or greater than the length of the longest service name to be returned. The TYPE=GETNAMEINFO returns the service name up to the length specified by SERVLLEN. On output, SERVLLEN contains the length of the returned resolved service name. This is an optional field, but if you specify it, you must also code the SERVICE parameter. One or both of the following groups of parameters are required:

- The HOST and HOSTLEN parameters
- The SERVICE and SERVLLEN parameters

Otherwise, an error occurs.

FLAGS

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.

GETNAMEINFO

Literal Value	Binary Value	Decimal Value	Description
'NI_DGRAM'	X'00000010'	16	Indicates that the service is a datagram service. The default behavior is to assume that the service is a stream service.
'NI_NUMERICSSCOPE'	X'00000020'	32	Only return the numeric form of the SCOPE-ID interface index, if applicable.

ERRNO

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

See “ERRNO Values” on page 74 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

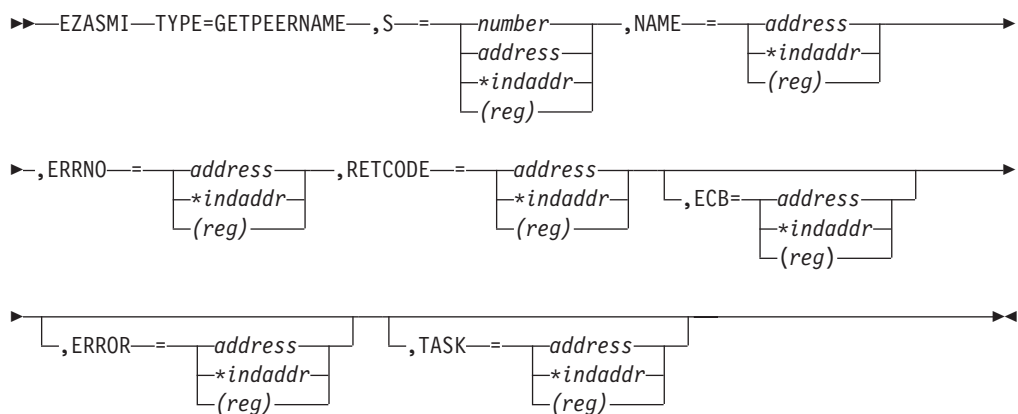
Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

GETPEERNAME

The GETPEERNAME macro returns the name of the remote socket to which the local socket is connected.

Important: IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply, if this program is used.

Format



Parameters

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 PRD1.MACLIB(EZBREHST) 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:

FAMILY

A halfword binary field specifying the IPv4 addressing family. For TCP/IP the value is a decimal 2, indicating AF_INET.

PORT A halfword binary field specifying the client's port number.

IPv4-ADDRESS

A fullword binary field specifying the 32-bit IPv4 Internet address.

RESERVED

Specifies eight bytes of binary zeros. This field is required, but is not used.

The IPv6 socket address structure contains the following fields:

NAMELEN

A 1-byte binary field that specifies the length of the IPv6 socket address structure. Any value specified by the use of this field is ignored, if processed as input. The field is set to 0, if processed as output.

FAMILY

A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary 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 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 "ERRNO Values" on page 74 for information about ERRNO return codes.

GETPEERNAME

RETCODE

Output parameter. A fullword binary field.

Value Description

- 0 Successful call
- 1 Check ERRNO for an error code

ECB Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.
- A 156-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.

ERROR

Input parameter. The location in your program to receive control, if 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.

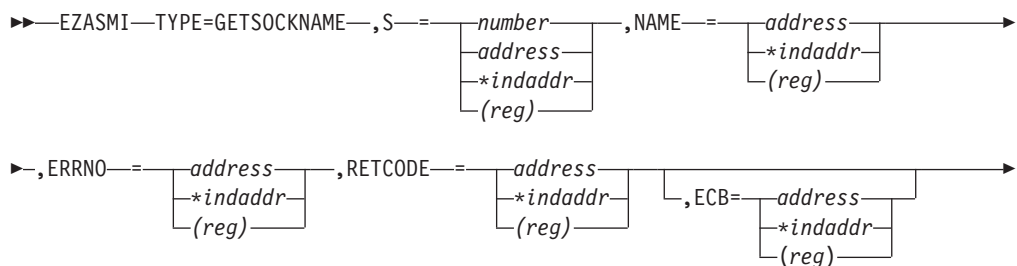
If the socket is not bound to an address, the macro returns with the FAMILY field completed and the rest of the structure set to zeros.

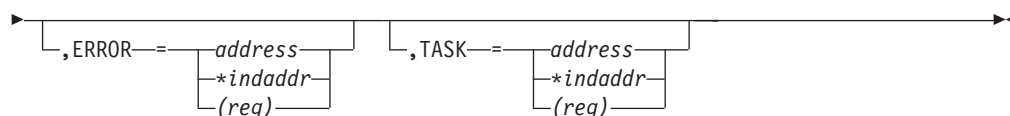
Stream sockets are not assigned a name until after a successful call to BIND, CONNECT, or ACCEPT.

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.

Important: IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply, if this program is used.

Format





Parameters

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 PRD1.MACLIB(EZBREHST) 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:

FAMILY

A halfword binary field specifying the IPv4 addressing family. For TCP/IP the value is a decimal 2, indicating AF_INET.

PORT A halfword binary field specifying the port number bound to this socket. If the socket is not bound, a zero is returned.

IPv4-ADDRESS

A fullword binary field specifying the 32-bit IPv4 Internet address.

RESERVED

Specifies eight bytes of binary zeros. This field is required, but is not used.

The IPv6 socket address structure contains the following fields:

NAMELEN

A 1-byte binary field that specifies the length of the IPv6 socket address structure. Any value specified by the use of this field is ignored, if processed as input. The field is set to 0, if processed as output.

FAMILY

A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary field specifying the port number bound to this socket. If the socket is not bound, a zero is returned.

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 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

GETSOCKNAME

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 "ERRNO Values" on page 74 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 Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.
- A 156-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.

ERROR

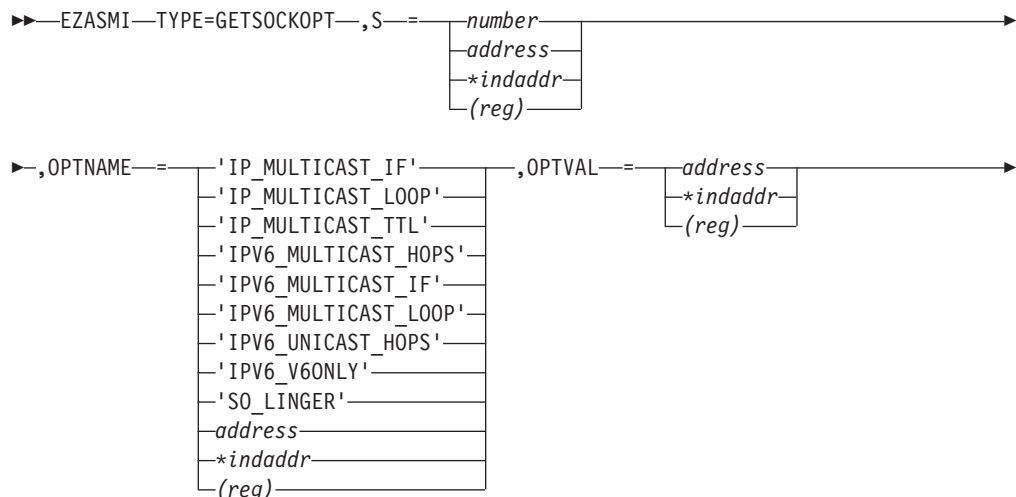
Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

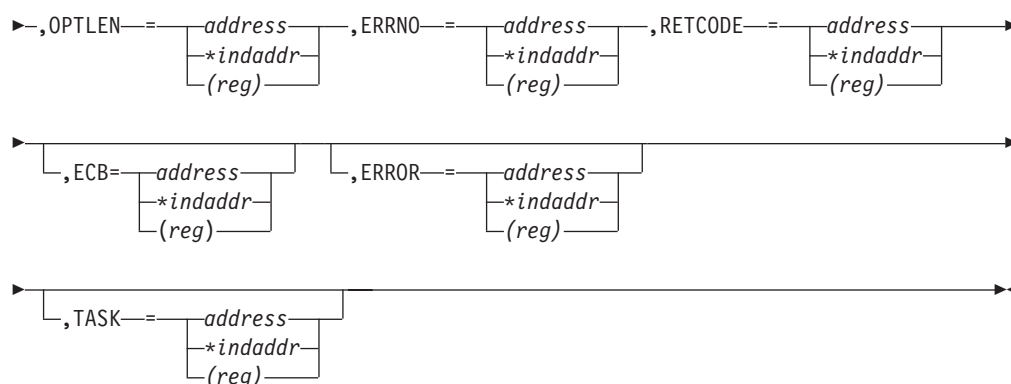
TASK Input parameter. The location of the task storage area in your program.

GETSOCKOPT

The GETSOCKOPT macro gets the options associated with a socket that were set using the "SETSOCKOPT" on page 371 macro. The options for each socket are described by the following parameters. You must specify the option that you want, if you issue the GETSOCKOPT macro.

Format





Parameters

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. Set OPTNAME to one of the following options before you issue GETSOCKOPT.

IP_MULTICAST_IF

Use this option to 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.

IP_MULTICAST_LOOP

Use this option to 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.

IP_MULTICAST_TTL

Use this option to 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.

IPV6_MULTICAST_HOPS

Use this option to obtain the hop limit used for outgoing multicast packets. This is an IPv6-only socket option.

IPV6_MULTICAST_IF

Use this option to obtain the index of the IPv6 interface used for sending outbound multicast datagrams from the socket application. This is an IPv6-only socket option.

IPV6_MULTICAST_LOOP

Use this option to determine whether a multicast datagram is looped back on the outgoing interface by the IP layer for local delivery, if 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.

GETSOCKOPT

IPV6_UNICAST_HOPS

Use this option to obtain the hop limit used for outgoing unicast IPv6 packets. This is an IPv6-only socket option.

IPV6_V6ONLY

Use this option to 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.

SO_LINGER

Requests the status of `SO_LINGER`.

- If the `SO_LINGER` option is enabled, and data transmission has not been completed, a `CLOSE` macro blocks the calling program until the data is transmitted or until the connection has timed out.
- If `SO_LINGER` is not enabled, a `CLOSE` call returns without blocking the caller and TCP/IP continues to try the send data function. Normally the send data function completes and the data is sent, but it cannot be guaranteed because TCP/IP can timeout before the send has been completed.

OPTVAL

Output parameter.

- If `SO_LINGER` is specified in `OPTNAME`, the following structure is returned:

ONOFF	DS	F
LINGER	DS	F

- A nonzero value returned in `ONOFF` indicates that the option is enabled and a zero value indicates that it is disabled.
- The `LINGER` value indicates the time in seconds that TCP/IP continues to try to send the data after the `CLOSE` call is issued. For information about how to set the `LINGER` time, see “`SETSOCKOPT`” on page 270.

OPTLEN

Input parameter. A fullword binary field containing the length of the data returned in `OPTVAL`.

- For `SO_LINGER`, `OPTVAL` contains two fullwords and `OPTLEN` is set to 8 (two fullwords).

ERRNO

Output parameter. A fullword binary field. If `RETCODE` is negative, this field contains an error number. See “`ERRNO Values`” on page 74 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 <code>ERRNO</code> for an error code |

ECB Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.

- A 156-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.

ERROR

Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

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

GIVESOCKET

The GIVESOCKET macro makes the socket available for a TAKESOCKET macro issued by another program.

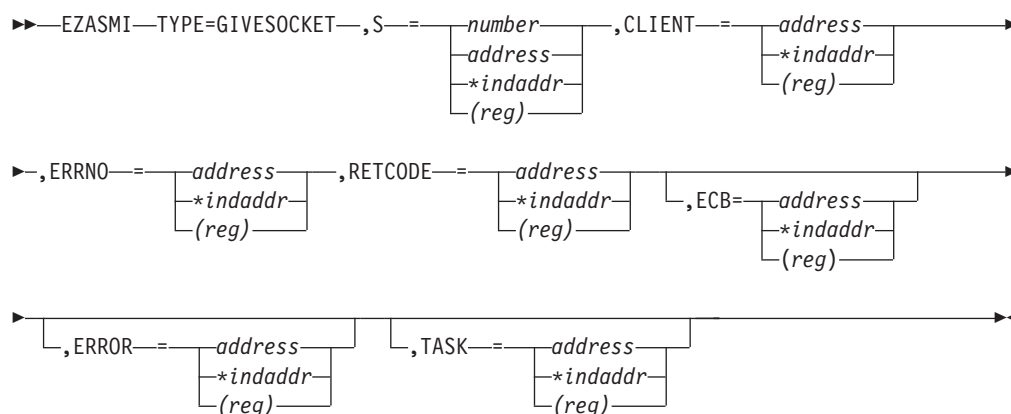
The GIVESOCKET macro can specify any connected stream socket. Typically, the GIVESOCKET macro is issued by a concurrent server program that creates sockets to be passed to a subtask.

After a program has issued a GIVESOCKET macro for a socket, it can only issue a CLOSE macro for the same socket.

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 CICS Listener program is an example of a concurrent server.

Important: IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply, if this program is used.

Format



Parameters

S Input parameter. A value, or the address of a halfword binary number specifying the descriptor of the socket to be given.

GIVESOCKET

CLIENT

A structure containing the identifier of the application to which the socket should be given.

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

Specifies an 8-character field, left-justified, padded to the right with blanks set to the address space name of the application (partition ID) going to take the socket. If this field is left blank, any z/VSE partition can take the socket.

TASK Specifies an eight-character field that can be set to blanks, or to the identifier of the socket-taking VSE subtask. If this field is set to blanks, any subtask in the partition specified in the NAME field can take the socket.

RESERVED

A 20-byte reserved field. This field is required, but only used internally.

ERRNO

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

See "ERRNO Values" on page 74 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 Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.
- A 156-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.

ERROR

Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

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

GSKFREEMEM

This function frees memory passed to the application on a previous call to an SSL function.

Format

```

▶▶—EZASMI—TYPE=GSKFREEMEM,—AREA=—address—,RETCODE=—address—,————▶
      |*indaddr|
      |(reg)|
      |*indaddr|
      |(reg)|

▶—ERRNO=—address————▶▶
      |*indaddr|
      |(reg)|
  
```

Parameters

AREA Input parameter. Specifies the address of the memory, returned to the application from a previous SSL call that is to be freed.

RETCODE

Output parameter. A value of 0 indicates the successful completion of the function. If RETCODE is negative, an error has occurred.

ERRNO

Output parameter. May show detailed error information.

Note: The distinguished name returned in the null-terminated string by the GSKGETDNBYLAB function must be freed using GSKFREEMEM.

GSKGETCIPHINF

This function requests cipher related information for SSL for VSE.

This information determines the encryption level that the system can support and returns a list of cipher specifications that SSL can use. This allows an application to determine, at run time, the level of SSL encryption that the installed application can request.

Format

```

▶▶—EZASMI—TYPE=GSKGETCIPHINF—,—CIPHLEVEL=—number—,————▶
      |address|
      |*indaddr|
      |(reg)|

▶—SECLEVEL=—address—,RETCODE=—address—,—ERRNO=—address—▶▶
      |*indaddr|
      |(reg)|
      |*indaddr|
      |(reg)|
      |*indaddr|
      |(reg)|
  
```

Parameters

CIPHLEVEL

Input Parameter. A value, or the address of a fullword binary number that determines the type of cipher information to be returned. Valid values are

- 1 only exportable cipher information is to be returned (GSK_LOW_SECURITY)

GSKGETCIPHINF

- 2 exportable and domestic cipher information is to be returned (GSK_HIGH_SECURITY)

SECLEVEL

Output Parameter. Point to an 104 byte area (to be allocated by the application) where the system returns the following information:

4 bytes

System SSL version (always 3 for GSK_VERSION3)

64 bytes

A character string (terminated with x00) with the SSL Version 3 cipher specs allowed for use on the system (these are passable on the V3CIPHER parameter on the GSKSSOCINIT call).

32 bytes

This field will always be filled with binary zeros because SSL for VSE does not support SSL Version 2 cipher specs.

4 bytes

One of the following

- 1 GSK_SEC_LEVEL_US
- 2 GSK_SEC_LEVEL_EXPORT
- 3 GSK_SEC_LEVEL_EXPORT_FR

RETCODE

Output Parameter. A value of 0 indicates the successful completion of the function. If RETCODE is not 0, an error occurred (please refer to VSE library member SSLVSE.A or to the *TCP/IP for VSE 1.5 Optional Features* for a detailed description of error return codes).

ERRNO

Output Parameter. May show detailed error information.

GSKGETDNBYLAB

This function returns the complete distinguished name for a key based on the label the key has in the key database file.

This value can be used for the DNAME field in the GSKSSOCINIT call.

Format

►►—EZASMI—TYPE=GSKGETDNBYLAB—, —————►
►—KEYLABEL=—

address
*indaddr
(reg)

, RETCODE=—

address
*indaddr
(reg)

, —ERRNO=—

address
*indaddr
(reg)

—►

Parameters

KEYLABEL

Input Parameter. Point to a character string that contains the label for the key in the key database file. The string must be terminated with x00.

RETCODE

Output parameter. A value greater 0 indicates the successful completion of the function and denotes a pointer to the character string with the distinguished name. A value of 0 indicates an error.

ERRNO

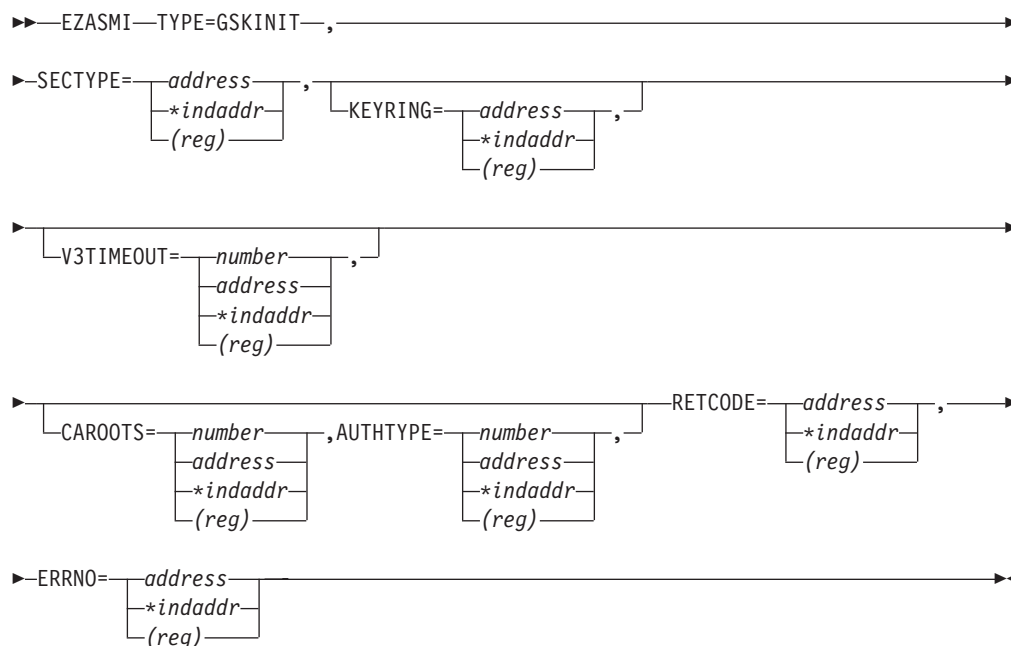
Output parameter. May show detailed error information.

Note: The distinguished name returned in the null-terminated string must be freed using the GSKFREEMEM function.

GSKINIT

This function sets the overall SSL for VSE environment for the current partition.

After the function completes successfully, the application is ready to call SSL for VSE interfaces and to create and use secure socket connections.

Format**Parameters****SECTYPE**

Input Parameter. Point to a character string that identifies the minimum acceptable security protocol. The value must be entered in upper case characters and terminated with x00. Valid values are (without double-quotes):

- "SSL30" for SSL Version 3.0
- "TLS31" for TLS Version 1.0 (not supported for client applications)

KEYRING

(Optional) Input Parameter. Point to a character string specifying the "lib.sublib" where the private key and certificates are stored. This string must be terminated with x00. If this parameter is used, the GSKGETDNBYLAB call must be used later on to identify the library member name that is specified in DNAME parameter of the GSKSSOCINIT call. If this parameter is not specified, the default "SSL for VSE" files as defined in procedure \$SSL4VSE.PROC are used (for details refer to the manual *TCP/IP for VSE 1.5 Optional Features*).

V3TIMEOUT

(Optional) Input Parameter. A value, or the address of a fullword binary number that specifies the number of seconds for the SSL V3 session identifier to expire. The valid range is 0 - 86400 (1 day). If this parameter is not specified, a default value of 86400 is assumed.

CARROOTS

(Optional) Input Parameter. A value, or the address of a fullword binary number that specifies which CA (Certificate Authority) root to use for certificate verification. The supported values are:

- 0 Use the CA roots from the local key database file for certificate verification.
- 1 Allow client authentication with certificates issued by the same certificate authority as VSE.

If this parameter is not specified, a default value of 0 is assumed.

AUTHTYPE

(Optional) Input Parameter. A value, or the address of a fullword binary number that specifies the method to use for verifying the client's certificate. This field is mandatory, if the CARROOTS field is set to 1. It is ignored, if CARROOTS is set to 0. The supported values are:

- 0 the client's certificate is verified using the local key database file.
- 1 the same meaning as with value 0
- 2 the same meaning as with value 0
- 3 the client's certificate is not verified.

RETCODE

Output Parameter. A value of 0 indicates the successful completion of the function. If RETCODE is not 0, an error occurred (please refer to VSE library member SSLVSE.A or to the *TCP/IP for VSE 1.5 Optional Features* for a detailed description of error return codes.).

ERRNO

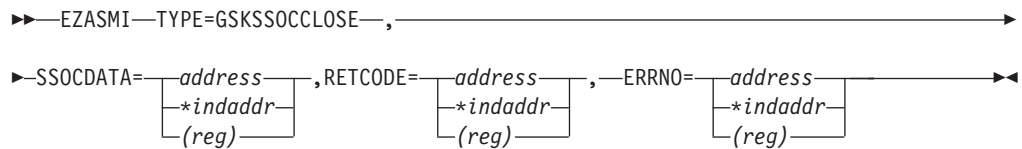
Output Parameter. May show detailed error information.

Note: Subsequent calls for GSKINIT without corresponding GSKUNINIT calls in between will be rejected.

GSKSSOCCLOSE

This function ends a secure socket connection and frees all SSL for VSE resources for that connection.

Format



Parameters

SSOCDATA

Input Parameter. Pointer to GSKSOCDATA as returned in RETCODE by EZASMI TYPE=GSKSSOCINIT.

RETCODE

Output parameter. A value of 0 indicates the successful completion of the function. If RETCODE is negative, an error has occurred (please refer to VSE library member SSLVSE.A or to the *TCP/IP for VSE 1.5 Optional Features* for a detailed description of error return codes).

ERRNO

Output parameter. May show detailed error information.

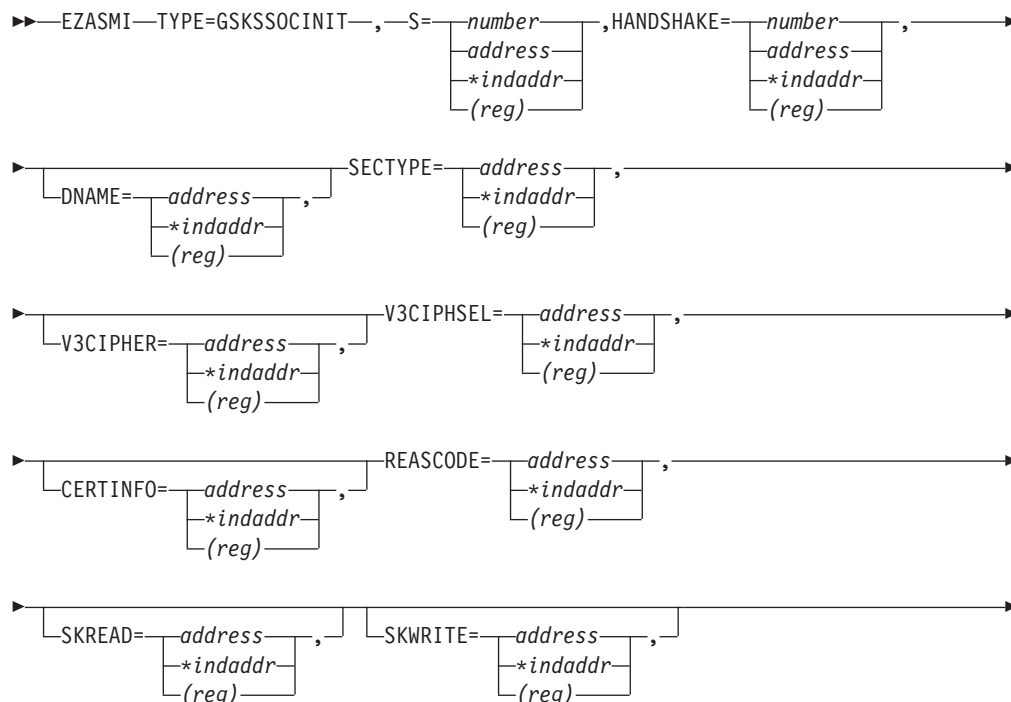
GSKSSOCINIT

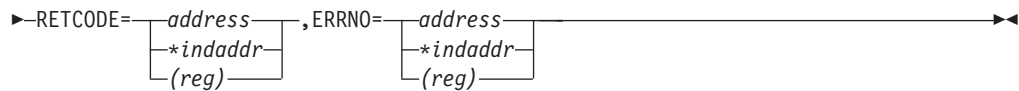
This function initializes the data areas necessary for SSL for VSE to initiate or accept a secure socket connection.

After the function is completed successfully, a pointer to a secured socket control block (in the following referred to as GSKSOCDATA) is returned to the application. Other calls using this secure socket connection must use this pointer.

During the call a complete handshake is performed based on the input specified with the GSKSSOCINIT call. While SSL for VSE performs the mechanics of the SSL handshake, "normal" RECV and SEND routines (either provided by the EZAAPI processing environment or provided by the application with the SKREAD and SKWRITE parameters) will be used to transport the SSL data during the SSL handshake, as well as for all subsequent read/write operations.

Format





Parameters

S Input Parameter. A value, or the address of a halfword binary number specifying the socket descriptor of the socket which is to be initialized for a secure socket connection.

HANDSHAKE

Input parameter. A value, or the address of a fullword binary number that specifies how the handshake is performed:

- 0** Perform the SSL handshake as a client (GSK_AS_CLIENT).
- 1** Perform the SSL handshake as a server (GSK_AS_SERVER).
- 2** Perform the SSL handshake as a server that requires client authentication (GSK_AS_SERVER_WITH_CLIENT_AUTH).
- 3** Perform the SSL handshake as a client without authentication (GSK_AS_CLIENT_NO_AUTH).

DNAME

(Optional) Input Parameter. Point to a character string that is the Distinguished name or label of the desired entry (certificate) in the key database file. This character string must be terminated with x00. To use the default key database file entry, omit the parameter. The distinguished name for a key database file entry may be determined via the EZASMI TYPE=GETDNBYLAB function call.

SECTYPE

Output Parameter. Point to a fullword where the address of a character string is stored that identifies the minimum acceptable security protocol. The character string is terminated with x00. Valid values are (without double-quotes):

- "SSL30" for SSL Version 3.0
- "TLS31" for TLS Version 1.0

V3CIPHER

(Optional) Input Parameter. Points to a character string that contains the list of SSL Version 3.0 ciphers in order of usage preference. Valid values as supported by TCP/IP for VSE are:

- 01 for RSA512_NULL_MD5
- 02 for RSA512_NULL_SHA
- 08 for RSA512_DES40CBC_SHA
- 09 for RSA1024_DESCBC_SHA
- 0A for RSA1024_3DESCBC_SHA
- 62 for RSA1024_EXPORT_DESCBC_SHA

You can use any combination of these values in any order. The list of values must be terminated with x00. The exportable cipher suites 01,02,08,62 can only be used with SSL30, and will not work with TLS31. To use the default SSL V3 cipher specs (which is 0A0908) omit this parameter.

V3CIPHSEL

Output parameter. Point to a 2-byte area (provided by the application) where the architected SSL Version 3.0 cipher spec value selected for this session is stored (for example: x0009).

CERTINFO

(Optional) Output parameter. Point to a fullword where the address of the Distinguished Name components from the client's certificate is stored. This parameter is only valid, if client authentication is requested for a server using SSL. The layout of this area is as follows:

- 4 bytes** Pointer to base64 certificate body
- 4 bytes** Length of base64 certificate body
- 4 bytes** Pointer to session ID for this connection
- 4 bytes** Flag to indicate if new session
- 4 bytes** Pointer to certificate serial number
- 4 bytes** Pointer to common name of client
- 4 bytes** Pointer to locality
- 4 bytes** Pointer to state or province
- 4 bytes** Pointer to country
- 4 bytes** Pointer to organization
- 4 bytes** Pointer to organizational unit
- 4 bytes** Pointer to issuer's common name
- 4 bytes** Pointer to issuer's locality
- 4 bytes** Pointer to issuer's state or province
- 4 bytes** Pointer to issuer's country
- 4 bytes** Pointer to issuer's organization
- 4 bytes** Pointer to issuer's organizational unit

SKREAD

(Optional) Input parameter. Point to an application-provided routine that performs a socket read function for SSL for VSE. This routine must fulfill the following requirements:

- It must use the EZASMI READ or RECV call for the actual read.
- It must use an own TIE (Task Interface Element) which is in its first bytes (use the TIECLLEN equate from the EZASMI TYPE=TASK,STORAGE=DSECT/CSECT macro) copied from the TIE used with the GSK calls.

If this parameter is not provided, a "read" routine provided by the EZAAPI processing environment will be used.

Example:

Main Routine

```

=====
.....
EZASMI TYPE=GSKSSOCINIT,      Issue GSKSSOCINIT call      x
      .....
      SKREAD=*SKREADA,      SKREAD routine                x
      .....
SKREADA DC      V(T9SKREAD)
MAINTIE EZASMI TYPE=TASK,STORAGE=CSECT      Task Interface Element
      ENTRY MAINTIE
.....

```

Sub Routine (to be linked to main routine)

```

=====
T9SKREAD START X'00'
T9SKREAD AMODE ANY
T9SKREAD RMODE ANY
      STM      R14,R12,12(R13)      Save Caller's Registers
      LR       R3,R15              Change base register to R3
      USING   T9SKREAD,R3
      LR       R11,R1             Save addr of parameter list
* *****
* Allocate our working storage
* *****
      LA       R0,T9SKRDYL         Load the length
      GETVIS  ADDRESS=(1),LENGTH=(0)
      LTR     R15,R15             Test the return code
      BZ      T9SKR010           ok
      SLR     R15,R15            not ok
      BCTR   R15,0              Set bad return code
      B       T9SKR090           Back to caller
T9SKR010 ST     R13,4(,R1)        Save caller's reg 13
      ST     R1,8(,R13)         Save our save area address
      LR     R13,R1             Load our save area address
      USING  T9SKRDYN,R13       Base or work area
* *****
* Process request
* *****
      L       R6,AMTIE          Load addr of main TIE
      MVC     T9SKRTIE(TIECLN),0(R6) Copy first 24 bytes
      L       R6,0(R11)         Get addr of socket descriptor
      MVC     T9RSOCK,0(R6)      Move to local field
      L       R6,4(R11)         Get addr of buffer
      ST     R6,T9RBUFA         Move to local field
      L       R6,8(R11)         Get length of buffer
      MVC     T9RBUFL,0(R6)      Move to local field
      EZASMI TYPE=READ,          Read request
      S=T9RSOCK+2,              for this socket descriptor
      BUF=*T9RBUFA,            to this buffer
      NBYTE=T9RBUFL,          with this length
      TASK=T9SKRTIE,          own task storage
      ERRNO=T9RERRNO,         own ERRNO
      RETCODE=T9RRETCD        own RETCODE
      L       R15,T9RRETCD      Move RETCODE to reg 15
T9SKR090 L     R13,4(,R13)      Load caller's reg 13
      L     R14,12(R13)
      LM     R0,R12,20(R13)
      BR     R14              Back to caller
* --- Constants -----
AMTIE DC      V(MAINTIE)        Address of main TIE
      EZASMI TYPE=TASK,STORAGE=DSECT TIE DSECT
* --- Dynamic work area -----
T9SKRDYN DSECT                Dynamic Storage for this module
T9SKRSV DS     18F             Own savearea (MUST: begin of dyn)
T9SKRTIE DS     XL(TIELENTH)   Own TIE
T9RSOCK DS     F              Socket descriptor

```


T9RBUFA	DS	F	Addr of read buffer
T9RBUFL	DS	F	Length of read buffer
T9RERRNO	DS	F	Addr of errno value
T9RRETC	DS	F	Addr of retcode value
T9SKRDYL	EQU	*-T9SKRDYN	Length of dynamic storage
*-----			
R0	EQU	0	
R1	EQU	1	
R2	EQU	2	
R3	EQU	3	
R4	EQU	4	
R5	EQU	5	
R6	EQU	6	
R7	EQU	7	
R8	EQU	8	
R9	EQU	9	
R10	EQU	10	
R11	EQU	11	
R12	EQU	12	
R13	EQU	13	
R14	EQU	14	
R15	EQU	15	
*	END	T9SKREAD	

SKWRITE

(Optional) Input parameter. Point to an application-provided routine that performs a socket write function for SSL for VSE. This routine must fulfill the following requirements:

- It must use the EZASMI WRITE or SEND call for the actual write.
- It must use an own TIE (Task Interface Element) which is in its first bytes (use the TIECLLEN equate from the EZASMI TYPE=TASK,STORAGE=DSECT/CSECT macro) copied from the TIE used with the GSK calls.

If this parameter is not provided, a "write" routine provided by the EZAAPI processing environment will be used.

Example:

Similar to the SKREAD example.

REASCODE

Output parameter. Point to a fullword where the failure reason code for the GSKSSOCINIT call is stored. A value of 0 indicates the successful completion of the function.

RETCODE

Output parameter. If REASCODE is 0, the RETCODE parameter contains the pointer to a GSKSOCDATA structure, which needs to be used in subsequent SSL for VSE operations.

ERRNO

Output parameter. May show detailed error information.

GSKSSOCREAD

This function receives data on a secure socket connection.

Format

►►—EZASMI—TYPE=GSKSSOCREAD—,—SSOCDATA=— $\boxed{\text{address}}$ —, BUF=— $\boxed{\text{address}}$ —, —►►
 $\boxed{\text{*indaddr}}$ $\boxed{\text{*indaddr}}$
 $\boxed{\text{(reg)}}$ $\boxed{\text{(reg)}}$

GSKSSOCREAD

►-NBYTE=

number
address
*indaddr
(reg)

, RETCODE=

address
*indaddr
(reg)

, -ERRNO=

address
*indaddr
(reg)

◄

Parameters

SSOCDATA

Input parameter. Address of GSKSOCDATA as returned in RETCODE by EZASMI TYPE=GSKSSOCINIT.

BUF Input parameter. The address of the user-supplied buffer in which the data is to be stored.

NYBTE

Input parameter. A value, or the address of a fullword binary number specifying the length of the data buffer. The length of the data buffer must be either 64Kb or at least 32 bytes larger than the largest send buffer that is to be received.

RETCODE

Output parameter. A value of 0 or greater 0 indicates the successful completion of the function and denotes the number of bytes which have been read. If RETCODE is negative, an error has occurred (please refer to VSE library member SSLVSE.A or to the *TCP/IP for VSE 1.5 Optional Features* for a detailed description of error return codes).

ERRNO

Output parameter. May show detailed error information. For nonblocking sockets, if no data is received, the GSKSSOCREAD may return with ERRNO set to EWOULDBLOCK.

GSKSSOCRESET

This function refreshes the security parameters, such as encryption keys, for a session.

Format

►-EZASMI—TYPE=GSKSSOCRESET—, ◄

►-SSOCDATA=

address
*indaddr
(reg)

, RETCODE=

address
*indaddr
(reg)

, -ERRNO=

address
*indaddr
(reg)

◄

Parameters

SSOCDATA

Input parameter. Address of GSKSOCDATA as returned in RETCODE by EZASMI TYPE=GSKSSOCINIT.

RETCODE

Output parameter. A value of 0 indicates the successful completion of the function. If RETCODE is negative, an error has occurred (please refer to VSE library member SSLVSE.A or to the *TCP/IP for VSE 1.5 Optional Features* for a detailed description of error return codes).

ERRNO

Output parameter. May show detailed error information.

GSKSSOCWRITE

This function sends data on a secure socket connection.

Format

```

▶▶EZASMI—TYPE=GSKSSOCWRITE—,—SSOCDATA=—address—,BUF=—address—,
      |*indaddr| |*indaddr|
      |(reg)|   |(reg)|
▶▶NBYTE=—number—,RETCODE=—address—,—ERRNO=—address—
      |address| |*indaddr| |*indaddr|
      |*indaddr| |(reg)|   |(reg)|
      |(reg)|
  
```

Parameters

SSOCDATA

Input parameter. Address of GSKSSOCDATA as returned in RETCODE by EZASMI TYPE=GSKSSOCINIT.

BUF Input parameter. The address of the data being transmitted.

NYBTE

Input parameter. A value, or the address of a fullword binary number specifying the number of bytes to be transmitted.

RETCODE

Output parameter. A value of 0 or greater 0 indicates the successful completion of the function and denotes the number of bytes which have been sent. If RETCODE is negative, an error has occurred (please refer to VSE library member SSLVSE.A or to the *TCP/IP for VSE 1.5 Optional Features* for a detailed description of error return codes).

ERRNO

Output parameter. May show detailed error information.

GSKUNINIT

The GSKUNINIT call removes the current overall settings for the SSL environment. It removes fields such as session timeout values and SSL protocols.

Format

```

▶▶EZASMI—TYPE=GSKUNINIT—,—RETCODE=—address—,ERRNO=—address—
      |*indaddr| |*indaddr|
      |(reg)|   |(reg)|
  
```

Parameters

RETCODE

Output Parameter. A value of 0 indicates the successful completion of the function. If RETCODE is not 0, an error occurred (please refer to VSE library member SSLVSE.A or to the *TCP/IP for VSE 1.5 Optional Features* for a detailed description of error return codes.).

ERRNO

Output Parameter. May show detailed error information.

INITAPI

The INITAPI macro connects an application to the TCP/IP interface.

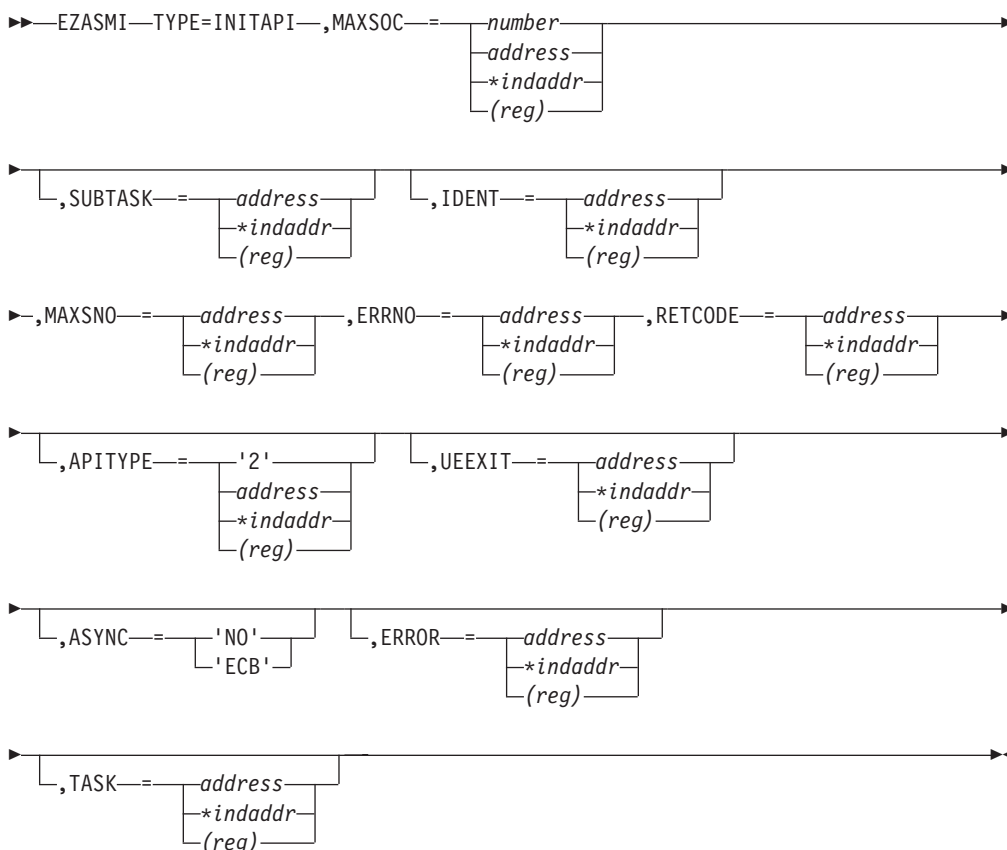
Almost all sockets programs that are written in COBOL, PL/I, or assembler language must issue the INITAPI macro before they issue other sockets macros.

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, generate a default INITAPI call:

- GETCLIENTID
- GETHOSTID
- GETHOSTNAME
- SELECT
- SELECTEX
- SOCKET
- TAKESOCKET

Format



Parameters

MAXSOC

Input parameter. A halfword binary field specifying the maximum number

of sockets that are supported for this application. Currently, TCP/IP for VSE/ESA ignores this input and defaults the maximum number of sockets that are supported to 8192. Socket descriptor numbers are in the range 0 – 8191.

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 ensures that, if you issue more than one INITAPI command from the same address space, each SUBTASK parameter is unique. If not specified or specified as 8 blanks, a default subtask name is used. In a batch environment, we have

byte 0-2

first 3 characters of the JOBNAME

byte 3

hex F0

byte 4-7

the VSE Task Identifier

In a CICS transaction environment we have

byte 0-2

the CICS EIBTRNID (transaction identifier)

byte 3 hex F1

byte 4-7

the CICS EIBTASKN (task number)

IDENT

A structure that contains 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

Starting with z/VSE 4.2, this parameter can be used to select the local TCP/IP stack used with this application. This 8-byte parameter can be set to "SOCKET nn " or just to " nn " (left- or right-adjusted, padded with six blanks). The value " nn " determines the ID of the selected TCP/IP stack as it is specified with the ID parameter in the TCP/IP startup JCL.

ADSNAME

The parameter can be used to specify the name of the TCP/IP Interface Routine used by the EZA processing environment. If nothing is specified here, the TCP/IP Interface Routine EZASOH99, supplied by IBM, is used. Note that this specification can be overwritten with the following JCL statement: // SETPARM [SYSTEM,] EZA\$PHA='routine-name'.

MAXSNO

Output parameter. A fullword binary field that contains the greatest descriptor number that can get assigned to this application. Currently, TCP/IP for VSE/ESA always returns 8191.

ERRNO

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

See "ERRNO Values" on page 74, 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.

APITYPE

Optional input parameter. A halfword binary field that specifies the APITYPE:

- 2 APITYPE 2. This is the default. Allows an asynchronous macro API program to have only one outstanding socket call per socket descriptor. An APITYPE=2 program can use both asynchronous and synchronous calls.

UEEXIT

Any parameter is ignored.

ASYNC

Optional input parameter. One of the following:

- The literal 'NO' indicating no asynchronous support.
- The literal 'ECB' indicating the asynchronous support using ECBs is to be used.

ERROR

Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

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

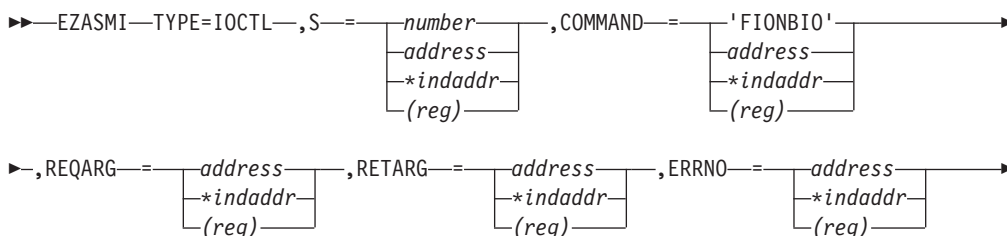
IOCTL

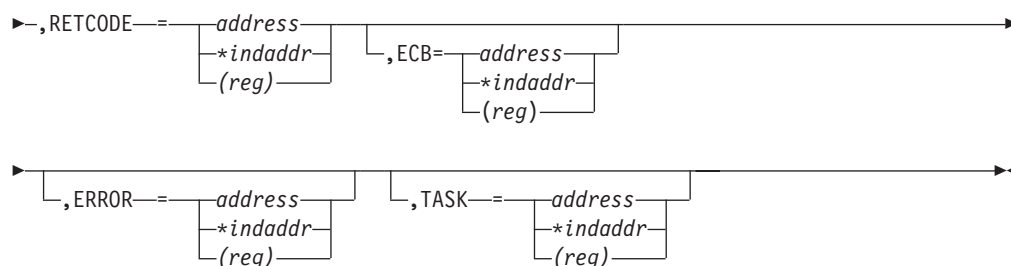
The IOCTL macro is used to control certain operating characteristics for a socket.

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

Format





Parameters

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 which communicates the requested operating characteristic to TCP/IP.

'FIONBIO'

Sets or clears blocking status.

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 see Table 9.

Table 9. 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

ERRNO

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

See "ERRNO Values" on page 74 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

Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.

- A 156-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 .

ERROR

Input parameter. The location in your program to receive control, if 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.

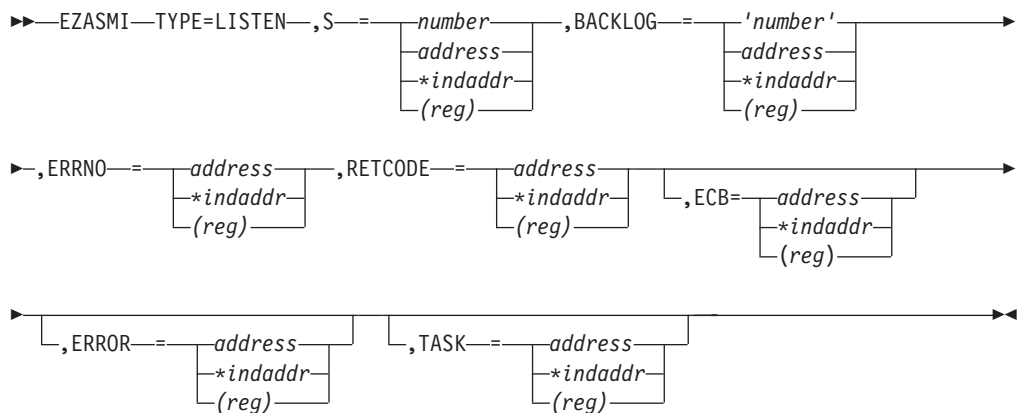
The LISTEN macro:

- Establishes the readiness to accept client connection requests.
- Creates a connection-request queue of a specified number of entries for incoming connection requests.
- The LISTEN macro requires a BIND request to be issued previously.

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 CICS Listener program is an example of a concurrent server.

Format



Parameters

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. This parameter is ignored. A value of 1 is always assumed.

ERRNO

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

See "ERRNO Values" on page 74, 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

Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.
- A 156-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.

ERROR

Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

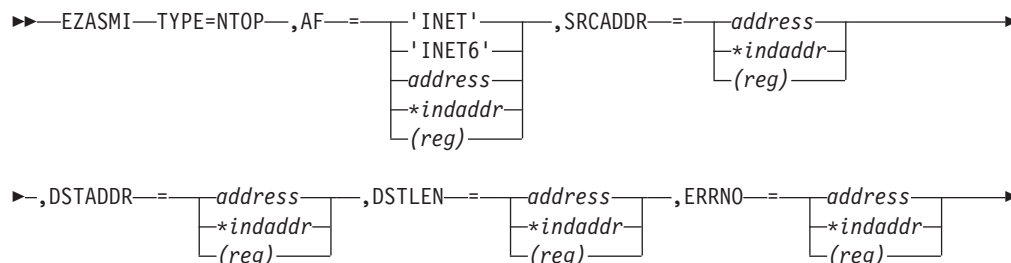
TASK Input parameter. The location of the task storage area in your program.

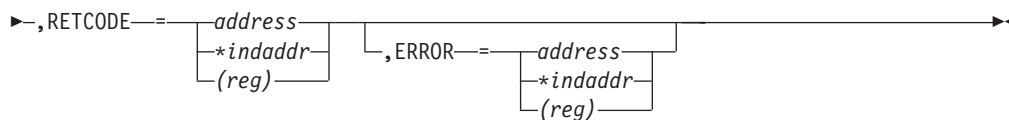
NTOP

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.

Important: This function call is not available with TCP/IP for VSE/ESA.

Format



Parameters

AF Input parameter. Specify one of the following:

'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 is in dotted-decimal format and for IPv6 the address is in colon-hex format. The size of the converted IPv4 address is a maximum of 15 bytes and the size of the converted IPv6 address is 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 contains 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 ERRNO.

See "ERRNO Values" on page 74, 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, if the application programming interface (API) processing module cannot be loaded.

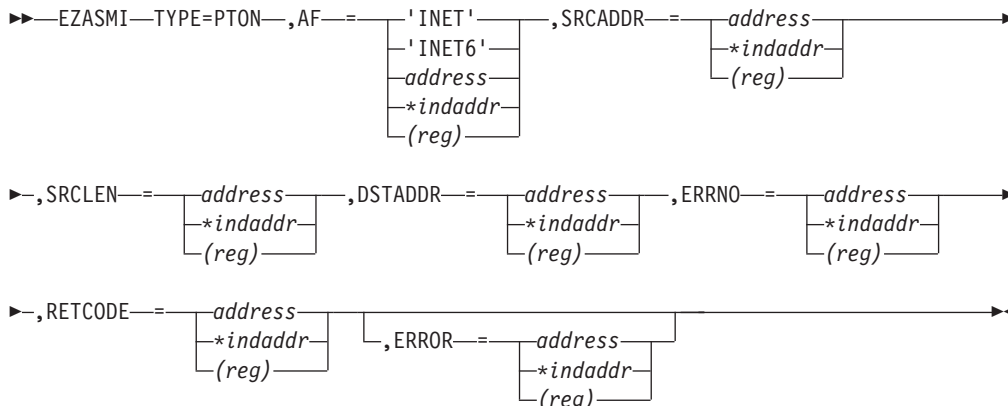
PTON

The PTON macro converts an IP address from its standard text presentation form to its numeric binary form.

On successful completion, PTON returns the converted IP address in the buffer provided.

Important: This function call is not available with TCP/IP for VSE/ESA.

Format



Parameters

AF Input parameter. Specify one of the following:

'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.

SRCLLEN

Input parameter. A binary halfword field that must contain the length of the IP address to be converted.

DSTADDR

A field that is 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 ERRNO.

See "ERRNO Values" on page 74, 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, if the application programming interface (API) processing module cannot be loaded.

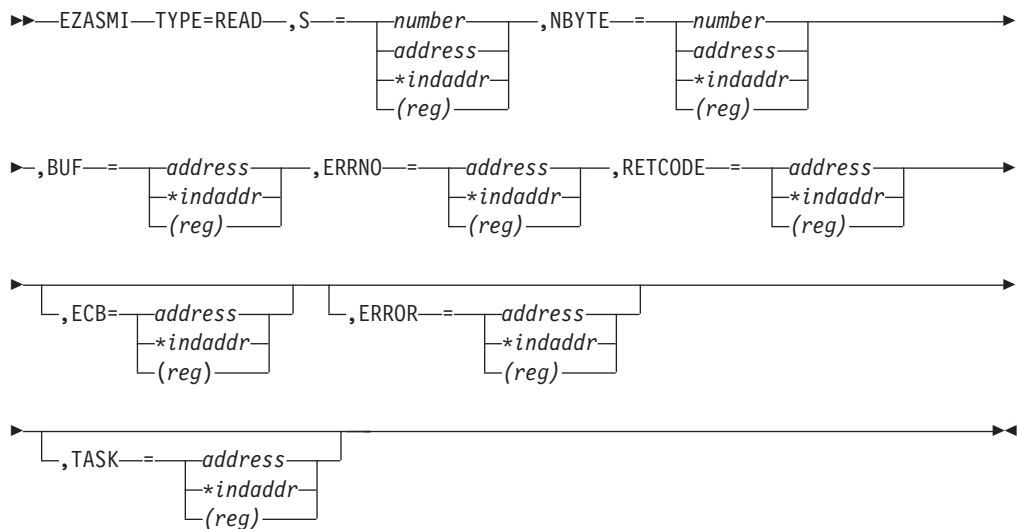
READ

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.

Format



Parameters

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.

ERRNO

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

See "ERRNO Values" on page 74 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

Value Description

- 0 A zero 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

Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.
- A 156-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.

ERROR

Input parameter. The location in your program to receive control, if 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.

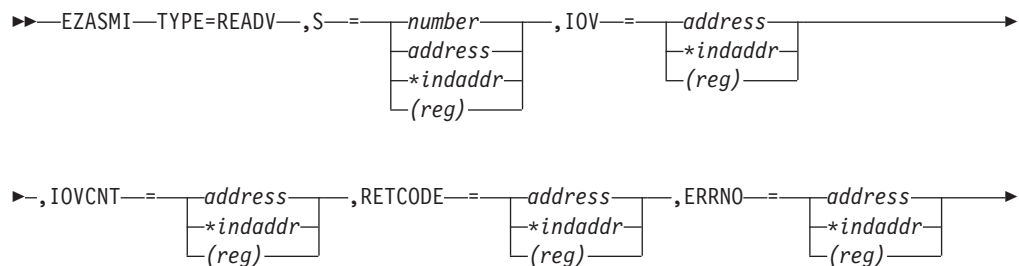
If data is not available for the socket and the socket is in blocking mode, the READ macro blocks the caller until data arrives. If data is not available, and the socket is in nonblocking mode, READ returns a -1 and sets ERRNO EWOULDBLOCK. See "IOCTL" on page 348 or "FCNTL" on page 305 for a description of how to set the nonblocking mode.

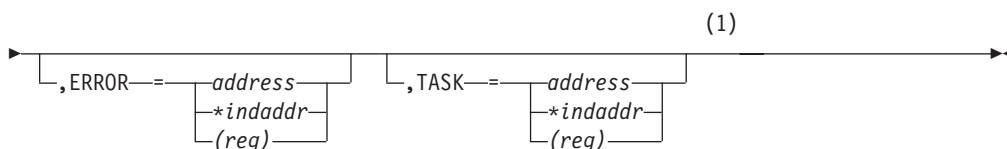
READV

The READV macro reads data on a socket and stores it into a set of buffers.

If a datagram socket is too long to fit into the supplied buffers, the extra bytes are discarded.

Format





Notes:

- 1 The ECB parameter for asynchronous processing is not supported with this call; unlike z/OS.

Parameters

- S** Input parameter. A value, or the address of a halfword binary number specifying the socket that is going to read the data.
- IOV** Input parameter. An array of three fullword structures with the number of structures equal to the value of **IOVCNT**.
- The format of the structures is as follows:
- Fullword 1: The address of the data buffer
 - Fullword 2: Reserved
 - Fullword 3: 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 maximum is 120.

RETCODE

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

Value Description

- 0** A zero return code indicates that the connection is closed and no data is available.
- >0** The number of bytes copied into the buffer set.
- 1** An error occurred. Check **ERRNO** for an error code.

ERRNO

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

See “**ERRNO Values**” on page 74 for information about **ERRNO** return codes.

ERROR

Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

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

READV returns up to the number of bytes specified with the sum of the Fullword 3 values in the IOV structures. If less than this sum is available, READV returns the number currently available.

RECV

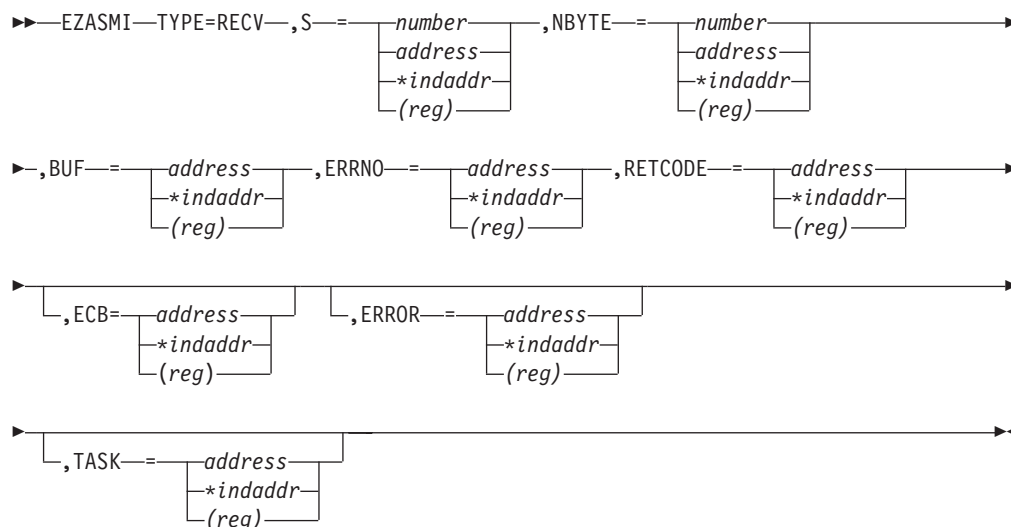
The RECV macro receives data on a socket and stores it in a buffer.

The RECV macro applies only to connected sockets.

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 one 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.

Format



Parameters

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.

ERRNO Output parameter. A fullword binary field. If RETCODE is negative, this field contains an error number. See "ERRNO Values" on page 74 for information about ERRNO return codes.

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

Value	Description
-------	-------------

RECV

- 0 A zero 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 Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.
- A 156-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.

ERROR

Input parameter. The location in your program to receive control, if 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 EWOULDBLOCK. See "FCNTL" on page 305 or "IOCTL" on page 348 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.

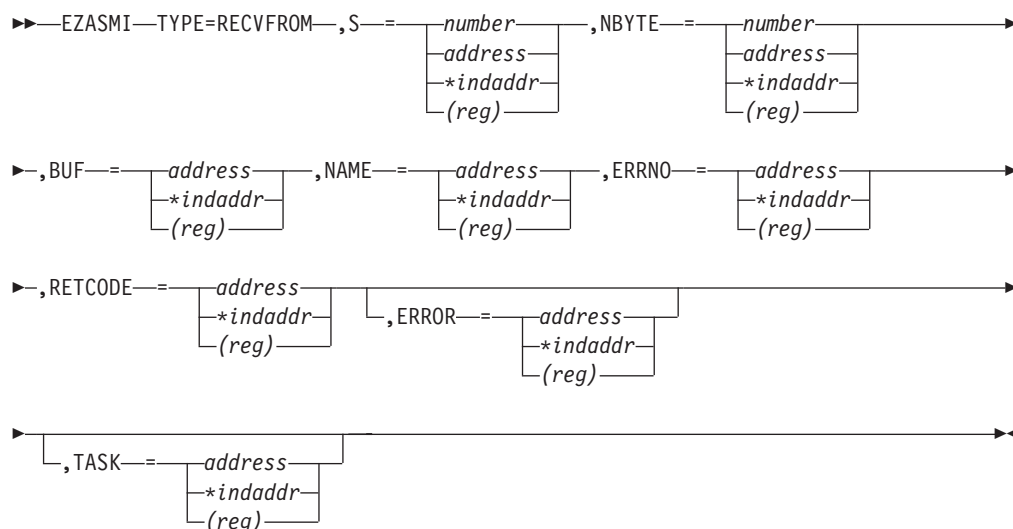
If data is not available for the socket designated by descriptor S, and socket S is in blocking mode, the RECVFROM call blocks the caller until data arrives.

If data is not available and socket S is in nonblocking mode, RECVFROM returns a -1 and sets ERRNO to 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 305 or "IOCTL" on page 348 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 one 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.

Important: IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply, if this program is used.

Format



Parameters

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. Include the `PRD1.MACLIB(EZBREHST)` 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:

FAMILY

A halfword binary field specifying the IPv4 addressing family. For TCP/IP the value is a decimal 2, indicating `AF_INET`.

PORT A halfword binary field specifying the port number of the sending socket.

IPv4-ADDRESS

A fullword binary field specifying the 32-bit IPv4 Internet address.

RESERVED

Specifies eight bytes of binary zeros. This field is required, but is not used.

The IPv6 socket address structure contains the following fields:

RECVFROM

NAMELEN

A 1-byte binary field that specifies the length of the IPv6 socket address structure. Any value specified by the use of this field is ignored, if processed as input. The field is set to 0, if processed as output.

FAMILY

A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary field specifying the port number of the sending socket.

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 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 "ERRNO Values" on page 74 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

Value	Description
-------	-------------

- | | |
|----|--|
| 0 | A zero 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. |

ERROR

Input parameter. The location in your program to receive control, if 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, if 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.

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. The bit strings are contained in 32-bit fullwords, numbered from right to left. The rightmost bit represents socket descriptor zero; the leftmost bit represents socket descriptor 31, and so on. If your process uses 32 or fewer sockets, the bit string is one fullword. If your process uses 33 sockets, the bit string is two full words. The first fullword represents socket descriptors 0 to 31, the second fullword is for socket descriptors 32 to 63. You define the sockets that you want to test by turning on bits in the string.

Read Operations

The ACCEPT, READ, RECV, and RECVFROM macros are read operations. A socket is ready for reading when data is received on it, or when an exception condition occurs.

To determine if a socket is ready for the read operation, set the appropriate bit in RSNDSK to '1' before issuing the SELECT macro. When the SELECT macro returns, the corresponding bits in the RRETMSK indicate sockets ready for reading.

Write Operations

A socket is selected for writing, ready to be written, if:

- 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 that has previously returned ERRNO EINPROGRESS, completes the connection.

The WRITE, SEND, or SENDTO macros block, if 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.

To determine if a socket is ready for the write operation, set the appropriate bit in WSNDSK to '1'.

Exception Operations

For each socket to be tested, the SELECT macro can check for an exception condition. The exception conditions are:

- The calling program (concurrent server) has issued a GIVESOCKET command and the target subtask has successfully issued the TAKESOCKET call. If 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 xSNDMSK, a bit string records the results of the check. The bit strings are RRETMSK, WRETMSK, and ERETMSK for read, write, and exceptional events. On return from the SELECT macro, each bit set to '1' in the xRETMSK is a read, write, or exceptional event for the associated socket.

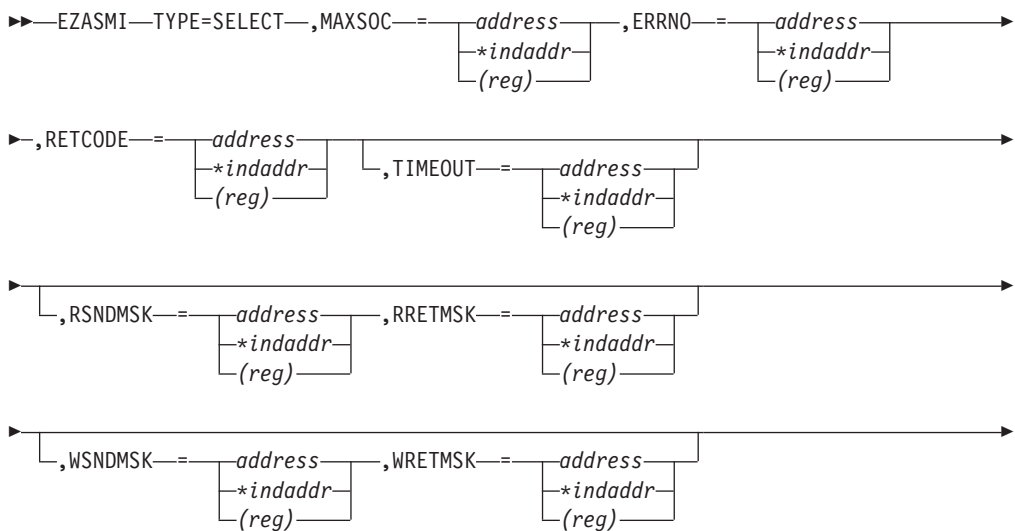
MAXSOC Parameter

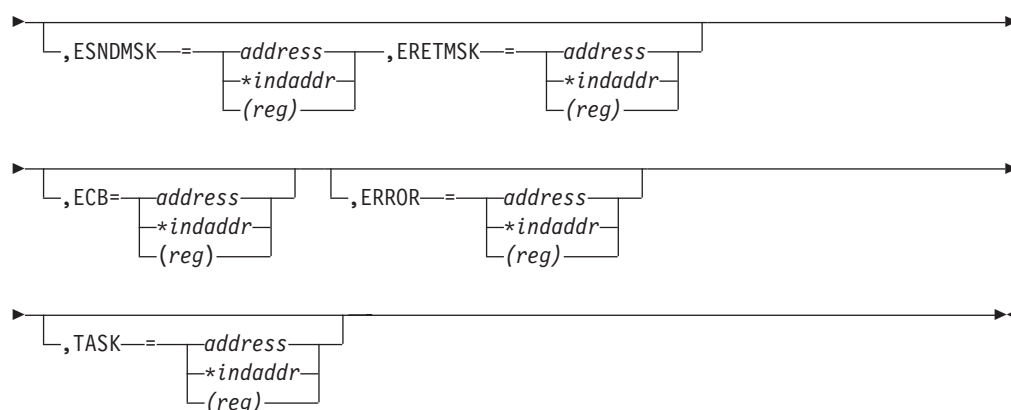
The SELECT call must test each bit in each string before returning any results. For efficiency, the MAXSOC parameter can be set to the largest socket number for any event type. The SELECT call tests only bits in the range 0 through the MAXSOC value.

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.

Format





Parameters

MAXSOC

Input parameter. A fullword binary field specifying the largest socket descriptor number to be checked plus 1 (remember, TCP/IP for VSE/ESA supports socket descriptor numbers from 0 to 8191).

ERRNO

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

See “ERRNO Values” on page 74, for information about ERRNO return codes.

RETCODE

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

Value Description

- >0 Indicates the number of ready sockets in the three return masks.
- =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 zero-valued TIMEVAL structure.

TIMEOUT is specified in the two-word TIMEOUT as follows:

- TIMEOUT-SECONDS, word one of TIMEOUT, is the seconds component of the timeout value.
- TIMEOUT-MICROSEC, word two of TIMEOUT, is the microseconds component of the timeout value (0–999999).

For example, if you want SELECT to timeout after 3.5 seconds, set TIMEOUT-SECONDS to 3 and TIMEOUT-MICROSEC to 500000.

RSNDMSK

Input parameter. A bit string sent to request read event status.

SELECT

- 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.

RRETMSK

Output parameter. A bit string that returns the status of read events.

- For each socket that is ready for 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 Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.
- A 156-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.

ERROR

Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

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

SELECTEX

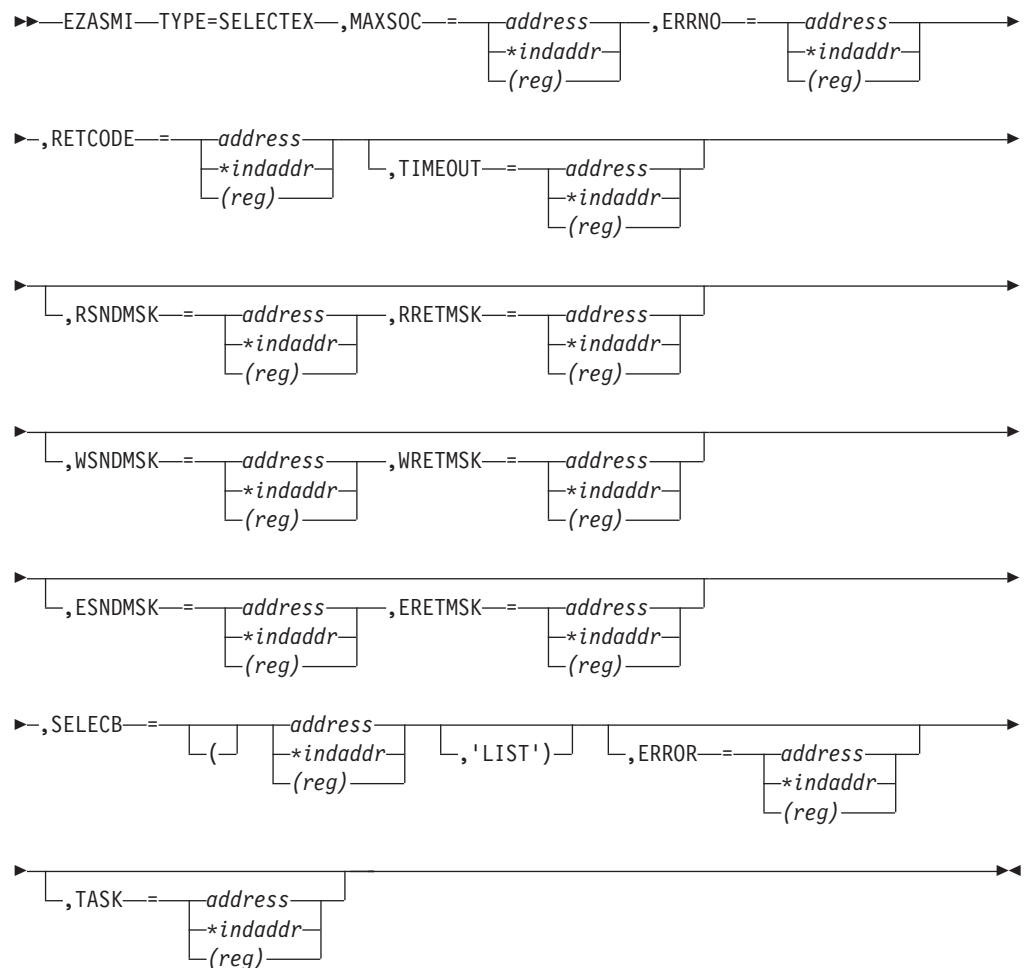
The SELECTEX macro monitors a set of sockets, a time value, and an ECB or list of ECBs.

It completes, if 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.

For a detailed description on testing sockets, refer to “SELECT” on page 360.

Format

Parameters**MAXSOC**

Input parameter. A fullword binary field specifying the largest socket descriptor number to be checked plus 1 (remember, TCP/IP for VSE/ESA supports socket descriptor numbers from 0 to 8191).

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 | 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 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:

- TIMEOUT-SECONDS, word one of TIMEOUT, is the seconds component of the time out value.
- TIMEOUT-MICROSEC, word two of TIMEOUT, is the microseconds component of the time out 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.

RRETMASK

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.

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.

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.

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.

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.

SELECB

Input parameter. An ECB or list of ECB addresses which, if posted, causes completion of the SELECTEX.

If the address of an ECB address list is specified you must set the high-order bit of the last entry in the ECB list to one and you must also add the LIST keyword. The ECBs must reside in the caller's home address space.

Note: The maximum number of ECBs that can be specified in a list is 254.

ERROR

Input parameter. The location in your program to receive control, if 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.

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.

Format

```

▶▶—EZASMI—TYPE=SEND—,S—=—

|          |
|----------|
| number   |
| address  |
| *indaddr |
| (reg)    |

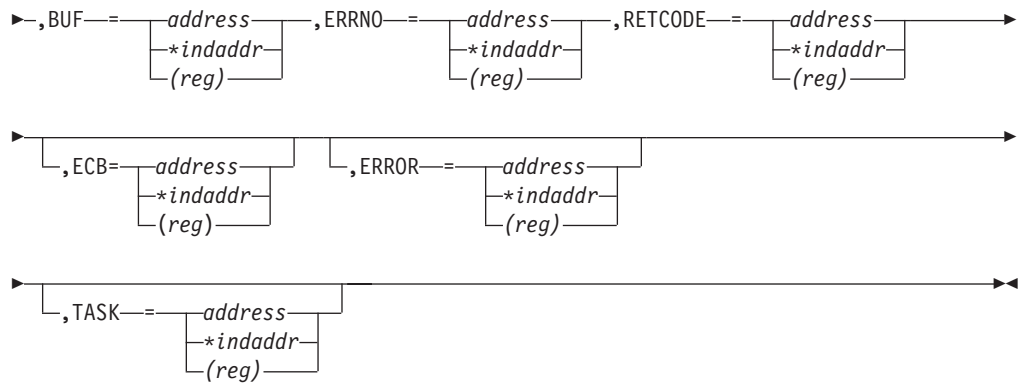
,NBYTE—=—

|          |
|----------|
| number   |
| address  |
| *indaddr |
| (reg)    |

—▶▶

```

SEND



Parameters

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.

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, this field contains an error number. See "ERRNO Values" on page 74, 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

ECB Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.
- A 156-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 .

ERROR

Input parameter. The location in your program to receive control, if 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.

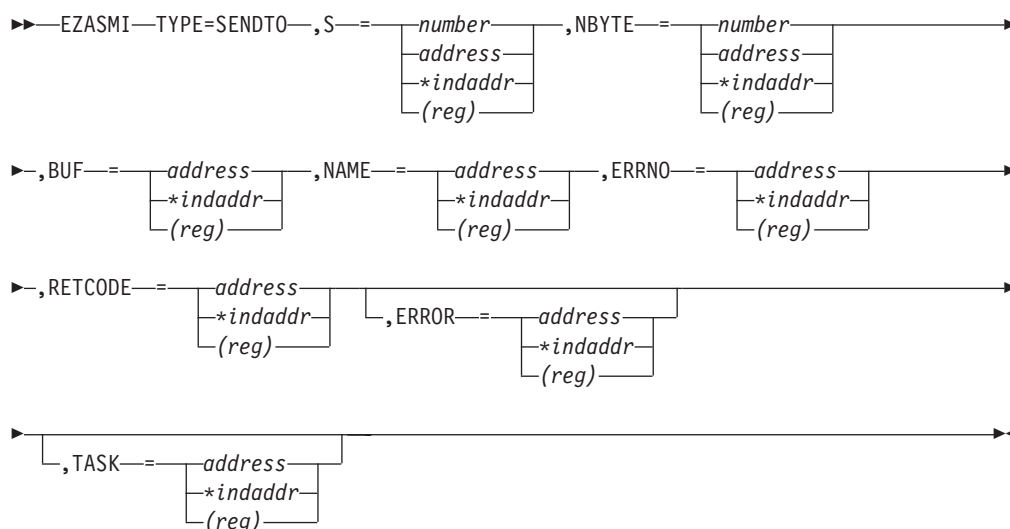
You can use the destination address on the SENDTO macro to send datagrams on a UDP socket that is connected or not connected.

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.

Important: IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply, if this program is used.

Format



Parameters

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.

NAME

Input parameter. The address of the IPv4 or IPv6 target. Include the PRD1.MACLIB(EZBREHST) 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:

FAMILY

A halfword binary field specifying the IPv4 addressing family. For TCP/IP the value is a decimal 2, indicating AF_INET.

PORT A halfword binary field specifying the port number bound to the socket.

IPv4-ADDRESS

A fullword binary field specifying the 32-bit IPv4 Internet address of the socket.

RESERVED

Specifies eight bytes of binary zeros. This field is required, but is not used.

The IPv6 socket address structure contains the following fields:

NAMELEN

A 1-byte binary field that specifies the length of the IPv6 socket address structure. Any value specified by the use of this field is ignored, if processed as input. The field is set to 0, if processed as output.

FAMILY

A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary field specifying the port number bound to the socket.

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 of the socket, in network byte order, of the client 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 "ERRNO Values" on page 74 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.

ERROR

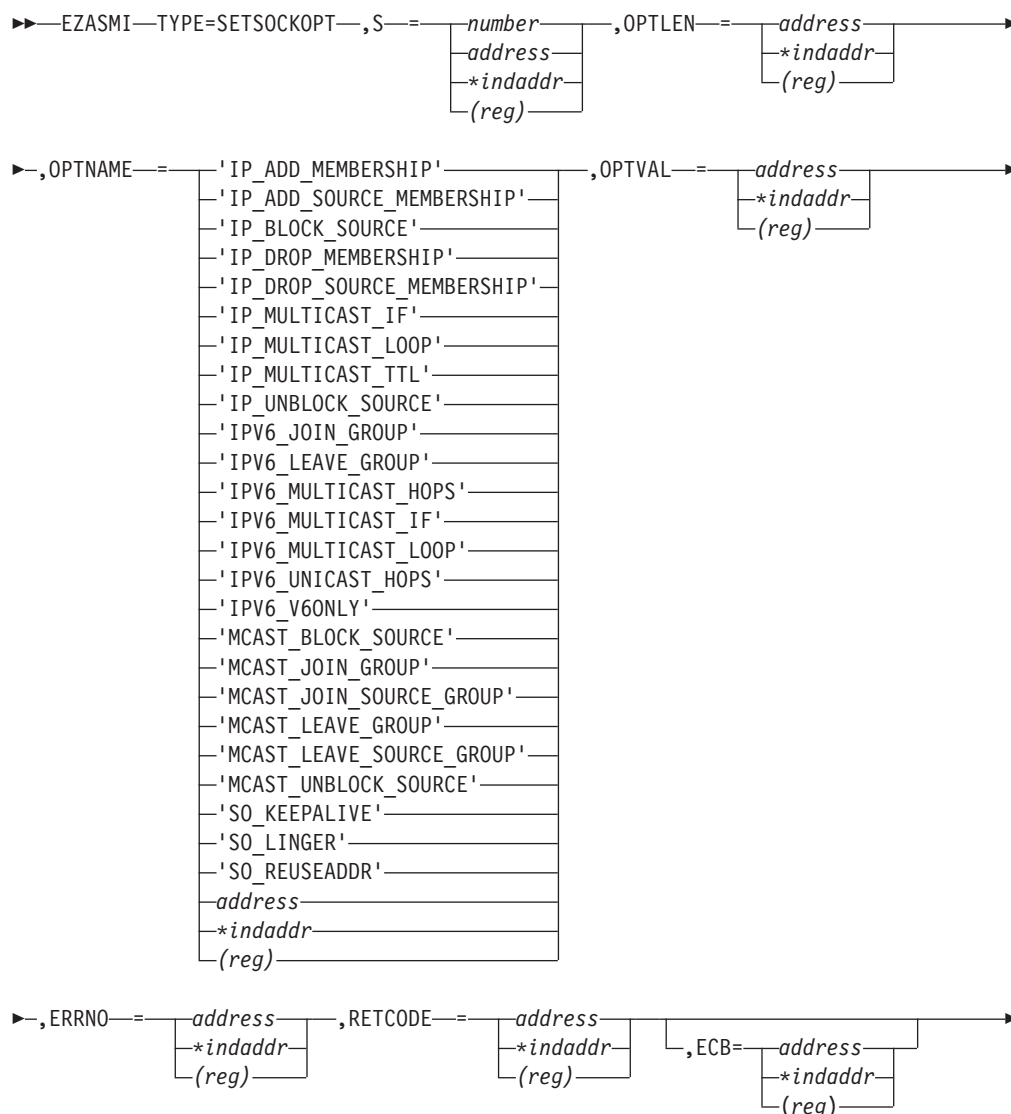
Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

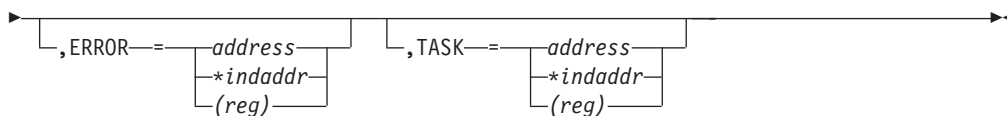
TASK Input parameter. The location of the task storage area in your program.

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**.

Format



Parameters

S A value, or the address of a halfword binary number specifying the socket sending the data.

OPTLEN

Input parameter. A fullword binary number specifying the length of the field specified by **OPTVAL**.

OPTNAME

Input parameter. Indicates the following values:

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.

IP_ADD_SOURCE_MEMBERSHIP

Use this option to enable an application to join a source multicast group on a specific interface and a specific source address. You must specify an interface and a source address with this option. Applications that want to receive multicast datagrams need to join source multicast groups. This is an IPv4-only socket option.

IP_BLOCK_SOURCE

Use this option to enable an application to block multicast packets that have a source address that matches the given IPv4 source address. You must specify an interface and a source address with this option. The specified multicast group must have been joined previously. This is an IPv4-only socket option.

IP_DROP_MEMBERSHIP

Use this option to enable an application to exit a multicast group or to exit all sources for a multicast group. This is an IPv4-only socket option.

IP_DROP_SOURCE_MEMBERSHIP

Use this option to enable an application to exit a source multicast group. This is an IPv4-only socket option.

IP_MULTICAST_IF

Use this option to set 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.

IP_MULTICAST_LOOP

Use this option to control 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.

IP_MULTICAST_TTL

Use this option to set 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.

IP_UNBLOCK_SOURCE

Use this option to enable an application to unblock a previously blocked source for a given IPv4 multicast group. You must specify an interface and a source address with this option. This is an IPv4-only socket option.

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.

IPV6_LEAVE_GROUP

Use this option to control the reception of multicast packets and specify that the socket leave a multicast group. This is an IPv6-only socket option.

IPV6_MULTICAST_HOPS

Use to set the hop limit used for outgoing multicast packets. This is an IPv6-only socket option.

IPV6_MULTICAST_IF

Use this option to set the index of the IPv6 interface used for sending outbound multicast datagrams from the socket application. This is an IPv6-only socket option.

IPV6_MULTICAST_LOOP

Use this option to control whether a multicast datagram is looped back on the outgoing interface by the IP layer for local delivery, if 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.

IPV6_UNICAST_HOPS

Use this option to set the hop limit used for outgoing unicast IPv6 packets. This is an IPv6-only socket option.

IPV6_V6ONLY

Use this option to set 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.

MCAST_BLOCK_SOURCE

Use this option to enable an application to block multicast packets that have a source address that matches the given source address. You must specify an interface index and a source address with this option. The specified multicast group must have been joined previously.

MCAST_JOIN_GROUP

Use this option to enable an application to join a multicast group on a specific interface. You must specify an interface index. Applications that want to receive multicast datagrams must join multicast groups.

MCAST_JOIN_SOURCE_GROUP

Use this option to enable an application to join a source multicast group on a specific interface and a source address. You must specify an interface index and the source address. Applications that want to receive multicast datagrams only from specific source addresses need to join source multicast groups.

MCAST_LEAVE_GROUP

Use this option to enable an application to exit a multicast group or exit all sources for a given multicast groups.

MCAST_LEAVE_SOURCE_GROUP

Use this option to enable an application to exit a source multicast group.

MCAST_UNBLOCK_SOURCE

Use this option to enable an application to unblock a previously blocked source for a given multicast group. You must specify an interface index and a source address with this option.

SO_KEEPAIVE

This option is provided for source compatibility reasons only. It will not perform any action. Instead the user should use the common TCP/IP setting: SET PULSE_TIME=nnn.

SO_LINGER

Controls how TCP/IP processes data that has not been transmitted, if a CLOSE macro is issued for the socket. This option has meaning only for stream sockets.

- If **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.
- If **SO_LINGER** is not set, the CLOSE macro 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. 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**.

The default is DISABLED.

SO_REUSEADDR

This option is provided for source compatibility reasons only. It will not perform any action. TCP/IP implicitly allows for immediate address reuse.

OPTVAL

Input parameter. Contains data about the option specified in **OPTNAME**.

- **OPTVAL** is a 32-bit binary number for all values of **OPTNAME**, except **SO_LINGER**. Set **OPTVAL** to a nonzero positive value to enable the option. set **OPTVAL** to zero to disable the option.
- For **SO_LINGER**, **OPTVAL** is:

ONOFF	DS	F	ON OR OFF
LINGER	DS	F	TIME IN SECONDS

Set **ONOFF** to a nonzero value to enable the option and set it to zero to disable the option. Set the **LINGER** value to the time in seconds that TCP/IP lingers after the CLOSE macro is issued.

ERRNO

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

See "ERRNO Values" on page 74, for information about ERRNO return codes.

RETCODE

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

Value Description

- >0 Indicates the number of ready sockets in the three return masks.
- =0 Indicates that the SELECT time limit has expired.
- 1 Check ERRNO for an error code

ECB

Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.
- A 156-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.

ERROR

Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

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

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. It is optional and can be set to the NULL pointer, if data is not needed by the command. The OPTLEN parameter must be set to the size of the data pointed to by **OPTVAL**.

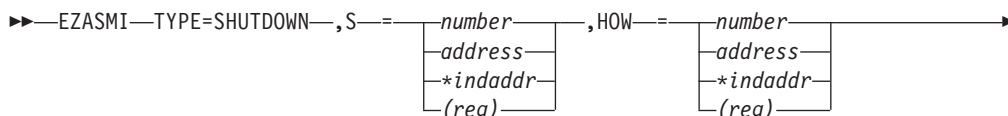
SHUTDOWN

The SHUTDOWN macro can be used to close one-way traffic while completing data transfer in the other direction.

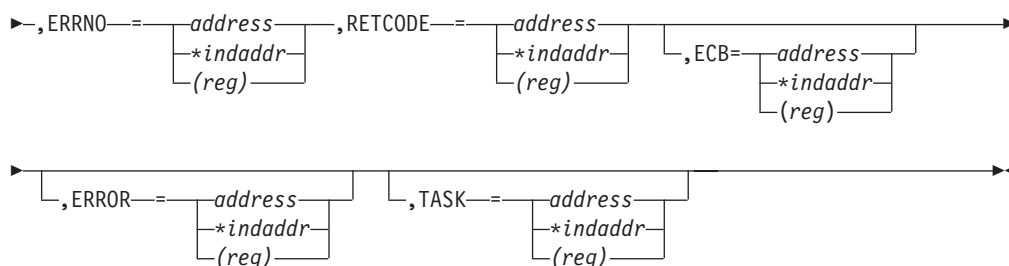
The HOW parameter determines the direction of the traffic to shutdown. A client program can use the SHUTDOWN macro to reuse a given socket with a different connection.

Another way to terminate a network connection is to issue a "CLOSE" on page 301 macro that attempts to complete all outstanding data transmission requests prior to breaking the connection.

Format



SHUTDOWN



Parameters

S Input parameter. A value, or the address of a halfword binary number specifying the socket to be shutdown.

HOW Input parameter. A fullword binary field specifying the shutdown method.

Value	Description
-------	-------------

2	Ends further send and receive operations.
---	---

ERRNO

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

See “ERRNO Values” on page 74, for information about ERRNO return codes.

RETCODE

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

Value	Description
-------	-------------

0	Successful call
---	-----------------

-1	Check ERRNO for an error code
----	-------------------------------

ECB Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.
- A 156-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 .

ERROR

Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

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

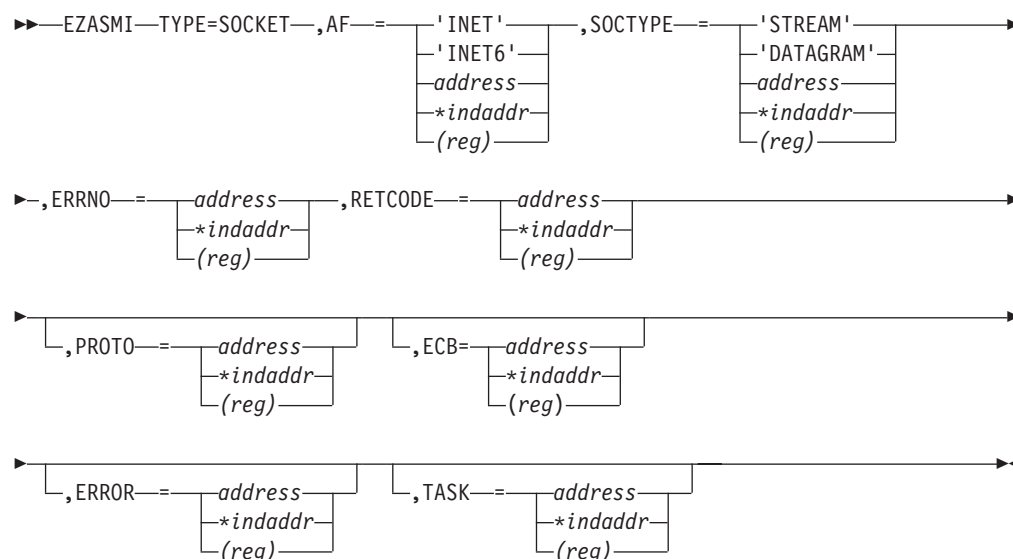
SOCKET

The **SOCKET** macro creates an endpoint for communication and returns a socket descriptor representing the endpoint.

Different types of sockets provide different communication services.

Important: IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply, if this program is used.

Format



Parameters

AF Input parameter. Specifies the literal INET or INET6, which indicates the internet or TCP/IP. Specify one of the following:

'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.

SOCTYPE

Input parameter. A fullword binary field set to the type of socket required. The types are:

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.

Note: RAW sockets are not supported.

ERRNO

Output parameter. A fullword binary field. If RETCODE is negative, this field contains an error number. See "ERRNO Values" on page 74 for information about ERRNO return codes.

SOCKET

RETCODE

Output parameter. 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
--	-------------------------------

PROTO

Input parameter. A fullword binary number specifying the protocol supported.

ECB Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.
- A 156-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 .

ERROR

Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

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

PROTO specifies a particular protocol to be used with the socket. 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. If PROTO is set to 17, the UDP Protocol is used. If it is set to 6, the TCP protocol is used.

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 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.

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 207.

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).

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.

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 which it obtained from the concurrent server.

Note:

1. If 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 CICS Listener program is an example of a concurrent server.

Important: IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply, if this program is used.

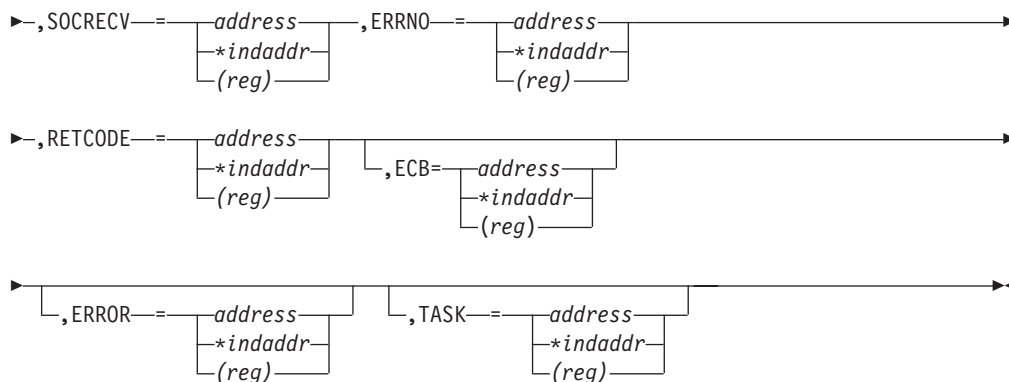
Format

```

▶▶—EZASMI—TYPE=TAKESOCKET—,CLIENT—=—address————→
*indaddr
(reg)

```

TAKESOCKET



Parameters

CLIENT

Input parameter. The client data returned by the GETCLIENTID macro.

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 eight-byte character field set to the VSE partition identifier of the program giving the socket.

TASK Input parameter. Specifies an eight-byte character field. This field must match the value of the SUBTASK parameter on the INITAPI for the VSE task that issued the GIVESOCKET request.

RESERVED

Input parameter. A 20-byte reserved field. This field is required and only used internally.

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, ERRNO contains a valid error number. Otherwise, ignore ERRNO.

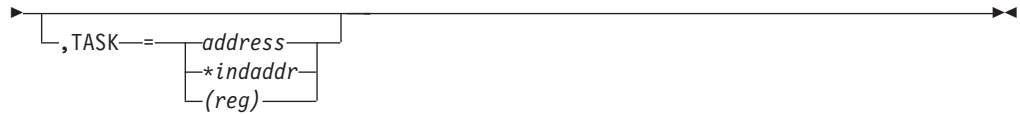
See "ERRNO Values" on page 74, for information about ERRNO return codes.

RETCODE

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

Value	Description
-------	-------------

- | | |
|----|--|
| >0 | Indicates the number of ready sockets in the three return masks. |
| =0 | Indicates that the SELECT time limit has expired. |
| -1 | Check ERRNO for an error code |



Parameters

ERROR

Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

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

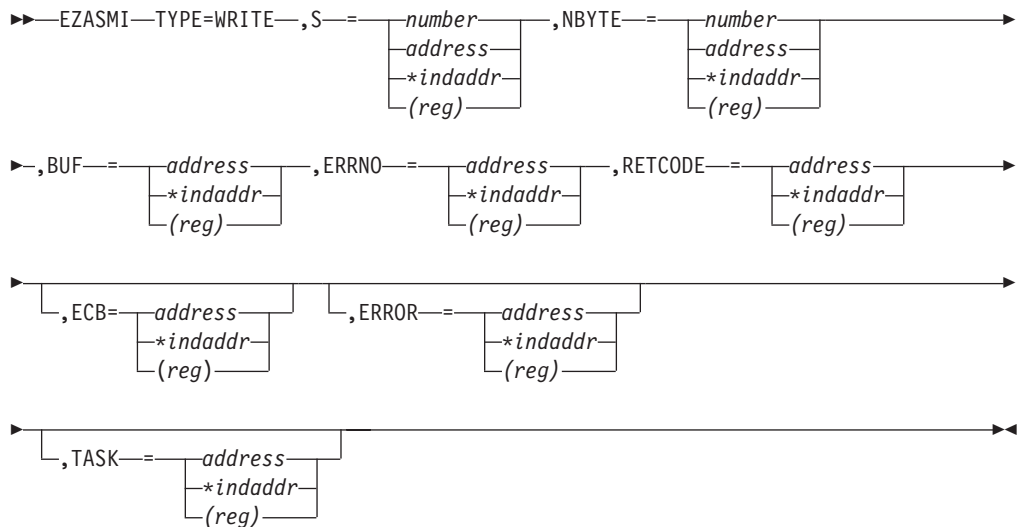
WRITE

The WRITE macro writes data on a connected socket. The WRITE macro is similar to the SEND macro.

For datagram sockets, this macro writes the entire datagram, if it fits 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 one byte, ten 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.

Format



Parameters

S Input parameter. A value, or the address of a halfword binary number specifying the socket descriptor of the socket to send 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.

ERRNO

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

See "ERRNO Values" on page 74, 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

ECB Input parameter. It points to a 160-byte field containing:

- A four-byte ECB posted by TCP/IP when the macro completes.
- A 156-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.

ERROR

Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

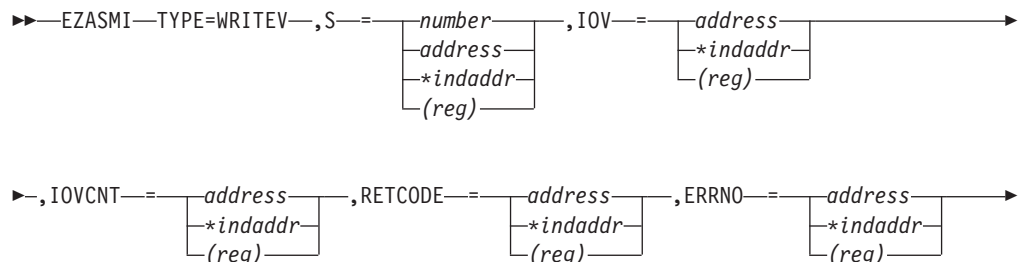
TASK Input parameter. The location of the task storage area in your program.

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 EWOULDBLOCK. See "FCNTL" on page 305 or "IOCTL" on page 348 for a description of how to set the nonblocking mode.

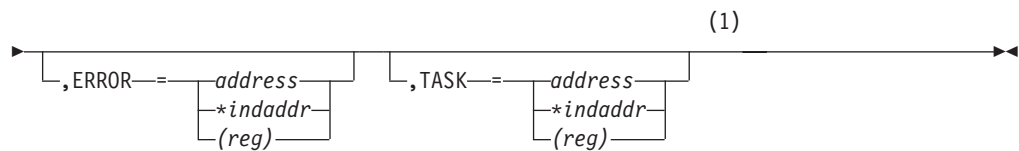
WRITEV

The WRITEV macro writes data on a socket from a set of buffers.

Format



WRITEV



Notes:

- 1 The ECB parameter for asynchronous processing is not supported with this call; unlike z/OS.

Parameters

S Input parameter. A value, or the address of a halfword binary number specifying the socket that will send the data.

IOV Input parameter. An array of three fullword structures with the number of structures equal to the value of **IOVCNT**.

The format of the structures is as follows:

- Fullword 1: The address of the data buffer
- Fullword 2: Reserved
- Fullword 3: 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 maximum is 120.

RETCODE

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

Value	Description
-------	-------------

<code>>=0</code>	The number of bytes sent.
---------------------	---------------------------

<code>-1</code>	An error occurred. Check <code>ERRNO</code> for an error code
-----------------	---

ERRNO

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

See "ERRNO Values" on page 74, for information about **ERRNO** return codes.

ERROR

Input parameter. The location in your program to receive control, if the application programming interface (API) processing module cannot be loaded.

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

Part 4. Using Fast Path to Linux

Chapter 13. Running z/VSE With a Linux Fast Path

This section describes the *Fast Path to Linux on System z* function, referred to simply as *Linux Fast Path* (or LFP).

Linux Fast Path allows selected TCP/IP applications to communicate with the TCP/IP stack on Linux on System z without using a TCP/IP stack on z/VSE.

The Linux Fast Path can be run in either a z/VM environment or an LPAR environment.

- If you run LFP in a z/VM environment, both z/VSE and Linux on System z run in the same z/VM-mode LPAR on an IBM System z10™ or later server. An IUCV connection is used between z/VSE and Linux on System z.
- If you run LFP in an LPAR environment, both z/VSE and Linux on System z run in their own LPARs on a zEnterprise® server. A HiperSockets connection is used between z/VSE and Linux on System z. LFP requires the HiperSockets Completion Queue function that is available with a zEnterprise server.

For a general introduction on Fast Path to Linux on System z refer to *z/VSE Planning*.

If using socket applications written in LE/C with LFP, you must configure the LE/C TCP/IP Socket API Multiplexer. Refer to “Preparing to use Socket APIs with Linux Fast Path on z/VSE” on page 395 and the *LE/VSE C Run-Time Library Reference* for details.

This section contains these main topics:

- “Overview of Linux Fast Path”
- “Prerequisites for Using Linux Fast Path” on page 388
- “Restrictions When Using Linux Fast Path” on page 389
- “Communication Flow When Not Using Linux Fast Path” on page 389
- “Communication Flow When Using Linux Fast Path in a z/VM Environment” on page 390
- “Communication Flow When Using Linux Fast Path in an LPAR Environment” on page 391
- “Preparing Linux on System z to Use Linux Fast Path” on page 393
- “Configuring Linux Fast Path” on page 397
- “Starting and Stopping Linux Fast Path ” on page 407
- “Administrative Tasks” on page 411

Overview of Linux Fast Path

LFP transparently forwards all socket requests to a Linux on System z system, where the LFP daemon (lfpd) must run. This daemon fulfills all socket requests by forwarding them to the Linux TCP/IP stack.

You can start *multiple* LFP instances:

- Each instance is identified by its ID (00 to 99) and represents a connection to an LFP daemon on a Linux system.

- The ID corresponds to the ID value you use for the TCP/IP stack (for example, `// EXEC IPNET, PARM= ' ID=nn '`).
- The IDs must be unique across all TCP/IP stacks and LFP instances. Therefore, you cannot have a TCP/IP stack with the same ID as an LFP instance.

The following APIs are available for use with the Linux Fast Path:

- LE/C socket API via an alternative `$EDCTCPV.PHASE (IJB LFPLE)`.
- EZA SOCKET and EZASMI interface via an alternative EZA interface phase `IJB LFPPEZ`.
- CSI's (*Connectivity Systems, Incorporated*) assembler socket interface via the SOCKET macro.

LFP is *not* intended to replace an existing TCP/IP stack on z/VSE that is used to communicate with another remote server. LFP only provides TCP/IP socket APIs for programs running on z/VSE:

- These APIs are compatible to existing APIs, and enable existing socket programs to run unchanged using LFP.
- Except for the basic socket API, no other tools are provided.
- You still require a TCP/IP stack on z/VSE in order to run FTP servers/daemons, TELNET servers, LPR/LPD, and so on.

Prerequisites for Using Linux Fast Path

These are the prerequisites for using LFP in a z/VM environment:

- If you use a z/VM-mode LPAR, z/VM Version 5 Release 4 or later. Otherwise, any z/VM release supported by z/VSE.
- If you use a z/VM-mode LPAR, IBM System z10 or later. Otherwise, any server supported by z/VSE.
- z/VSE Version 4 Release 3 or later.
- One of these Linux on System z operating systems:
 - SUSE SLES 10 SP3 together with security update kernel 2.6.16.60-0.57.1.
 - SUSE SLES 11 SP1
 - Red Hat RHEL 5 Update 5
 - Red Hat RHEL 6
- z/VSE and Linux on System z have to be configured as z/VM guests within the *same* z/VM, because an IUCV connection is used.
- The IUCV (“Inter-User Communication Vehicle”) has to be configured and enabled in both z/VM guests (z/VSE and Linux on System z).

These are the prerequisites for using LFP in an LPAR environment:

- A zEnterprise server at driver level 93 or later. LFP requires the HiperSockets Completion Queue function, which is only available with a zEnterprise server.
- z/VSE Version 5 Release 1 with the following APARs/PTFs installed: DY47300/UD53758, PM56023/UK76218 and PM56056/UK76252.
- One of these Linux on System z operating systems:
 - SUSE SLES 11 SP2
 - Red Hat: IBM is working with its Linux distribution partners to include support in future Linux on System z distribution releases.
- One z/VSE system and one Linux on System z system running in LPAR mode.

Restrictions When Using Linux Fast Path

These are the restrictions when using LFP:

- For the CSI (*Connectivity Systems, Incorporated*) interface, which is the SOCKET macro, LFP only supports connection types:
 - TCP
 - UDP
 - CONTROL

Other connection types (such as CLIENT, TELNET, FTP, RAW, and so on) are *not* supported and are rejected, if used with LFP.

- For CONTROL type connections, the only commands supported are:
 - GETHOSTBYNAME
 - GETHOSTBYADDR
 - GETHOSTNAME
 - GETHOSTID

For details, refer to the individual macro descriptions in the "TCP/IP for VSE V1R5F Programmers Guide".

- For CONTROL type connections, these commands (from Barnard Software, Incorporated) are also supported:
 - NTOP
 - PTON
 - GETVENDORINFO

For details, refer to the *IPv6/VSE Programming Guide* that is published by Barnard Software, Incorporated.

- If you use the SSL API (using gsk_nnn functions), you need to have a TCP/IP for VSE/ESA TCP/IP stack up and running using a different ID. The SSL functions use services from the TCP/IP stack, although the communication flow uses LFP.

Communication Flow When Not Using Linux Fast Path

This topic provides a detailed description of a typical configuration that does not use a Linux Fast Path. This is shown in Figure 22 on page 390. In this configuration, a DB2[®] client running under z/VSE communicates with the DB2 Server for Linux running under Linux.

This configuration also applies, if you run other z/VSE-supplied programs that run on Linux on System z (the VSE Redirector Server, Redirector Handler, and so on). In Figure 22 on page 390:

- The communication between z/VSE and is established using HiperSockets.
- All communication must be routed through the TCP/IP stacks on z/VSE and Linux on System z.

A fast HiperSockets connection can easily lead to a high system utilization resulting from the required overhead of the TCP-processing and IP-processing.

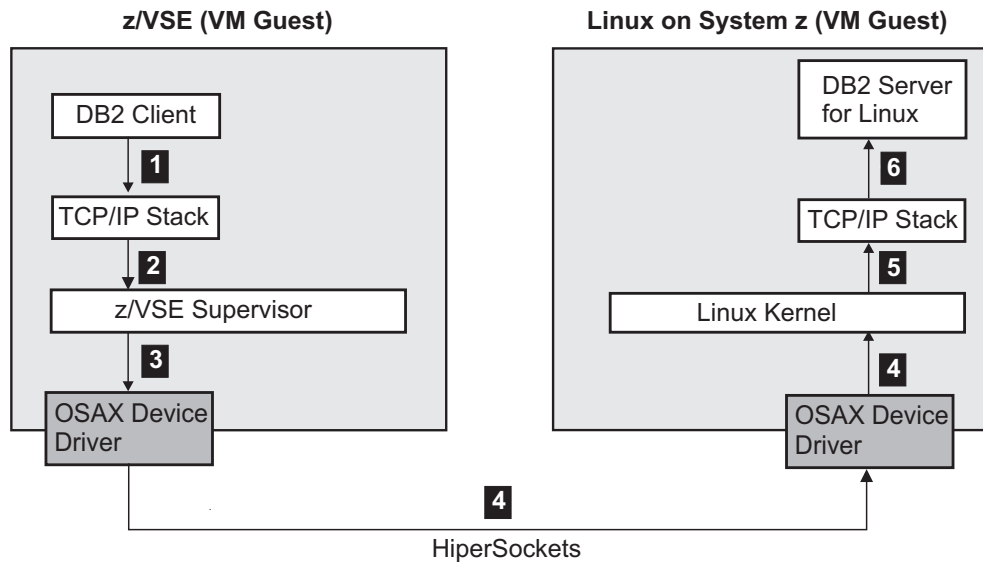


Figure 22. Communication Between VM Guests via HiperSockets

The processing shown in Figure 22 is as follows:

- 1** The data is passed from the z/VSE application (in this example, the DB2 Client) to the TCP/IP Stack running in another partition. This is done using cross-partition communication mechanisms and involves some dispatching activities in the z/VSE supervisor.
- 2** TCP/IP builds one or multiple TCP packets with the data from the user applications. It builds a TCP header as well as an IP header. This processing includes handling for retransmissions, sequence numbers and acknowledging, calculating checksums, and so on.
- 3** The TCP/IP stack passes the packets to the network device driver for use with HiperSockets (for example, the OSAX device driver).
- 4** The HiperSockets network forwards the packets to the Linux image.
- 5** The Linux HiperSockets device driver receives the packets and passes them to the TCP/IP stack. The TCP/IP stack on Linux checks and unpacks the IP and TCP header. This processing includes handling for retransmissions, sequence numbers and acknowledging, validating checksums and so on.
- 6** The TCP/IP stack passes the data to the application (in this example, the DB2 Server) running on Linux. The DB2 Server receives and processes the data.

For data to be sent from the DB2 Server on Linux back to the DB2 Client on z/VSE, the same six steps as above are performed in reverse order.

Communication Flow When Using Linux Fast Path in a z/VM Environment

This topic provides a detailed description of a typical configuration that uses a Linux Fast Path in a z/VM environment . This is shown in Figure 23 on page 391. This configuration also uses the example of a DB2 client running under z/VSE communicating with the DB2 Server for Linux running under Linux.

In Figure 23 on page 391:

- The communication between z/VSE and Linux on System z is established using an *IUCV* communication path.
- The Linux Fast Path establishes an IUCV communication path between z/VSE and Linux on System z *without* using:
 - A TCP/IP stack on z/VSE
 - OSA Express adapters

The flow of actions described in this example also apply when running other z/VSE-supplied programs that run on Linux (the VSE Redirector Server, VSE Script Server, Redirector Handler, and so on).

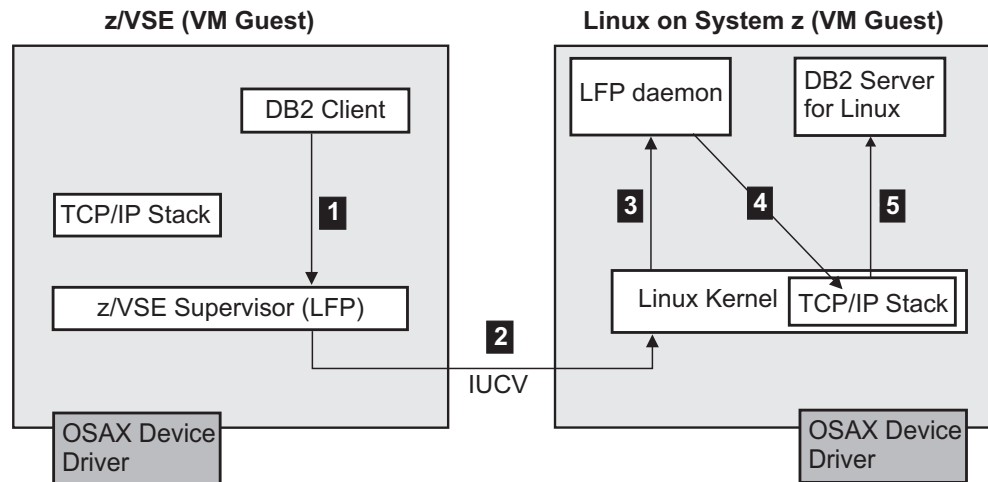


Figure 23. Communication Between VM Guests via a Linux Fast Path

The processing shown in Figure 23 is as follows:

- 1** The data to be sent is passed to LFP running on z/VSE.
- 2** LFP sends the data via the IUCV channel to the Linux image.
- 3** The Linux IUCV device driver receives the data and passes them to the `lfpd` running on the Linux image. The `lfpd` processes the data and translates it into a socket call.
- 4** The socket call is processed by the TCP/IP stack. Because the data is to be sent to an application (in this example, the DB2 Server) that runs on the *same* Linux system, the TCP/IP stack simply forwards the data directly to the DB2 Server. TCP/IP is *not* required to perform the processing-intensive steps that were required in Figure 22 on page 390.
- 5** The DB2 Server receives and processes the data.

For data to be sent from the DB2 Server on Linux back to the DB2 client on z/VSE, the same five steps as above are performed in *reverse order*.

Communication Flow When Using Linux Fast Path in an LPAR Environment

This topic provides a detailed description of a typical configuration that uses a Linux Fast Path in an LPAR environment. This is shown in Figure 24 on page 392. This configuration also uses the example of a DB2 client running under z/VSE communicating with the DB2 Server for Linux running under Linux.

In Figure 24:

- The communication between z/VSE and Linux on System z is established using a HiperSockets communication path.
- LFP establishes a HiperSockets connection between z/VSE and Linux on System z *without* using:
 - A TCP/IP stack on z/VSE
 - OSA Express adapters

For an LFP connection via HiperSockets, both systems must run in LPAR mode. The flow of actions described in this example also apply, if running other z/VSE-supplied programs that run on Linux (the VSE Redirector Server, VSE Script Server, Redirector Handler, and so on).

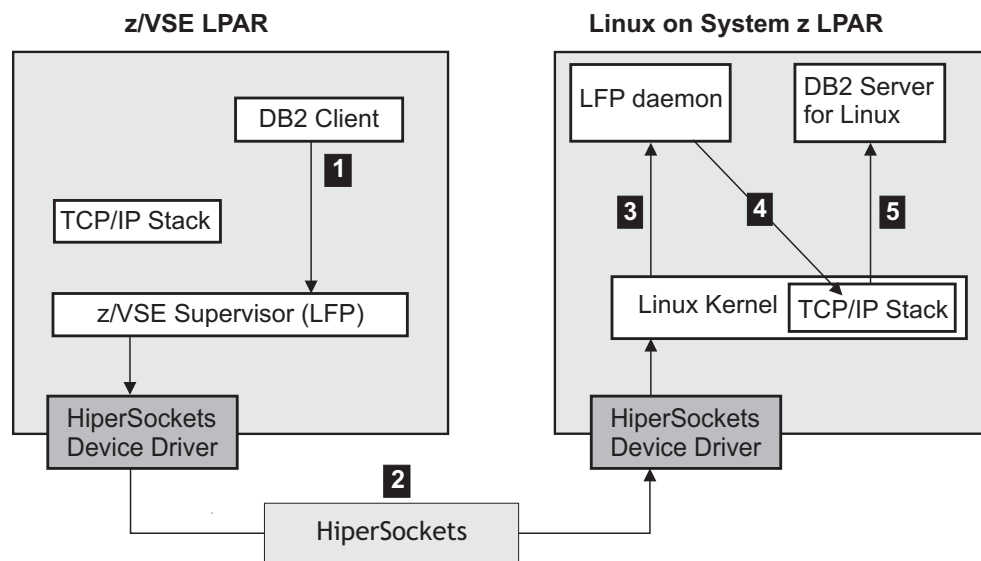


Figure 24. Communication Between LPARs via HiperSockets

The processing shown in Figure 23 on page 391 is as follows:

- 1** The data to be sent is passed to LFP running on z/VSE.
- 2** LFP invokes the HiperSockets device driver to send the data to the Linux image. The HiperSockets Completion Queue function guarantees successful data transmission.
- 3** The Linux device driver receives the data and passes it to the lfpd running on the Linux image. The lfpd then processes the data received and translates it into a socket call.
- 4** The socket call is processed by the TCP/IP stack. Because the data is to be sent to an application (in this example, the DB2 Server) that runs on the *same* Linux system, the TCP/IP stack simply forwards the data directly to the DB2 Server. TCP/IP is *not* required to perform the processing-intensive steps that were required in Figure 22 on page 390.
- 5** The DB2 Server receives and processes the data.

For data to be sent from the DB2 Server on Linux back to the DB2 client on z/VSE, the same five steps as above are performed in *reverse order*.

Preparing Linux on System z to Use Linux Fast Path

The LFP daemon (*lfpd*) that runs on Linux on System z must be installed before it can be used, to do so you have to be logged on as user root.

There are two ways to obtain a copy of the LFP daemon:

1. From the Internet

- To obtain the client from the Internet go to the z/VSE website at <http://www.ibm.com/systems/z/os/zvse/downloads/>.

Go to the *Linux Fast Path* section and download the file *vselfpdnnn.zip* to a temporary directory to install the Linux Fast Path.

Note: *nnn* refers to the current VSE version (for example, *vselfpd510.zip*).

2. From the Extended Base Tape

- To obtain the client from the Extended Base Tape, install the VSE Connectors Workstation Code component 5686CF8-38 CONN.C/W located on the Extended Base Tape.

After you have installed the VSE Connectors Workstation Code component, the *lfpd* W-book *ijblfplx.w* is stored in z/VSE sublibrary PRD2.PROD.

Use the FTP (file transfer program) utility of TCP/IP for VSE/ESA to download *ijblfplx.w* to a temporary directory with the following settings:

- Download *ijblfplx.w* in **binary** mode.
- Make sure that UNIX mode is turned **off**. Otherwise *ijblfplx.w* will be downloaded in ASCII mode, even if you specify binary. Unix mode is one parameter of your VSE FTP daemon. Some FTP clients might force Unix mode to be turned on!

The example below shows a successful transfer of *ijblfplx.w* using a batch FTP client. The UNIX mode setting is highlighted in bold.

```
linlfp:/root/vselfpd # ftp 10.0.0.1
Connected to 10.0.0.1.
220-TCP/IP for VSE -- Version 01.05.F -- FTP daemon
Copyright (c) 1995,2006 Connectivity Systems Incorporated

220 Service ready for new user.
User (10.0.0.1:(none)): sysa
331 User name okay, need password.
Password:
230 User logged in, proceed.
ftp> cd prd2
250 Requested file action okay, completed.
ftp> cd prod
250 Requested file action okay, completed.
ftp> binary
200 Command okay
ftp> get ijblfplx.w
200 Command okay.
150-File: PRD2.PROD.IJBLFPLX.W
  Type: Binary Recfm: FB Lrecl:   80 Blksize:   80
  CC=ON UNIX=OFF RECLF=OFF TRCC=OFF CRLF=ON NAT=NO
150 File status okay; about to open data connection
226-Bytes sent:          123,456
  Records sent:           12,345
  Transfer Seconds:       16.52 (   290K/Sec)
  File I/O Seconds:       3.94 (  1,548K/Sec)
```

```
226 Closing data connection.  
123456 bytes received in 17,12 seconds (277,91 Kbytes/sec)  
ftp> bye  
221 Service closing control connection.
```

Rename the downloaded file to `ijblfplx.zip` with the following command:

```
linlfp:/root/vselfpd # mv IJBLFPLX.W ijblfplx.zip
```

Extract the zip file:

```
linlfp:/root/vselfpd # unzip ijblfplx.zip
```

The rpm installation package `vselfpd-1.3.0-1.s390x.rpm` is extracted from the zip file. Note that the actual file name can be different depending on the version or release.

To install the rpm installation package invoke the rpm program:

```
linlfp:/root/vselfpd # rpm -i vselfpd-1.3.0-1.s390x.rpm
```

If an older version of `vselfpd` is already installed, you can upgrade the installed package:

```
linlfp:/root/vselfpd # rpm -U vselfpd-1.3.0-1.s390x.rpm
```

The rpm package installs the following files:

Usage File location

lfpd binary

`/opt/ibm/vselfpd/sbin/lfpd`

lfpd admin tool binary

`/opt/ibm/vselfpd/sbin/lfpd-admin`

lfpd control script

`/opt/ibm/vselfpd/sbin/lfpd-ctl`

lfpd SysV init script

`/etc/init.d/vselfpd`

profile for user root to establish path to the lfpd binaries

`/etc/profile.d/vselfpd.sh`

man pages for the command line programs

`/usr/share/man/man8/lfpd.8.gz`

`/usr/share/man/man8/lfpd-admin.8.gz`

`/usr/share/man/man8/lfpd-ctl.8.gz`

directory containing available configurations, and a sample configuration

`/etc/opt/ibm/vselfpd/confs-available/`

`/etc/opt/ibm/vselfpd/confs-available/lfpd-SAMPLE.conf`

directory containing enabled configurations

`/etc/opt/ibm/vselfpd/confs-enabled`

Note: The logon profile sets the `PATH` environment variable to the `/opt/ibm/vselfpd/sbin` directory. The `PATH` is set only for the user `root`, because only this user can start the `lfpd`. The logon profile is only executed, if you

log on as user **root** **directly**. If you are logged on to the system as another user, you can switch to user **root** with the command `su -`, but the logon profile is **not** executed and the `PATH` is not set.

If you want to run LFP via HiperSockets, you must set the `qeth sysfs` attribute `hsuid` of the HiperSockets device / link (e.g. 0.0.8200 / hsi0), before you can start the `lfpd`. The system generates an IPv6 link-local address from the specified `hsuid`. Therefore the `hsuid` must be unique across the HiperSockets network and in the system, which means you can not have 2 HiperSockets devices (for example `hsi0` and `hsi1`) with the same `hsuid` name.

The `hsuid` attribute value is case sensitive. For use with LFP, you must specify the value in **all** uppercase. Do not specify an IP address or subnet mask for the HiperSockets device. A HiperSockets device can either be used for LFP, or it can be used for a regular TCP/IP network, but not for both at the same time.

If you plan to use the auto start function (via `init.d`) of `lfpd` to start configured LFP instances at boot time, the `hsuid` must be set before `lfpd` gets started. To configure the `qeth sysfs` attribute `hsuid` use the network configuration tools provided by the Linux distribution. Set the `QETH` option "`hsuid=nnnnnn`" in the `sysconfig` configuration files. For SuSE distributions you can use `YAST` to configure.

To set the `hsuid` manually, you can use the following commands (this example uses link name `hsi0` and device address `8200`):

```
ifconfig hsi0 down
echo 0 > /sys/bus/ccwgroup/devices/0.0.8200/online
echo "SYSNAME " > /sys/bus/ccwgroup/devices/0.0.8200/hsuid
echo 1 > /sys/bus/ccwgroup/devices/0.0.8200/online
ifconfig hsi0 up
```

To avoid out of memory situations of the linux kernel in both VM and LPAR environments, you should increase the following values to at least 1MB:

```
sysctl -w net.core.rmem_max=1048576
sysctl -w net.core.rmem_default=1048576
sysctl -w net.core.wmem_max=1048576
sysctl -w net.core.wmem_default=1048576
```

If you are using a huge MTU size and a high message limit (`HS_MSGLIMIT` or `IUCV_MSGLIMIT`), you might have to increase these settings even more. The `sysctl` commands must be performed before an `lfpd` is started.

Preparing to use Socket APIs with Linux Fast Path on z/VSE

Use the skeleton `SKLFPACT` in ICCF library 59 to do all necessary changes to the z/VSE system that allow applications to use LFP. Some of the steps in `SKLFPACT` must be repeated after every IPL. Some steps should be added to the `JCL` when running the applications (e.g. `SYSPARM` settings).

LE/C Socket API

The LE/C socket API is implemented in `$EDCTCPV.PHASE`. The following `$EDCTCPV` phases can exist in your system:

- The LE Dummy phase located in `PRD2.SCEEBASE`.
- The interface phase for TCP/IP for VSE/ESA located in `PRD2.TCPIP.C`.
- Other TCP/IP stacks located in other libraries.

The LIBDEF SEARCH statement controls which interface phase is used.

The LE/C interface phase for LFP is shipped as IJBLFPLE.PHASE in IJSYSRS.SYSLIB. You must configure the LE/C TCP/IP Socket API Multiplexer to use the LFP LE/C TCP/IP interface phase IJBLFPLE for the IDs of all LFP instances that are running. To configure the multiplexer, use skeleton EDCTCPMC in ICCF library 62.

You can add entries for all your LFP instances with the following command:

```
EDCTCPME SYSID='01',PHASE='IJBLFPLE'
```

Submit the skeleton and make sure you have a return code zero. If you use TCP/IP sockets under CICS, you might have to reload the multiplexer configuration with the following command to activate it:

```
CEMT SET PROG(EDCTCPMC) NEW
```

Refer to Chapter 10, “TCP/IP Support for the LE/VSE C Socket Interface,” on page 85 for details.

Note: The LFP LE/C TCP/IP interface phase IJBLFPLE only works for LFP, not with any other TCP/IP stack.

EZA SOCKET and EZASMI interfaces

With the EZA socket and EZASMI interfaces you can specify which interface module is to be used. For LFP, you must use the EZA interface module IJBLFPEZ.

You must set the JCL parameter “EZA\$PHA” in all your jobs that you want to use LFP. To do so use the following command:

```
// SETPARM [SYSTEM,]EZA$PHA=IJBLFPEZ
```

If you are using the EZA SOCKET or EZASMI interface under CICS, you need to activate the EZA 'TASK-RELATED-USER-EXIT' (TRUE). Refer to “CICS Considerations for the EZA Interfaces” on page 83 for details.

Specifying the instance ID

Table 10 shows how to specify instance IDs and where they can be applied. The settings are checked from top to bottom as listed in the table

Table 10. Specifying Instance IDs

	LE/C API	EZA API	CSI SOCKET Macro
'LFP\$ID' (environment variable)	x		
// SETPARM [SYSTEM,]LFP\$ID=NN	x	x	
'SYSID' (environment variable)	x		
IDENT.TCPNAME passed to INITAPI call		x	
ID parameter on SOCKET macro			x
// OPTION SYSPARM='NN'	x	x	x
Default '00'	x	x	x

If no ID is specified, an ID of '00' is used.

Note: If you are using SSL functions (gsk_*), you need to have a CSI TCP/IP stack running, even if you are using LFP for network communication. In this case, the ID of the CSI TCP/IP stack must be specified with the // OPTION SYSPARM statement or the // SETPARM [SYSTEM,]LFP\$SSL=NN statement. The ID of the LFP Instance must be different.

CICS task isolation options

LFP isolates CICS tasks from each other. This means that sockets that are allocated by one CICS task, cannot be used by another CICS task, except the socket is passed to the other CICS task via GIVESOCKET/TAKESOCKET calls. However, some programs rely on passing sockets from one CICS task to another without the use of GIVESOCKET/TAKESOCKET. To allow such programs to work with LFP, you need to specify the following JCL statement for the program:

```
// SETPARM [SYSTEM,]LFP$CIC=SHARE
```

This setting applies to the LE/C socket interface as well as the EZA interfaces.

For example, DB2 (client or server) application requestor requires socket sharing, if running under CICS.

Configuring Linux Fast Path

To enable LFP you have to set several configuration parameters in z/VM, Linux on System z and z/VSE. Setup and enable the configurations in the same order as described in the following sections.

Note: If a configuration parameter description says that a decimal number is required, this number indicates bytes if not stated otherwise.

z/VM

In a z/VM environment LFP uses IUCV as the underlying communication vehicle. The z/VSE and the Linux on System z guests need to be configured for IUCV. For details about the parameters refer to the z/VM Documentation.

The following z/VM parameters for the guest systems are relevant:

IUCV ALLOW

Allows any other virtual machine to establish a communication path with this virtual machine. No further authorization is required in the virtual machine that initiates the communication.

IUCV ANY

Allows this z/VM guest virtual machine to establish a communication path with any other z/VM guest virtual machine.

IUCV MSGLIMIT

MSGLIMIT defines the maximum number of outstanding messages allowed on any authorized path. The value must be in the range of 16 to 65535. If MSGLIMIT is specified in the z/VM guest virtual machine configuration **and** the z/VSE LFP configuration, the lowest value of both applies to the IUCV communication path.

This parameter is optional.

OPTION MAXCONN *maxno*

Specifies the maximum number of IUCV connections allowed for this virtual machine. If the MAXCONN operand is omitted, the default is 64. The maximum is 65535.

Note: z/VSE can handle up to 10 parallel IUCV connections.

Linux on System z

Each LFP daemon (lfpd) has its own configuration. It is read during the start of lfpd and cannot be changed while lfpd is running. The configuration has to be provided in a file.

Note: The lfpdcontrol script lfpdctl requires a naming convention for the configuration file names. Each configuration file must be named lfpd-XXX.conf, where XXX is the IUCV_SRC_APPNAME or HS_SRC_APPNAME specified in the configuration file. The XXX characters in the file name must be specified in uppercase. For example, the configuration file with an IUCV_SRC_APPNAME of LINR02 is named lfpd-LINR02.conf.

You have to store all available configuration files in the directory /etc/opt/ibm/vsel fpd/confs-available. The directory /etc/opt/ibm/vsel fpd/confs-enabled contains the configuration files that are enabled. If a configuration is enabled it can easily be started with the lfpd control script lfpdctl. Additionally the SysV init script will start all enabled configurations during startup of the Linux system. For each available configuration that is to be enabled, you have to create a symbolic link.

For example, you have created a configuration with the name LINR02, the configuration file is /etc/opt/ibm/vsel fpd/confs-available/lfpd-LINR02.conf. Now you plan to enable the configuration. To place a symbolic link from the configuration file to the enabled configurations directory use the following command:

```
ln -s /etc/opt/ibm/vsel fpd/confs-available/lfpd-LINR02.conf
    /etc/opt/ibm/vsel fpd/confs-enabled
```

Each parameter in the configuration file must be specified as key = value. Empty lines in the configuration are ignored. The character “#” at the beginning of a line indicates a comment line, which is also ignored. The configuration parameters are not case sensitive.

IUCV specific configuration

These parameters are only used with IUCV connections.

IUCV_SRC_APPNAME

The value for IUCV_SRC_APPNAME must be specified as 1 to 8 alphanumeric characters. The value defines a local port and must be unique for the Linux on System z system. This parameter value must match with the IucvDestAppName used for an LFP instance on z/VSE. This parameter is mandatory.

PEER_IUCV_APPNAME

The value for PEER_IUCV_APPNAME must be specified as 1 to 8 alphanumeric characters. The value defines the IUCV application name of an instance on z/VSE (IucvSrcAppName). If this parameter is set, lfpd checks the name of the source application of any incoming IUCV connection. If the name does not match the name in the configuration, the incoming connection is revoked. If the parameter PEER_IUCV_APPNAME

is not specified in the configuration, the name of the source application of an incoming IUCV connection is not checked. This parameter is optional.

PEER_IUCV_VMID

The value for `PEER_IUCV_VMID` must be specified as 1 to 8 alphanumeric characters. The value defines the name of a z/VM guest system. If this parameter is set, `lfpd` CHECKS the name of the source z/VM guest of any incoming IUCV connection. If the name does not match the name in the configuration, the incoming connection is revoked. If the parameter `PEER_IUCV_VMID` is not specified in the configuration, the name of the source z/VM guest of an incoming IUCV connection is not checked. This parameter is optional.

IUCV_MSGLIMIT

The value for `IUCV_MSGLIMIT` must be a decimal number in the range of 16 to 65535. The `IUCV_MSGLIMIT` value specifies the maximum number of outstanding IUCV messages that are not yet received by an instance on z/VSE. The recommended value is 1024, higher values might result in a better performance, but need more memory on z/VSE. This parameter is mandatory.

IUCV_MTU_SIZE | MTU_SIZE

The value for `IUCV_MTU_SIZE` (`MTU_SIZE` is also accepted) must be a decimal number in the range of 256 to 65535 (64 KB). `IUCV_MTU_SIZE` specifies the maximum size of packets that are transferred. The recommended value is 8192 (8 KB), higher values might result in a better performance, but need more memory. The `IUCV_MTU_SIZE` value must exactly match the `IucvMTU` value specified for the instance on z/VSE. This parameter is mandatory.

Note: If you are using z/VSE VIA, you must specify `MTU_SIZE`. `IUCV_MTU_SIZE` is not accepted.

HiperSockets specific configuration

These parameters are only used with HiperSockets connections.

HS_MSGLIMIT

The value for `HS_MSGLIMIT` must be a decimal number in the range of 1 to 128. The default value is 128. `HS_MSGLIMIT` specifies the maximum number of outstanding messages that are not yet received by `lfpd`. Higher values might result in a better performance, but need more memory. This parameter is optional.

On a dedicated HiperSockets network the value should match the `buffer_count` value that is configured for the HiperSockets device. The default value for `buffer_count` is also 128. This parameter is optional.

HS_SRC_APPNAME

The value for `HS_SRC_APPNAME` must be specified as 1 to 8 alphanumeric characters. The value defines a local port and must be unique for the Linux on System z system. This parameter value must match with the `HSDestAppName` used for an LFP instance on z/VSE. This parameter is mandatory.

HS_SRC_SYSTEMNAME

The value for `HS_SRC_SYSTEMNAME` must be specified as 1 to 8 alphanumeric characters. The value defines the `hsiud` of the HiperSockets device that `lfpd` will use. Internally the `hsiud` is converted to an IPv6 link-local address that is used to initialize the HiperSockets device. Multiple `lfpds` can run on the same HiperSockets device, if each one uses

a different HS_SRC_APPNAME. This parameter value must match with the HsDestSystemName used for an LFP instance on z/VSE. This parameter is mandatory.

PEER_HS_APPNAME

The value for PEER_HS_APPNAME must be specified as 1 to 8 alphanumeric characters. The value defines the HsSrcAppName of an instance on z/VSE. If this parameter is set, l fpd checks the name of the source application of any incoming connection. If the name does not match the name in the configuration, the incoming connection is revoked. If PEER_HS_APPNAME is not specified in the configuration, the name of the source application of an incoming connection is not checked. This parameter is optional.

PEER_HS_SYSTEMNAME

The value for PEER_HS_SYSTEMNAME must be specified as 1 to 8 alphanumeric characters. The value defines the HsSrcSystemName of an instance on z/VSE. If this parameter is set, l fpd checks the name of the source system of any incoming connection. If the name does not match the name in the configuration, the incoming connection is revoked. If PEER_HS_SYSTEMNAME is not specified in the configuration, the name of the source system of an incoming connection is not checked. This parameter is optional.

Common configuration

These parameters can be used with both IUCV and HiperSockets connections.

DEVICETYPE

The value for DEVICETYPE must be either *IUCV* or *HS*. If *IUCV* is specified, *IUCV* will be used as transport mechanism. *HS* specifies that HiperSockets should be used. *IUCV* is the default. This parameter is optional.

Note: If you are using z/VSE VIA, you must omit this parameter. z/VSE VIA always uses *IUCV* as transport mechanism, and does not accept the DEVICETYPE parameter.

INITIAL_IO_BUFS

The value for INITIAL_IO_BUFS must be a decimal number. The INITIAL_IO_BUFS value specifies the count of I/O buffers that are allocated during startup of l fpd. L fpd allocates buffers of a fixed size to send and receive data. The size of a buffer mainly depends on the MTU size of a connection to an instance on z/VSE. During runtime, l fpd automatically allocates more buffers if needed. A recommended value is 128. This parameter is optional.

MAX_SOCKETS

The value for MAX_SOCKETS must be a decimal number. The MAX_SOCKETS value specifies the maximum possible count of sockets opened by l fpd. A normal user on a Linux system is usually allowed to open a maximum of 1024 sockets per program. This value might be too low for an l fpd. Therefore you can set the MAX_SOCKETS value to a higher value. L fpd started under the root user will set the maximum socket limit to the specified value. For performance reasons MAX_SOCKETS must be a multiple of 256. The default value is 1024. This parameter is optional.

MAX_VSE_TASKS

The value for the MAX_VSE_TASKS parameter must be a decimal number.

The `MAX_VSE_TASKS` value specifies the maximum count of tasks running on z/VSE. The value depends on the setup of your z/VSE system. The default is 512. This parameter is optional.

WINDOW_SIZE

The value for `WINDOW_SIZE` must be a decimal number. The value specifies the size of the lfpd receive window for a socket. The lfpd receive window defines the amount of data that can be received from an instance on z/VSE by lfpd without lfpd actually forwarding the data to the external socket. Higher values can result in a better performance, but need more memory. The minimum value is 4096. A recommended value is 65535.

WINDOW_THRESHOLD

The value for `WINDOW_THRESHOLD` must be a decimal number in the range of 1 to 100, indicating percentage. The `WINDOW_THRESHOLD` specifies the percentage the `WINDOW_SIZE` must grow until lfpd sends the notification to the LFP instance. Higher values can result in a better performance, but need more memory. A recommended value is 25.

VSE_CODEPAGE

The value for `VSE_CODEPAGE` must be a string that specifies a codepage known to `iconv` on the Linux system. For functions that receive or return textual data (such as `gethostbyname()`) lfpd needs to do an EBCDIC (z/VSE) <-> ASCII (Linux) translation. Lfpd uses the specified `VSE_CODEPAGE` as EBCDIC codepage. A possible value is EBCDIC-US.

LINUX_CODEPAGE

The value for `LINUX_CODEPAGE` must be a string that specifies a codepage known to `iconv` on the Linux system. This codepage is the counterpart of `VSE_CODEPAGE`. The default is 850. This parameter is optional.

SUPPORT_GETXXXENT

The value for `SUPPORT_GETXXXENT` must be either *yes* or *no*. *Yes* enables the support for the functions `gethostent()`, `getprotoent()`, `getnetent()`, `getservent()`. The support takes approximately 500 KB of memory on Linux per running lfpd. The default is *yes*. This parameter is optional.

VSE_HOSTID

The value for `VSE_HOSTID` must be a valid IPv4 IP address. Lfpd uses the IP address as local host id, and returns this address, when z/VSE applications run the `getHostId()` function. This setting together with `RESTRICT_TO_HOSTID` allows to “bind” applications running on z/VSE to a specific IP address. Note that outgoing connections started by applications running on z/VSE are not checked for the network device they use. To prevent outgoing connections from leaving the specified network device, you have to configure firewall rules.

VSE_HOSTID6

The value for `VSE_HOSTID6` must be a valid IPv6 IP address. If `RESTRICT_TO_HOSTID` is enabled, this parameter is mandatory. Together these parameters bind applications running on z/VSE to a specific IP address. Note that outgoing connections started by applications running on z/VSE are not checked for the network device they use. To prevent outgoing connections from leaving the specified network device, you have to configure firewall rules.

RESTRICT_TO_HOSTID

The value for `RESTRICT_TO_HOSTID` must be either *yes* or *no*. If *yes* is specified, lfpd will restrict all `bind()` calls to use only the configured

VSE_HOSTID. This prevents applications running on z/VSE from binding to any Linux device or to a specific Linux IP address. The default is *yes*. This parameter is optional.

LOG_INFO_MSG

The value for LOG_INFO_MSG must be either *yes* or *no*. By default lfpd writes messages of type EMERG, ALERT and ERR to the system logger (syslogd). If LOG_INFO_MSG is set to *yes*, lfpd will additionally write INFO messages to the system logger. The default is *no*. This parameter is optional.

z/VSE

Each LFP instance has its own configuration. The configuration is read during the start of an instance and cannot be changed while the instance is running. Make sure you have prepared the socket APIs for LFP as described in “Preparing to use Socket APIs with Linux Fast Path on z/VSE” on page 395 before you proceed.

The LFP operator tool IJBLFPOP processes the configuration, when the user has requested the start of an instance. IJBLFPOP takes the location of the configuration as a parameter. The configuration is provided in a library member or directly in the job input (SYSIPT) of the job that invokes IJBLFPOP.

Each configuration parameter must be specified as *key = value*. Empty lines in the configuration are ignored. The characters “*” and “#” at the begin of a line indicate a comment line, which is also ignored. The configuration parameters are not case sensitive.

IUCV specific configuration

These parameters are only used with IUCV connections.

IucvMTU | MTU

The value for IucvMTU (MTU is also accepted) must be a decimal number in the range of 256 to 65535. A recommended value is 8192. The IucvMTU size specifies the maximum size of packets that are transferred. Higher values might result in a better performance, but also need more SVA PFIX memory. The IucvMTU value must exactly match the IUCV_MTU_SIZE specified in the target lfpd running on Linux on System z. This parameter is mandatory.

IucvMsgLimit

The value for IucvMsgLimit must be a decimal number in the range of 16 to 65535. The IucvMsgLimit specifies the maximum number of outstanding IUCV messages that have not yet been received by lfpd on Linux. The recommended value is 1024, higher values might result in a better performance, but need more SVA PFIX memory. This parameter is mandatory.

IucvSrcAppName

The value for IucvSrcAppName must be specified as 1 to 8 alphanumeric characters. The value defines an entry in the z/VSE IUCV application table and must be unique for the z/VSE system. An lfpd on Linux checks this value to ensure that only a specific z/VSE LFP instance can connect to this lfpd on Linux. This parameter is mandatory.

IucvDestAppName

The value for IucvDestAppName must be specified as 1 to 8 alphanumeric characters. The value must match the IUCV_SRC_APPNAME configuration

parameter for an lfpd that is running on Linux. The instance will connect to the lfpd with this IUCV_SRC_APPNAME. This parameter is mandatory.

IucvDestVmId

The value for IucvDestVmName must be specified as 1 to 8 alphanumeric characters. The value specifies the z/VM user ID of the target Linux on System z that runs an lfpd. This parameter is mandatory.

PacketConsolidationThreshold

The value for PacketConsolidationThreshold is specified in bytes. The maximum allowed value is the configured MTU size. The default is 100 bytes. This parameter is used to optimize the trade-off between CPU consumption and memory consumption. The following conditions apply:

- A new packet containing user data has arrived.
- The packet size does not exceed the PacketConsolidationThreshold.
- Another packet is already enqueued for the target task.
- The data packet already enqueued has enough free space to include the data of the packet that just arrived.
- The socket to which the packet belongs is a STREAM socket.

If all these conditions are met, the data bytes of the newly arrived packet are copied into the free space of the first packet. The buffer space previously needed for the second package can now be reused. This parameter is optional.

HiperSockets specific configuration

These parameters are only used with HiperSockets connections.

HsDevices

The value for HsDevices must be specified as 3 device addresses (CUUs) which are separated by commas: cuu1,cuu2,cuu3 (where cuu2 must always be cuu1+1). The device addresses must represent a HiperSockets device and must be defined with the device type OSAX in the IPL procedure. This parameter is mandatory.

Note: For HiperSockets devices, the MTU size/frame size is configured in the IOCP. Larger MTU sizes might require more buffer space. For details on configuring HiperSockets devices in IOCP refer to *z/VSE Administration*.

HsSrcAppName

The value for HsSrcAppName must be specified as 1 to 8 alphanumeric characters. The value defines a local port and must be unique in combination with the HsSrcSystemName for the z/VSE system. An lfpd on Linux checks this value to ensure that only a specific z/VSE LFP instance can connect to this lfpd on Linux. This parameter is mandatory.

HsSrcSystemName

The value for HsSrcSystemName must be specified as 1 to 8 alphanumeric characters. The value defines the system name of the LFP instance. The name must be unique across the HiperSockets network that is used. Internally the value is converted to an IPv6 link-local address that is used to initialize the HiperSockets device. An lfpd on Linux checks this value to ensure that only a specific z/VSE LFP instance can connect to this lfpd on Linux. This parameter is mandatory.

HsDestAppName

The value for HsDestAppName must be specified as 1 to 8 alphanumeric characters. The value must match the HS_SRC_APPNAME configuration

parameter for an lfpd that is running on Linux. The instance will connect to the lfpd with this HS_SRC_APPNAME. This parameter is mandatory.

HsDestSystemName

The value for HsDestSystemName must be specified as 1 to 8 alphanumeric characters. The value must match the HS_SRC_SYSTEMNAME configuration parameter (*hsuid*) for an lfpd that is running on Linux. The instance will connect to the lfpd with this HS_SRC_SYSTEMNAME. This parameter is mandatory.

HsKeepAliveTime

The value for HsKeepAliveTime must be a decimal number, which represents seconds. The default is 5 seconds. The keep alive mechanism is used to detect an unexpected termination of the connection between z/VSE and Linux. The HsKeepAliveTime value controls the frequency of exchanged keep alive packets. This parameter is optional.

HsMsgLimit

The value for HsMsgLimit must be a decimal number in the range of 8 to 128 and it must be a multiple of 8. The HsMsgLimit value specifies the maximum number of outstanding messages that are not yet received by the LFP instance. Higher values might result in a better performance, but will need more memory . A recommended value is 32. This parameter is mandatory.

Common configuration

These parameters can be used with both IUCV and HiperSockets connections.

DeviceType

You can specify *IUCV* or *HS* as device type. *IUCV* is the default. This parameter is optional.

Id

The value for the Id parameter must be a 2-digit decimal number, in the range of 00 to 99. This number is the instance id of the instance that will be started. Each instance can only be started once, therefore no instance with the same id should be running. A job with the statement OPTION SYSPARM='xx' will use the LFP instance with the id 'xx'. Other TCP/IP stacks can also use the OPTION SYSPARM statement to specify the id of the stack to use. If another stack is started with a specific id, an LFP instance with the same id cannot be started. This parameter is mandatory.

InitialBufferSpace

The value for InitialBufferSpace can be specified

- as a decimal number indicating bytes (for example 512),
- in kilobyte (for example 1024K),
- or in megabyte (for example 2M).

The value specifies the memory size that the instance will initially allocate for packet buffers during startup. The memory is allocated as SVA PFIX in the ANY area. The minimum value is 256 K. A recommended value for an MTU size of 8K is 512 K. The maximum value depends on your system layout.

Note: The amount of buffer space needed also depends on the MTU size. Larger MTU sizes might require more buffer space.

MaxBufferSpace

The value for MaxBufferSpace can be specified

- as a decimal number indicating kilobytes (for example 512),

- in kilobyte (for example 1024K),
- or in megabyte (for example 2M).

The value specifies the maximum memory size that the instance will allocate for packet buffers. The memory is allocated as SVA PFX in the ANY area. The minimum value must not be lower than the value of the InitialBufferSpace parameter. A recommended value is 4 MB. The maximum value depends on your system layout.

WindowSize

The value for WindowSize must be a decimal number. The value specifies the size of the receive window for a socket. The receive window defines the amount of data that can be received, without the application actually receiving the data from the socket. The recommended value is 65535 (64 KB), higher values might result in a better performance, but need more memory. The minimum value is 4096.

WindowThreshold

The value for WindowThreshold must be a decimal number in the range of 1 to 100, indicating percentage. The WindowThreshold specifies the percentage the WindowSize must grow until the LFP instance sends the notification to lfpd. The recommended value is 25, higher values might result in a better performance, but need more memory.

Sample configurations

Figure 25 shows a z/VSE startup job containing a configuration via IUCV with the following sample entries:

- The LFP instance should connect to a Linux system with the z/VM user ID LINLFP.
- The IUCV application name of the lfpd on Linux is LINR02.
- The instance on z/VSE has the IUCV application name TESTV.
- The z/VM user ID of the z/VSE system is VSER05. This name is specified in the lfpd configuration on Linux.

```
* $$ JOB JNM=LFPSTART,CLASS=0,DISP=L
// JOB LFPSTART
// EXEC IJBLFPOP,PARM='START DD:SYSIPT LOGALL'
* Instance ID
ID                = 01
MTU               = 8192
IucvMsgLimit     = 1024
InitialBufferSpace = 512 K
MaxBufferSpace   = 4M
IucvSrcAppName   = TESTV
IucvDestAppName  = LINR02
IucvDestVMId     = LINLFP
WindowSize       = 65535
WindowThreshold  = 25
/*
/&
* $$ E0J
```

Figure 25. z/VSE LFP via IUCV Configuration Example

The following configuration for an lfpd on Linux is the counterpart to the LFP instance configuration on z/VSE that is shown above.

```

# lfpd configuration file

IUCV_SRC_APPNAME = LINR02

# ensure that only TESTV from VSER05 can connect

PEER_IUCV_VMID      = VSER05
PEER_IUCV_APPNAME   = TESTV

IUCV_MSGLIMIT       = 1024

MTU_SIZE             = 8192
MAX_SOCKETS          = 1024
INITIAL_IO_BUFS     = 128

WINDOW_SIZE         = 65535
WINDOW_THRESHOLD    = 25

VSE_CODEPAGE        = EBCDIC-US

VSE_HOSTID           = 10.0.0.1
RESTRICT_TO_HOSTID  = yes

LOG_INFO_MSG        = no

```

Figure 26. Linux LFP via IUCV Configuration Example

- IUCV_SRC_APPNAME defines the name of the lfpd configuration file, it has to match the IucvDestAppName defined in the z/VSE configuration file.
- PEER_IUCV_VMID and PEER_IUCV_APPNAME ensure that only TESTV (defined in the z/VSE configuration file) from user ID VSER05 (z/VM user ID of the z/VSE system) can connect.

Figure 27 shows a z/VSE startup job containing a configuration via HiperSockets with the following sample entries:

- The LFP instance should connect to a Linux system with the system name LINXLPAR.
- The application name of the lfpd in Linux is LNXXSYS1.
- The instance on z/VSE has the application name TESTV.
- The system name of the LFP instance in z/VSE is VSELPAR.

```

* $$ JOB JNM=LFPSTART,CLASS=0,DISP=L
// JOB LFPSTART
// EXEC IJBLFPOP,PARM='START DD:SYSIPT LOGALL'
* Instance ID
ID = 02
InitialBufferSize = 1M
MaxBufferSize = 4M
WindowSize = 65535
WindowThreshold = 25
DeviceType = HS
HSDevices = 500,501,502
HSMsgLimit = 128
HSSrcAppName = TESTV
HSDestAppName = LNXXSYS1
HSSrcSystemName = VSELPAR
HSDestSystemName = LINXLPAR
HSKeepAliveTime = 30
/*
/&
* $$ E0J

```

Figure 27. z/VSE LFP via HiperSockets Configuration Example

The following configuration for an lfpd on Linux is the counterpart to the LFP instance configuration on z/VSE that is shown above.

```
# lfpd sample configuration file
#
DEVICETYPE      = HS
HS_MSGLIMIT     = 128
HS_SRC_APPNAME  = LNXXSYS1
HS_SRC_SYSTEMNAME = LNXLPAR

# ensure that only TESTV from VSELPAR can connect
PEER_HS_APPNAME = TESTV
PEER_HS_SYSTEMNAME = VSELPAR

MAX_SOCKETS     = 1024
INITIAL_IO_BUFS = 128

WINDOW_SIZE     = 65535
WINDOW_THRESHOLD = 25

VSE_CODEPAGE    = EBCDIC-US
VSE_HOSTID      = 10.0.0.1
RESTRICT_TO_HOSTID = no

LOG_INFO_MSG = no
```

Figure 28. Linux LFP via HiperSockets Configuration Example

- HS_SRC_APPNAME defines the name of the lfpd configuration file, it has to match the HS_DEST_APPNAME defined in the z/VSE configuration file.
- PEER_HS_SYSTEMNAME and PEER_HS_APPNAME ensure that only TESTV (defined in the z/VSE configuration file) from user VSELPAR (z/VM guest) can connect.

Starting and Stopping Linux Fast Path

z/VSE

Starting an LFP instance

To start an LFP instance on z/VSE use the LFP operator program IJBLFPOP. The invocation parameter format is:

```
EXEC IJBLFPOP,PARM='START <CFGFILE> [LOGALL]'
```

- All parameters are positional and must appear in the displayed order.
- CFGFILE is a mandatory parameter that specifies from where IJBLFPOP reads the instance configuration.
- CFGFILE has to be declared as a valid Language Environment file name preceded by DD:.
- If the configuration is in member LFPCFG00.L in library PRD2.CONFIG, specify CFGFILE as DD:PRD2.CONFIG(LFPCFG00.L).
- If the configuration is inside the job that executes IJBLFPOP, specify CFGFILE as DD:SYSIPT.

- LOGALL is an optional parameter. If the parameter is specified, IJBLFPOP writes **all** messages to the console. If the parameter is omitted, only error messages are printed to the console. All messages are always printed into the job listing.
- The skeleton SKLFPSTA in ICCF library 59 can be used to create an LFP instance startup job which contains the instance configuration in the job itself.

The instance was started successfully, if the following message is shown on the console:

```
LFPB013I  STARTED LFP INSTANCE '00'
```

Stopping an LFP instance

To stop an LFP instance on z/VSE use the LFP operator program IJBLFPOP. The invocation parameter format is:

```
EXEC IJBLFPOP,PARM='STOP <INSTID>'
```

- All parameters are positional and must appear in the displayed order.
- INSTID is the ID of the instance to stop.
- If the instance is running, IJBLFPOP sends a message to the console that must be confirmed to complete the stop of the instance. You can enter YES to stop the instance, NO to cancel, and LIST to list all tasks and their partitions that currently use this instance.
- You can stop an instance at any time.
- Active tasks that use the instance will get appropriate error codes when they call a function of the stopped instance.
- The skeleton SKLFPSTO in ICCF library 59 can be used to create a job to stop a specific LFP instance.

Following is an example of a successfully stopped instance:

```
LFPB012I  REALLY STOP INSTANCE '01' (3 TASKS ACTIVE)? (YES/NO/LIST)
0 list
LFPB028I  ACTIVE TASK IDS FOR INSTANCE '01':
          2E (Z1)    2F (Z2)    30 (Z3)
LFPB029I  END OF LIST OF ACTIVE TASK IDS.
LFPB012I  REALLY STOP INSTANCE '01' (3 TASKS ACTIVE)?(YES/NO/LIST)
0 yes
LFPB020I  STOPPED LFP INSTANCE '01'.
```

Linux on System z

You can start and stop LFP daemons by invoking the `lfpds` directly, or by using the control script `lfpd-ctl`. Usually only the control script is used, because it provides a comfortable way to control `lfpds`. However, for tasks like debugging, you have to start `lfpd` directly without using the control script. The following sections describe how to:

- Control an `lfpd` using the control script.
- Start an `lfpd` without using the control script.
- Stop an `lfpd` without using the control script.

Controlling an LFP daemon using the control script

The `lfpd` control script `lfpd-ctl` requires that all enabled configuration files respect the naming conventions described in “Configuring Linux Fast Path” on page 397. `Lfpd-ctl` can only start configurations that are linked in the directory `/etc/opt/ibm/vse/lfpd/confs-enabled`.

The invocation parameters for `lfpd-ctl` are:

```
lfpd-ctl (start|stop|restart|force-reload|status|startdbg) <APPNAME>
lfpd-ctl (list|startall|stopall)
```

lfpd-ctl start APPNAME

This command starts an LFP daemon with the specified configuration. The above command, for example, invokes an lfpd with the configuration file `/etc/opt/ibm/vselfpd/confs-enabled/lfpd-APPNAME.conf`. If the start of the lfpd fails, check the Linux system log (usually the file `/var/log/messages`) for any messages related to the startup failure. Alternatively you can use the command `lfpd-ctl startdbg APPNAME`, which is described below.

lfpd-ctl stop APPNAME

This command stops a running LFP daemon that was started with the configuration named APPNAME.

lfpd-ctl restart | force-reload APPNAME

The restart and force-reload parameters stop the lfpd and then start it again. Use them, if you have changed a configuration, it cannot be applied dynamically to a running lfpd.

lfpd-ctl status APPNAME

The status command shows the status of the lfpd. The status can be:

- *not running*,
- *listening* for z/VSE LFP instances to connect,
- *connected to xxx*, where *xxx* is the application name of the connected z/VSE LFP instance.

lfpd-ctl startdbg APPNAME

This command starts an lfpd with the specified configuration. It works similar as `lfpd-ctl start APPNAME`. However, if the lfpd startup fails, the related lfpd messages are shown on the console where the command was issued. There is no need to check the Linux system log for the messages, although the messages are shown there, too.

lfpd-ctl list

The list command generates a table with information about each running or enabled lfpd, together with the individual status of the daemon.

lfpd-ctl startall

The startall command will try to start all enabled configurations that are not already started. This command is also invoked by the SysV init script during startup of the system.

lfpd-ctl stopall

The stopall command will stop all running lfpds, no matter if they are still enabled or not. This command is also invoked by the SysV init script during shutdown of the system, or if the lfpd rpm install package is removed from the system.

Starting an LFP daemon without using the control script

To start the LFP daemon on Linux on System z run the program lfpd. Lfpd is a daemon that can run in the background and serves one LFP instance on a z/VSE system. The invocation parameter format is

```
lfpd <--file|-f configfile> [--debug|-d] [--packets|-p] [--startuplogfile|-s logfile]
[--help|-h]
```

- The parameters are not positional and can be given in any order.

- `configfile` is a mandatory parameter and specifies the absolute path of the configuration file for `lfpd`. For example: `/etc/lfpd/lfpd.cfg`.
- If `--debug` or `-d` is specified, `lfpd` will not run in the background but will write debug messages to the console from where it was invoked. The debug output is only needed by the support personnel to track down problems.
- If `-packets` or `-p` is specified, `lfpd` will not run in the background but will write packet dumps to the console from where it was invoked. All packets transferred from and to the z/VSE system are displayed in a textual format. The packet dumps are usually only needed by the support personnel to track down problems.
- After invoking `lfpd`, check the Linux system log (usually `/var/log/messages`) for any error messages from the `lfpd` or for its startup messages, to ensure that the `lfpd` is started correctly.
- If `--startuplogfile` or `-s logfile` is specified, `lfpd` will additionally log all startup messages to the given logfile.

```
linlfp lfpd[6185]: Beginning startup of lfpd
linlfp lfpd[6185]: Now listening on application name 'LINR02'
linlfp lfpd[6185]: Accepted new connection on LINR02 from VSER05
/TESTV , new socketfd=11
linlfp lfpd[6185]: Configuration matched, accepting connection.
```

Figure 29. Linux System Log Output Example

Stopping an LFP daemon without using the control script

Usually the `lfpd` runs in the background. To terminate an `lfpd`, send a `SIGTERM` signal to the pid (process identifier) of the running `lfpd` that should terminate. The local application name (`HS_SRC_APPNAME` or `IUCV_SRC_APPNAME` in `lfpd` configuration) uniquely identifies a running `lfpd`. There are two ways to determine the pid of an `lfpd` that is to be stopped:

1. Open the system log messages file (usually `/var/log/messages`) and check for a line like:

```
linlfp lfpd[6185]: Now listening on application name 'LINR02'
```

The value 6185 is the pid for the `lfpd` that uses the application name `LINR02`.

2. Alternatively the lock file of a running `lfpd` can be used to find its pid. Each running `lfpd` creates a lock file in the directory `/var/run`. The lock file name for the application name `LINR02` for example, is `lfpd-LINR02.pid`. The content of the lock file is the pid of the `lfpd` process.

Make sure that **no** z/VSE system is currently connected to the `lfpd`, before sending the `SIGTERM` signal to a running `lfpd`. If you terminate the `lfpd`, when a z/VSE system is still connected, the LFP instance on z/VSE will detect this and set its own status to down.

Go to `/var/log/messages` to check if the z/VSE system is disconnected from the running `lfpd`:

```
linlfp lfpd[6185]: All tasks abended, status set to DOWN.
linlfp lfpd[6185]: Data socket closed, ready for new connection on
application name 'LINR02'.
```

Figure 30. Linux System Log Output Example of a Disconnected z/VSE System

If this is the last output from the `lfpd` with the pid 6185, no z/VSE system is currently connected and the termination of `lfpd` will not result in unpredictable

results from applications running on z/VSE.

Administrative Tasks

z/VSE

List active instances

To show a list of currently active instances on the z/VSE system use the LFP operator program IJBLFPOP.

The invocation parameter format is:

```
EXEC IJBLFPOP,PARM='LIST'
```

The output is always written to the console as well as to the job listing.

The skeleton SKLFPLST in ICCF library 59 can be used to create a job that lists all running LFP instances.

```
LFPB025I ACTIVE LFP INSTANCES: 1
          INSTANCE 01 HAS 3 ACTIVE TASKS
LFPB026I END OF ACTIVE LFP INSTANCES LIST.
```

Figure 31. Active Instance List Output Example

Display information about an active instance

To display detailed information about an LFP instance on z/VSE use the LFP operator program IJBLFPOP.

The invocation parameter format is:

```
EXEC IJBLFPOP,PARM='INFO <INSTID> [SHOWTASKS] [LOGALL]'
```

- All parameters are positional and must appear in the displayed order.
- INSTID is the ID of the instance to display information about.
- SHOWTASKS is an optional parameter. Use this parameter to display detailed information about each task that currently uses LFP.
- LOGALL is an optional parameter. If the parameter is specified, IJBLFPOP writes **all** messages to the console. If the parameter is omitted, only error messages are printed to the console. All messages are always printed into the job listing.
- The displayed information shows various values of the running instance. Some of them show the current memory consumption and details about the active tasks and their sockets. Other values are only of interest for the support personnel to track down problems.
- The skeleton SKLFPINF in ICCF library 59 can be used to create a job that displays information about a specific LFP instance.

Following is an output example for an LFP via IUCV connection with all parameters set:

Figure 32. LFP Instance on z/VSE via IUCV Connection Output Example

```
LFPB023I INFO ABOUT LFP INSTANCE '01':
*** INSTANCE ***
STATUS ..... : UP
WINDOW SIZE ..... : 65,535
WINDOW THRESHOLD ..... : 25% (16,383 bytes)
```

```

TASKS WAITING FOR MSGLIMIT ..... : 0
TIMES IN WAIT FOR MSGLIMIT ..... : 0
TASKS WAITING FOR PACKET ..... : 0
TIMES IN WAIT FOR PACKET ..... : 0
GETVIS POOL ID ..... : ILFP01
*** DEVICE ***
DEVICE STATUS ..... : ACTIVE
DEVICE TYPE ..... : IUCV
MTU SIZE ..... : 8,192
PACKETS SENT ..... : 88
PACKETS RECEIVED ..... : 53
PACKETS DROPPED ..... : 0
PACKETS WAITING FOR MSG COMPLETE : 0
FREE NOTIFY BUFFERS ..... : 1,024
CONFIGURED LOCAL IUCV MSGLIMIT . : 1,024
ACTUAL LOCAL IUCV MSGLIMIT ..... : 1,024
ACTUAL REMOTE IUCV MSGLIMIT .... : 1,024
IUCV MSGLIMIT EXCEEDED ..... : 0
IUCV SOURCE APPL. NAME ..... : TESTV
IUCV DEST. APPL. NAME ..... : LFP61
IUCV DEST. VM USERID ..... : LINLFP
IUCV DEST. VM SYSTEM ..... : (LOCAL)
*** BUFFER MANAGER ***
CURRENTLY USED MEMORY ..... : 524,160
INITIAL MEMORY SIZE ..... : 524,288
MAXIMUM MEMORY SIZE ..... : 4,194,304
-- POOL #1 --
PACKET SIZE ..... : 64
INITIAL POOL SIZE ..... : 181,568
CURRENT POOL SIZE ..... : 181,440
POOL COUNT ..... : 1
AVAILABLE PACKET COUNT ..... : 266
OVERALL PACKET COUNT ..... : 266
MAXIMUM PACKETS USED ..... : 2
-- POOL #2 --
PACKET SIZE ..... : 3,552
INITIAL POOL SIZE ..... : 178,864
CURRENT POOL SIZE ..... : 176,160
POOL COUNT ..... : 1
AVAILABLE PACKET COUNT ..... : 43
OVERALL PACKET COUNT ..... : 43
MAXIMUM PACKETS USED ..... : 2
-- POOL #3 --
PACKET SIZE ..... : 8,192
INITIAL POOL SIZE ..... : 174,762
CURRENT POOL SIZE ..... : 166,560
POOL COUNT ..... : 1
AVAILABLE PACKET COUNT ..... : 18
OVERALL PACKET COUNT ..... : 19
MAXIMUM PACKETS USED ..... : 6
*** DEBUG ***
DEBUG IS ENABLED
DEBUG AREA SIZE ..... : 320,000
DEBUG ENTRY COUNT ..... : 10,000
*** DIAGNOSIS ***
DIAGNOSIS IS DISABLED
*** TASKS ***
ACTIVE TASK COUNT ..... : 1
-- TASK #1 --
TASK ID (PARTITION ID) ..... : 2D (R1)
SOCKET COUNT ..... : 2
L2 SOCKET LIST COUNT ..... : 1
PENDING REQUESTS ..... : 0
PENDING SOCKET REQUESTS ..... : 0
REQUESTS WAITING FOR MSGLIMIT .. : 0
TIMES IN WAIT FOR MSGLIMIT ..... : 0
PACKETS WAITING FOR RECEIVE .... : 0

```

```

TIMES IN WAIT FOR PACKET ..... : 0
FREE REQUEST COUNT ..... : 2
LFPB024I  END OF INFO ABOUT LFP INSTANCE '01'.

```

Following is an output example for an LFP via HiperSockets connection with all parameters set:

Figure 33. LFP Instance on z/VSE via HiperSockets Connection Output Example

```

LFPB023I  INFO ABOUT LFP INSTANCE '99':
*** INSTANCE ***
STATUS ..... : UP
WINDOW SIZE ..... : 65,535
WINDOW THRESHOLD ..... : 25% (16,383 bytes)
PACKET CONSOLIDATION THRESHOLD . : 100
TASKS WAITING FOR CQ OR WINDOW . : 0
TIMES IN WAIT FOR CQ OR WINDOW . : 0
TASKS WAITING FOR PACKET ..... : 0
TIMES IN WAIT FOR PACKET ..... : 0
GETVIS POOL ID ..... : ILFP99
*** DEVICE ***
DEVICE STATUS ..... : ACTIVE
DEVICE TYPE ..... : HS
MTU SIZE ..... : 8,192
PACKETS SENT ..... : 4
PACKETS RECEIVED ..... : 3
PACKETS DROPPED ..... : 0
HS COMPLETION QUEUE STATES ..... : 0
PROTOCOL WINDOW FULL STATES .... : 0
QUEUE LENGTH ..... : 16
PROTOCOL WINDOW LENGTH ..... : 10
KEEP-ALIVE TIME SECONDS ..... : 999,999
HS DEVICES ..... : E00,E01,E02
HS SOURCE APPL. NAME ..... : TESTV2
HS SOURCE SYSTEM NAME ..... : VSELP61
HS DEST. APPL. NAME ..... : LFP612
HS DEST. SYSTEM NAME ..... : LINHSIO
*** BUFFER MANAGER ***
.
.   ...same output as for IUCV ...
.
LFPB024I  END OF INFO ABOUT LFP INSTANCE '99'

```

Enable or disable socket diagnosis

To display detailed information about an LFP instance on z/VSE use the LFP operator program IJBLFPOP.

The invocation parameter format is:

```
EXEC IJBLFPOP,PARM='DIAG <INSTID> <ON|OFF>'
```

- All parameters are positional and must appear in the displayed order.
- INSTID is the ID of the instance of which you want to change the socket diagnosis setting.
- Specify ON to enable socket diagnosis, or OFF to disable it. If socket diagnosis is ON for an LFP instance, socket statistics are printed to the job listing each time a socket is closed by the application that is using the LE/C socket API (\$EDCTCPV).

Following is an output example of a socket diagnosis:

Figure 34. Socket Diagnosis Output Example

```
LFP Statistics for Socket '0':
  Number of Bytes Sent:      1020364776
  Number of Bytes Received:  943718400
  Number of Packets Sent:    233923
  Number of Packets Received: 183220
  Times in Wait for Send Window: 18158
  Times in Wait for Receive Window: 28002
  Times in Wait for Message Limit: 0
  Times in Wait for Packets: 0
```

Linux on System z

Display LFP daemon status

To display the status of a running LFP daemon use the program `lfpd-admin`.

The invocation parameter format is:

```
lfpd-admin <--appname|-appname> <--status|-s>
```

- The parameters are not positional and can be given in any order.
- For `appname` enter the `HS_SRC_APPNAME` or `IUCV_SRC_APPNAME` that is used by a running `lfpd`. `HS_SRC_APPNAME` and `IUCV_SRC_APPNAME` are parameters of the `lfpd` configuration.
- The status command display various status information on the console where `lfpd-admin` was invoked.
- The `lfpd-admin` program can also be used to start a debug trace to a file. This functionality is usually only used by the support personnel who will send you instructions about the parameters when needed.

Following is an output example of `lfpd-admin` via IUCV:

Figure 35. `lfpd-admin` via IUCV Connection Output Example

```
linlfp:~ # lfpd-admin -a lfp61 -s
Configuration:
-----
DEVICETYPE ..... = IUCV
LOCAL_IUCV_APPNAME = LFP61
PEER_IUCV_VMID ... = VSELP61
IUCV_MSGLIMIT .... = 1024
IUCV_MTU_SIZE .... = 8192
MAX_VSE_TASKS .... = 512
MAX_SOCKETS ..... = 1024
INITIAL_IO_BUFS .. = 128
WINDOW_SIZE ..... = 65535
WINDOW_THRESHOLD . = 25% (16383 bytes)
LINUX_CODEPAGE ... = '850'
VSE_CODEPAGE ..... = 'EBCDIC-US'
SUPPORT_GETXXXENT = yes
VSE_HOSTID ..... = 9.152.84.210
RESTRICT_TO_HOSTID = no
LOG_INFO_MSG ..... = no

Status:
-----
z/VSE instance is connected.
Peer system name ... : VSELP61
Peer appl. name .... : TESTV
Actual IUCV MSGLIMIT : 1024
```



```

Applied host id .... : 9.152.84.210
Applied host name .. : linlfp

Allocated I/O buffers ..... : 128
Free I/O buffers ..... : 128
Allocated request buffers ... : 128
Free request buffers ..... : 126
Allocated socket buffers .... : 128
Free socket buffers ..... : 125
Buffers waiting for send .... : 0

Number of active z/VSE tasks : 1
Number of active sockets .... : 2

```

Trace Status:

```

-----
Running in daemon mode
No trace is running

```

Following is an output example of lfpd-admin via HS CQ:

Figure 36. lfpd-admin via HS CQ Connection Output Example

```

[root@linhsi0 ~]# lfpd-admin -a lfp612 -s
Configuration:

```

```

-----
DEVICETYPE ..... = HS
HS_SRC_APPNAME ... = LFP612
HS_SRC_SYSTEMNAME = LINHSI0
HS_MSGLIMIT ..... = 128
MAX_VSE_TASKS .... = 512
MAX_SOCKETS ..... = 1024
INITIAL_IO_BUFS .. = 128
WINDOW_SIZE ..... = 65535
WINDOW_THRESHOLD . = 25% (16383 bytes)
LINUX_CODEPAGE ... = '850'
VSE_CODEPAGE ..... = 'EBCDIC-US'
SUPPORT_GETXXXENT = yes
VSE_HOSTID ..... = 9.152.131.42
RESTRICT_TO_HOSTID = no
LOG_INFO_MSG ..... = no

```

Status:

```

-----
z/VSE instance is connected.
Peer system name ... : VSELP61
Peer appl. name .... : TESTV2

Applied host id .... : 9.152.131.42
Applied host name .. : linhsi0

MTU size ..... : 8088
Allocated I/O buffers ..... : 128
Free I/O buffers ..... : 128
Allocated request buffers ... : 128
Free request buffers ..... : 128
Allocated socket buffers .... : 128
Free socket buffers ..... : 127
Buffers waiting for send .... : 0

Number of active z/VSE tasks : 0
Number of active sockets .... : 0

```

Trace Status:

Running in daemon mode
No trace is running

Chapter 14. z/VSE - z/VM IP Assist

Overview

The z/VSE - z/VM IP Assist function, simply referred to as z/VSE VIA, provides the z/VM counterpart to the Linux Fast Path on z/VSE. The “traditional” z/VSE Linux Fast Path function requires the user to install, administrate and configure a Linux on System z system, in order to run the Linux Fast Path daemon (lfpd). On this Linux on System z system, the lfpd has to be installed and configured. For customers who prefer not to install and maintain a Linux on System z system on their own, the z/VSE VIA function provides an easy to use and ready to run z/VM guest image that provides all services required to use the z/VSE Linux Fast Path.

Minimum requirements

- IBM zEnterprise system (z196 or z114) driver level 86 or later
- z/VM 6.1 or later
- z/VSE 5.1

Communication Flow when Using z/VSE VIA

Figure 37 shows a communication flow using z/VSE VIA over an IUCV connection.

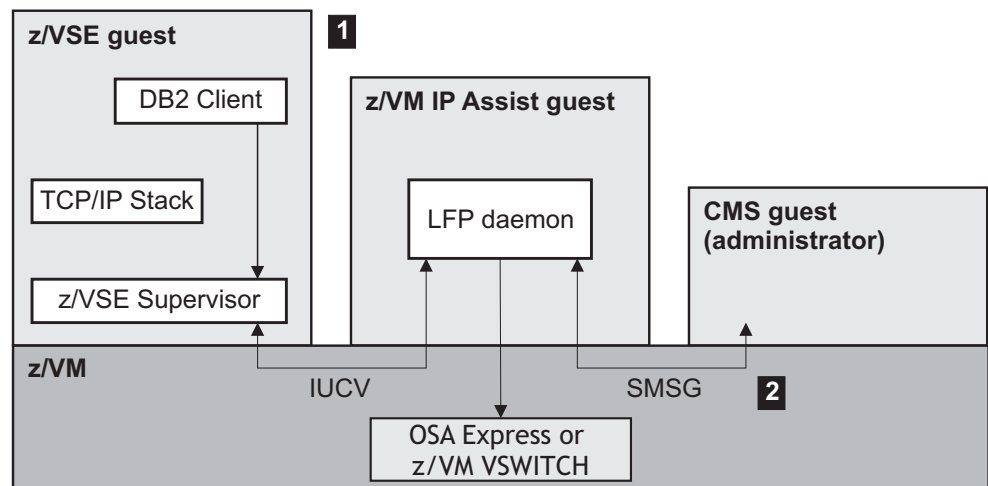


Figure 37. Communication using z/VSE VIA and an IUCV connection

The processing shown in Figure 37 is as follows:

- 1** The communication via IUCV is the same as described in Figure 23 on page 391
- 2** The administrator interacts with the z/VSE VIA guest by sending SMSG messages from a CMS guest to the z/VSE VIA guest.

Note: Refer to the z/VM Documentation for details on the SMSG command and the VSWITCH controller.

z/VSE VIA z/VM guest configuration

The z/VSE VIA guest image must be configured using the SCPDATA parameter of the LOADDEV directory control statement of a z/VM directory entry. The SCPDATA parameter specifies the network configuration that is used for the z/VSE VIA guest. The information is formatted in JSON (JavaScript Object Notation, for details refer to www.json.org). SCPDATA must be specified using EBCDIC code page 924. Usually the z/VSE VIA guest is configured to start up automatically when z/VM starts.

The z/VSE VIA guest is configured to have access to 2 CMS minidisks:

Configuration Disk

The minidisk contains the lpfed configuration files (one for each lpfed instance) and the SENDERS.ALLOWED file. It must be linked as 0D4C in read-only mode. The configuration disk must be linked during startup by the z/VM Directory entry. The configuration using the files on the configuration disk is usually performed through an administrator CMS guest. This CMS guest should therefore have access the configuration disk in read-write mode.

Data Disk

Trace files are written to this minidisk when the user starts a trace. This minidisk is optional and only required, if the user wants to start a trace. It must be linked as 0D4D in read-write mode. The data disk can be linked during startup, or if a trace needs to be started by using the CP command:

```
CP LINK ZVSEVIA 4321 0D4D MR
```

The SCPDATA parameter supports the following elements:

profiles

Specifies the profile name. You must specify “zVSE-VIA” This entry is required.

hostName

Must be specified as a string containing the hostname to use. If no hostname is specified, the name of the first profile will be used. This entry is optional.

networkCards

Defines the array of 0-n network cards to use in the guest. Each of the array elements is a JSON object, which includes two elements:

- **card type:** OSA, OSM, OSX, HiperSockets followed by the respective device address or all. If all is specified, no other device of the same type can be defined.
- **IP configuration:** linkLocalIPv6, staticIPv4, staticIPv6 followed by the IP address or null, if no further configuration is required.

Defining an IPv4 and an IPv6 configuration for the same card is valid. This entry is optional.

defaultGateway

String with the default gateway to be used. This entry is optional.

DNS

Array of strings containing IP addresses of DNS servers. This entry is optional.

Note: Refer to the z/VM Documentation for details on the LOADDEV SCPDATA control statement. Be aware that the naming convention in z/VM is “SCPDATA operand” not “SCPDATA parameter” as in z/VSE.

Example

The following is an example user directory entry for a z/VSE VIA guest:

```
USER ZVSEVIA AUTOONLY 1G 1G G
COMMAND SET D8ONECMD * OFF
COMMAND SET RUN ON
COMMAND TERM LINEND #
COMMAND SPOOL CONS START *
IPL 0 1
LOADDEV PORT 0
LOADDEV LUN 0
LOADDEV BOOT 0
LOADDEV BR_LBA 601
* Network adapters and configuration
LOADDEV SCPDATA '{',
  "profiles":["zVSE-VIA"],',
  "networkCards": ['],
  { "OSM": "all", "linkLocalIPv6": null},',
  { "OSA": "2408", "staticIPv4": "9.152.11.86/24"},',
  { "OSX": "110", "staticIPv6": "2001:0db8:85a3::7334/64"},',
  { "hipersockets": "9000", "linkLocalIPv6": null},',
],',
  "defaultGateway":"y.y.y.y/nn",',
  "DNS":["y.y.y.y/nn","z.z.z.z/nn"],',
  "hostName":"myServer"',
}'
* machine type and number of CPUs
MACH XA 1
OPTION LXAPP LANG AMENG
OPTION MAXCONN 128
* IUCV authorizations
IUCV ANY PRIORITY MSGLIMIT 1024
IUCV ALLOW
* Standard virtual devices
CONSOLE 009 3215 T
SPOOL 000C 2540 READER *
SPOOL 000D 2540 PUNCH A
SPOOL 000E 1403 A
* Network definitions (use either DEDICATE or NIC+VSWITCH)
DEDICATE dddd vvvv
* COMMAND DEFINE NIC vvvv TYPE QDIO
* COMMAND COUPLE vvvv TO SYSTEM vswitch
* Minidisks
LINK MAINT 0190 0190 RR
LINK MAINT 019D 019D RR
LINK MAINT 019E 019E RR
* 191 minidisk is optional
* MDISK 191 3390 XXXX 007 AUTOV MR XXXXXXXX XXXXXXXX XXXXXXXX
* Disk for configuration (D4C) and log file (D4D)
MDISK D4C 3390 XXXX 009 AUTOV RR XXXXXXXX XXXXXXXX XXXXXXXX
MDISK D4D 3390 XXXX 071 AUTOV MR XXXXXXXX XXXXXXXX XXXXXXXX
```

z/VSE VIA Linux Fast Path Configuration

Besides the z/VM guest that runs the z/VSE VIA image, you also have to configure one or more lfpds that the z/VSE VIA guest should run. The lfpds are configured with files that reside on the configuration minidisk (0D4C) and SMSGs that the user sends to the z/VM guest where z/VSE VIA runs.

The administrator can interact with the z/VSE VIA guest by sending SMSG messages from a CMS guest to the z/VSE VIA guest. To prevent unauthorized SMSG traffic each sender is validated against a list of authorized users contained in the CMS file "SENDERS.ALLOWED" residing on the configuration disk (0D4C).

This file contains one single z/VM user ID per line. All specified IDs are authorized to send SMSG commands to the z/VSE VIA guest.

The configuration files for the lfpds must also reside on the configuration disk (0D4C). The configuration files have the same content as described on page “Linux on System z” on page 408.

Each configuration file must have the same file name as the IUCV_SRC_APP that it configures, and must have the CMS file type “LFPDCONF”. All configurations that exist on the configuration disk during the start of the z/VSE VIA guest are started automatically. When the startup of the z/VSE VIA guest has completed, the user can administrate the configurations using the supported SMSG commands.

Sample content of a configuration disk:

```
CONFIG-1 LFPDCONF
CONFIG-2 LFPDCONF
SENDERS  ALLOWED
```

z/VSE VIA Linux Fast Path Administration

The administration of the lfpd is usually done using the LFP control script lfpd-ctl and the lfpd-admin program. For the z/VSE VIA function, SMSGs are used to administrate the lfpds. SMSG is used to send textual messages from one z/VM guest to another z/VM guest. The z/VSE VIA function implements a user verification system that allows only authorized users to access the lfpd administrative interface.

The general syntax to send an SMSG to the z/VSE VIA administrative interface is:
SMSG <ZVSEVIA> APP <CMD PARAMS>

The administrative commands return their output by sending it line by line back to the issuer of the SMSG command.

Figure 38. z/VSE VIA Administrative Command Output Example

```
smsg vsevia app lfpd
Ready; T=0.01/0.01 15:39:24
15:34:24 * MSG FROM VSEVIA : USAGE: SMSG <GUEST> APP LFPD (START|STOP|RESTART|FORCE-RE
LOAD|STATUS|STARTDBG) <IUCVNAME>
15:34:24 * MSG FROM VSEVIA :          SMSG <GUEST> APP LFPD (LIST|STARTALL|STOPALL)

smsg vsevia app lfpd-admin
Ready; T=0.01/0.01 15:39:59
15:34:59 * MSG FROM VSEVIA : USAGE: SMSG <GUEST> APP LFPD-ADMIN <IUCVNAME> TRACE START
<FILENAME> (DEBUG|PACKETS|ALL) (SINGLE|WRAP) (MAXSIZE)
15:34:59 * MSG FROM VSEVIA :          SMSG <GUEST> APP LFPD-ADMIN <IUCVNAME> TRACE STOP
15:34:59 * MSG FROM VSEVIA :          SMSG <GUEST> APP LFPD-ADMIN <IUCVNAME> STATUS
```

The administrative commands are:

- **LFPD**, which provides the functionality of the lfpd-ctl script.
- **LFPD-ADMIN**, which provides the functionality of the lfpd-admin script.

LFPD Command Format

The LFPD command has the following syntax:

```
smsg <GUEST> app lfpd (start|stop|restart|force-reload|status|startdbg) <IUCVNAME>
smsg <GUEST> app lfpd (list|startall|stopall)
```

The meaning of the functions is the same as described on page “Linux on System z” on page 408. The startdbg parameter can be used to get diagnostic information when starting an lfpd instance.

Example

The output of the startdbg command displays all lfpd startup messages.

Figure 39. z/VSE VIA startdbg Command Output Example

```
smsg vsevia app lfpd start fails
Ready; T=0.01/0.01 15:48:16
 15:48:19 * MSG FROM VSEVIA : STARTING LFPD (FAILS): FAILED

smsg vsevia app lfpd startdbg fails
Ready; T=0.01/0.01 15:48:35
 15:48:38 * MSG FROM VSEVIA : STARTING LFPD (FAILS): FAILED
 15:48:38 * MSG FROM VSEVIA : STARTUP LOG
 15:48:38 * MSG FROM VSEVIA : LOGGING STARTUP MESSAGES TO FILE '/TMP/LFPD-CTL.2854.tmp'.
 15:48:38 * MSG FROM VSEVIA : READING CONFIGURATION FROM FILE '/ETC/OPT/IBM/VSELFPD/CONFS-
ENABLED/LFPD-FAILS.CONF'.
 15:48:38 * MSG FROM VSEVIA : IUCV_INITIALIZESOCKET: ERROR: THE BIND CALL FOR THE AFIUCV
SOCKET FAILED: ADDRESS ALREADY IN USE
 15:48:38 * MSG FROM VSEVIA : ERROR: COULD'NT INITIALIZE THE SOCKETS.
 15:48:38 * MSG FROM VSEVIA : STOPPING TO LOG THE STARTUP BECAUSE LFPD IS ABOUT TO EXIT.
```

LFPD-ADMIN Command

Format

The LFPD-ADMIN command has the following syntax:

```
smsg <GUEST> app lfpd-admin <IUCVNAME> trace start <FILENAME> (debug|packets|all)
(single|wrap) (maxsize)
smsg <GUEST> app lfpd-admin <IUCVNAME> trace stop
smsg <GUEST> app lfpd-admin <IUCVNAME> status
```

The “trace start” function starts the lfpd trace to a specified file. This trace file is created on the CMS mini disk 0D4D. The FILENAME must be a valid CMS file name without the file type. The meaning of the functions is the same as described on page “Linux on System z” on page 408. All parameters are optional and can be specified in any order. Before a trace can be started, the user **must** attach a disk with the CMS file format to the device number 0D4D. The disk can be detached after the trace is stopped. The textual lfpd trace is written in **binary** mode to the CMS disk. The trace file is not readable on z/VM.

Note: Only one trace can be started at one time.

Chapter 15. OpenSSL

This section describes how OpenSSL is implemented in z/VSE.

Overview

OpenSSL is an open source project that provides an SSL implementation and key management utilities. OpenSSL is written in C and is available for many operating systems and hardware platforms. OpenSSL on z/VSE provides SSL for the IPv6/VSE IP stack, which does not have an own SSL implementation. For details on OpenSSL, refer to <http://www.openssl.org/>

On z/VSE 5.2 OpenSSL is provided as part of a new system component:

VSE CryptoServices
5686-CF9-17
CLC=52S

OpenSSL is installed in PRD1.BASE and includes:

- IJBSSL phase (the OpenSSL implementation)
- SPEEDTST phase (invokes the built-in OpenSSL speed test)
- NOTICES.Z (license information)
- IJBVLVSE.OBJ (provides access to the APIs and must be linked to your application)
- IJBSSL.H (provides function prototypes)

Note: The current OpenSSL version on z/VSE 5.2 is based on OpenSSL 1.0.0e.

Unique features on z/VSE

A z/OS compatible SSL programming interface

This API is described in z/OS Cryptographic Services System SSL Programming, and is used by all existing SSL applications on z/VSE, such as CICS Web Support, VSE Connector Server, and WebSphere® MQ for z/VSE. Wrapping the native OpenSSL functions by this z/OS SSL API allows existing z/VSE SSL applications to run unchanged with OpenSSL.

Support for IBM System z cryptographic hardware

Although OpenSSL can perform all encryption algorithms with all key lengths in software, performance is dramatically improved by using hardware crypto support. Additionally, hardware functionality that is not available in software can be used, such as hardware-based generation of random numbers.

Features not available on z/VSE

In z/VSE 5.2 the following functionality is not available.

- The OpenSSL command-line tool is not available on z/VSE. Therefore, key management is done on a workstation (Windows, Linux, etc.). Created keystores are then uploaded to z/VSE.
- The following algorithms are not available on z/VSE: IDEA, RC5, MDC2.
- Non-LE/C applications are currently not supported. Because OpenSSL is coded in C, an LE/C-runtime environment is required for callers.

Runtime Variables

There are two variables for controlling the behavior of OpenSSL on z/VSE.

The use of crypto hardware can be turned on and off via parameter SSL\$IICA.

```
// SETPARM SSL$IICA = [ 'YES' | 'NO' ]
```

The debug trace can be controlled via variable SSL\$DBG.

```
// SETPARM SSL$DBG = [ 'YES' | 'NO' ]
```

The following sections contain these main topics:

- “Key store considerations”
- “Programming z/VSE Applications for OpenSSL” on page 427
- “Performing the OpenSSL speed test” on page 435

Key store considerations

A key store, which contains the RSA public/private key pair and the SSL certificates, is needed before any SSL application can be run.

Many different types of key stores exist. Key stores that are relevant for z/VSE are:

- PFX** The Personal Information Exchange (PFX) format was initially defined by RSA Security and conforms to the PKCS#12 standard. PFX files can contain multiple keys and certificates and are themselves password-protected. PFX files are supported by Web Browsers like Microsoft Internet Explorer and Mozilla Firefox. The file extension p12 is also used for the PFX format.
- JKS** The Java Key Store (JKS) format is provided by Oracle and is the standard key store format for Java applications. JKS files are password protected. JKS files usually cannot be handled by web browsers. The `keytool.exe` is part of every Java installation, it can also be used for maintaining JKS key stores.

VSE keyring members

This type of key store was invented by Connectivity Systems International and consists of three VSE library members with the same member name, but different member types:

- PRVK (contains the RSA key pair)
- ROOT (contains the SSL root certificate)
- CERT (contains the SSL server certificate)

Whereas other formats store all keys and certificates in one file, the CSI keyring format uses one separate VSE library member for each item.

The z/VSE specific public key infrastructure (PKI), for example the creation of RSA key pairs and SSL certificates, can be managed with Keyman/VSE.

- PEM** The Privacy-enhanced mail (PEM) format is used by OpenSSL. PEM files can contain an RSA key pair, an SSL certificate or both. It can but must not be password protected. Whereas other keystore formats are just binary, the PEM file content is base-64 encoded.

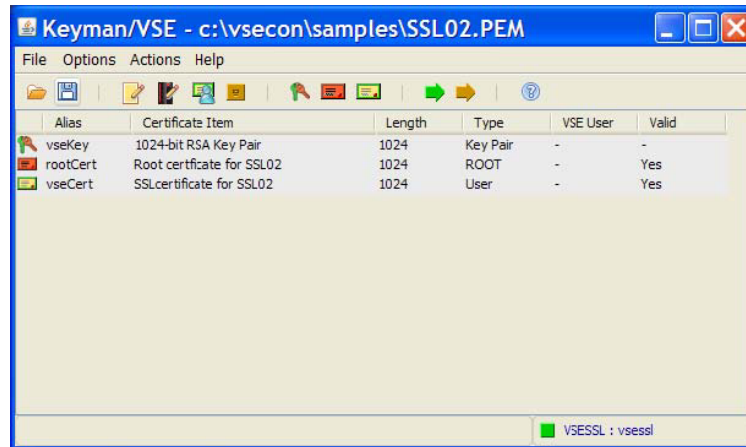
Creating the Key Store Using Keyman/VSE

With the latest version of Keyman/VSE (build date January 2013 or later) you can easily create a PEM file and upload it to z/VSE.

In a test environment it might be desirable to use self-signed certificates, in a production environment you should use certificates issued by a certificate authority (CA).

The following steps describe how to create and upload your PEM file using Keyman/VSE.

1. Create RSA key, SSL and SSL root certificate.

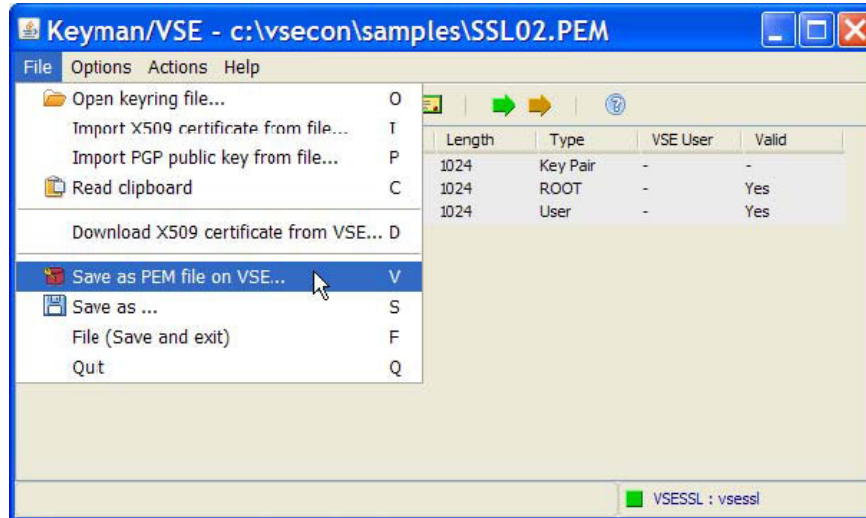


2. Open "Local Properties" and specify PEM file name.

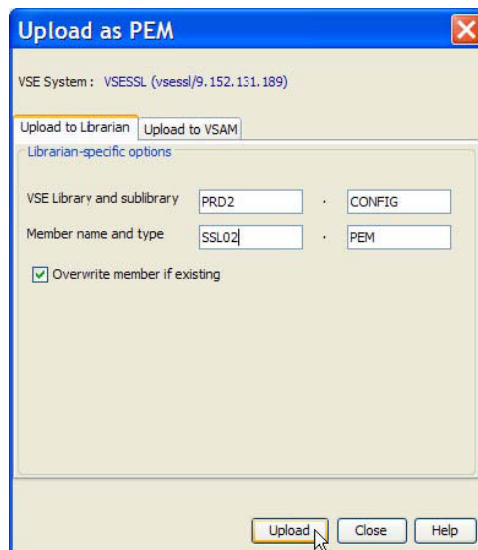


Specifying a PEM password is currently not supported, because IPv6/VSE currently does not support password protected PEM files.

3. Save your PEM file and upload it to z/VSE.



The Upload as PEM popup window is now displayed.



4. Your OpenSSL application is now being configured.
IPv6/VSE is currently the only OpenSSL application on z/VSE. The next steps give an overview on how to configure IPv6/VSE to use OpenSSL. For details refer to the IPv6/VSE SSL Installation, Programming and User's Guide.
5. Configure BSTTATLS
KEYRING PRD2.CONFIG
*
KEYFILE SSL02
SECTYPE SSL30
OPTION SERVER
ATTLS 23 AS 992 SSL
6. Start BSTTATLS and the Telnet server BSTTVNET provided by IPv6/VSE.
7. Setup client if necessary.
8. Connect

```
VSESSL_SSL02 - wc3270
File Options Keypad
TCP/IP-TOOLS TN3270-E Server Version V 253 9.152.131.189 01/17/2
Application Selection Menu 127.0.0.1 15:04
*** *** WELCOME TO BARNARD SOFTWARE, INC. *** ***
Name Status Description
BSTTVNET V-Inactive PRINTER SHARING APPLICATION
DBDCICIS V-Ready CICS/TS AND ICCF
Copyright (c) 1998-2009 by Barnard Software, Inc. VTAM LU Name=T010
```

Programming z/VSE Applications for OpenSSL

You can write z/VSE applications using OpenSSL in two ways.

Using the native OpenSSL API

The OpenSSL API is described on the OpenSSL website and consists of several hundreds of functions. On z/VSE, a subset of this API is provided by IJBSLVSE.OBJ that you link to your application to get access to the functions in the IJBSSL phase.

Using the z/OS SSL API

This is the recommended way of using OpenSSL on z/VSE. The API is described in z/OS Cryptographic Services System SSL Programming. The C data structures are defined in include file gskssl.h, which is part of LE/C and shipped in PRD2.SCEEBASE. The object file IJBSLVSE must also be linked to your application, when using the z/OS SSL API.

Although this API is now deprecated on z/OS and should no more be used by new z/OS applications, it is still the actual SSL API for z/VSE. All existing VSE SSL applications currently use this API, because TCP/IP for VSE/ESA provides exactly this API, and can use OpenSSL unchanged. Refer to “z/OS SSL API” on page 436 for an API description of the GSK functions.

The following sections provide details on using the z/OS SSL API and switching between the two APIs.

Include Files

There are two SSL-related include files provided with z/VSE.

- **sslvse.h** - shipped with the CSI SSL-implementation in the TCP/IP lib.sublib.
- **gskssl.h** - shipped with the LE/C socket interface in PRD2.SCEEBASE.

OpenSSL, as well as OpenSSL applications must be compiled against gskssl.h, because function prototypes in sslvse.h are defined with #pragma linkage OS. This causes errors in passing parameters to the gsk functions.

The gsk API functions use return codes described in the z/OS API, if gskssl.h is used. These return codes are different from the ones currently used by the CSI SSL implementation. The VSE Connector Server for example, is compiled against gskssl.h, but there might be customer applications that use the CSI include file.

Passed Socket Number

The socket number as passed by a gsk-application (int s) is not necessarily just the socket number.

VSE applications like CWS, VSE Connector Server, and MQ, pass a pointer to a private structure, which contains the socket somewhere in the structure.

```
typedef struct _mysock {
    int somefield;
    char* someptr;
    int socketno;
} MYSOCK;

#pragma linkage (skread,OS)
int skread(int s, void * data, int len)
{
    int rc;
    MYSOCK* mys = (MYSOCK*)s;
    rc = recv(mys->socketno, data, len, 0);
    return(rc);
}
```

Note: The #pragma statements are necessary for correct parameter passing only when the application phase is a separate phase. If you want to link an application together with the OpenSSL code (OpenSSL OBJs), don't use the #pragma statements.

The application then specifies the function pointers of skread and skwrite in the sk_soc_init_data structure:

```
typedef struct _gsk_soc_init_data { /* Secure soc init data */
    int fd; /* file descriptor */
    gsk_handshake hs_type; /* client or server handshake */
    char * DName; /* keyring entry Distinguished */
    /* name. When NULL the default */
    /* keyring entry is used */
    char * sec_type; /* Type of application */
    /* 0 CLIENT */
    /* 1 SERVER */
    /* 2 SERVER_WITH_CLIENT_AUTH */
    /* 3 CLIENT_NO_AUTH */
    char * cipher_specs; /* SSLV2 not used by VSE */
    char * v3cipher_specs; /* SSLV3 cipher suites */
    int (* skread) /* User supplied READ routine */
    (int fd, void * buffer, int num_bytes);
    int (* skwrite) /* User supplied WRITE routine */
    (int fd, void * buffer, int num_bytes);
    unsigned char cipherSelected[3]; /* SSLV2 not used by VSE */
    unsigned char v3cipherSelected[2]; /* Cipher Spec used */
    int failureReasonCode; /* failure reason code */
    gsk_cert_info * cert_info; /* This information is read from*/
    /* from the client certificate */
    /* when client authentication is*/
    /* enabled */
    gsk_init_data * gsk_data; /* Pointer to init data */
} gsk_soc_init_data;
```

If you specify the function pointers, fetchep must be used.

```
typedef void (*FETCH_PTR)(int);
init_data.skread = (int (*)(int,void*,int))fetchep((FETCH_PTR)skread);
init_data.skwrite = (int (*)(int,void*,int))fetchep((FETCH_PTR)skwrite);
```

The reason is, that skread and skwrite functions are implemented by the application, but are called from phase IJBSSL.

Callback routines

When using the GSK API, the caller specifies a read and a write routine to be used for sending data over the socket.

This means that the SSL component never accesses the socket directly. It also means that the GSK API has no way of using other socket calls, for example, givesocket, takesocket, ctrl. Unfortunately, OpenSSL does socket calls. The low-level socket functions are defined in eos.h where for VSE these #defines are used:

```
#define readsocket(s,b,n) read((s),(b),(n))
#define writesocket(s,b,n) write((s),(b),(n))
```

This causes OpenSSL to use the application specified read/write routines (skread/skwrite). We currently don't have a solution or other socket calls, because the GSK API only considers these two routines. We also have to keep in mind that the socket number as passed by the application (int s) is not necessarily just the socket number. VSE applications like CWS, VSE Connector Server, MQ, pass a pointer to a private struct which contains the socket somewhere in the struct. In order to be flexible, you would write your code as shown in the following example and put the socket number into a data structure.

```
typedef struct _mysock {
    int somefield;
    char* someptr;
    int socketno;
} MYSOCK;

/* I/O routine to perform a read function for SSL VSE */
#pragma linkage (skread,OS)
int skread(int fd, void * data, int len)
{
    int rc;
    MYSOCK* mys = (MYSOCK*)fd;
    rc = recv(mys->socketno, data, len, 0);
    return(rc);
}

/* I/O routine to perform a write function for SSL VSE */
#pragma linkage (skwrite,OS)
int skwrite (int fd, void * data, int len)
{
    int rc;
    MYSOCK* mys = (MYSOCK*)fd;
    rc = send(mys->socketno, data, len, 0);
    return(rc);
}
```

The addresses of the two callback routines skread() and skwrite() are specified in the gsk_soc_init_data structure before calling gsk_secure_soc_init().

Socket Calls

When using the gsk-API, the caller specifies a read and a write callback routine to be used for sending data over the socket.

The SSL component never accesses the socket directly and the gsk-API has no way of using other socket calls, for example givesocket, takesocket, ctrl.

OpenSSL, however, does socket calls. The low-level socket functions are defined in eos.h, where the following #defines are used for VSE:

```
#define get_last_socket_error() errno
#define clear_socket_error() errno=0
#define ioctlsocket(a,b,c) ioctl(a,b,c)
#define closesocket(s) close(s)
#define readsocket(s,b,n) read((s),(b),(n))
#define writesocket(s,b,n) write((s),(b),(n))
```

This causes OpenSSL to use the application specified read/write routines (skread/skwrite). Currently other socket calls are not supported, because the gsk-API only considers these two routines. The gsk-API implementation in turn implements the vse_readsocket and vse_writesocket routines and calls the application specified read/write routines.

```
int (*skread)(int s, void* b, int n);
int (*skwrite)(int s, void* b, int n);

int vse_readsocket(int s, void* b, int n)
{
    return skread(s, b, n);
}

int vse_writesocket(int s, void* b, int n)
{
    return skwrite(s, b, n);
}
```

Switching between gsk and OpenSSL Socket Calls

To allow applications to use the gsk API and the native OpenSSL API dynamically, gsk_initialize() sets a global variable ssl_use_gsk_callbacks in Phase IJBSSL.

This causes the OpenSSL code to use the gsk callbacks. There are only two OpenSSL modules (bssconn.c and bsssock.c) which do socket calls. The following change is made for VSE:

```
if (ssl_use_gsk_callbacks)
    ret=vse_readsocket(b->num,out,out1);
else
    ret=readsocket(b->num,out,out1);
```

The global variable is reset in gsk_uninitialize().

The following sections describe some z/VSE specific aspects, for example how to specify the name and location of your keyring file and the list of SSL ciphers to be used by your application.

Specifying the Key Ring

There are two ways to specify the keyring in a gsk-application.

This depends on whether you uploaded the PEM file as a VSE library member or as a VSAM file. In both cases, the keyring location is specified when calling the gsk_initialize() function and the keyring name is specified when calling the gsk_secure_soc_init() function.

If an empty string for `lib.sublib` is specified, the `gsk-wrapper` expects the keyring label (DName) to be a VSAM file name. If valid values for `lib` and `sublib` are specified, the keyring label is handled as a VSE library member name. The member type must be PEM in this case.

Keyring type Librarian

For Librarian type keyrings a VSE library and sublibrary are specified in the `gsk_initialize()` call.

```
char * keyring = "CRYPTO.KEYRING";
gsk_init_data init_data;
...
init_data.keyring = keyring;
rc = gsk_initialize(&init_data);
```

The member name of the PEM file is specified in the `gsk_secure_soc_init()` call with the DName parameter. The member type must always be PEM.

```
gsk_soc_data * socdata;
gsk_soc_init_data sock_init_data;
...
sock_init_data.DName = "MYKEY"; // VSE library member name
socdata = gsk_secure_soc_init(&sock_init_data);
```

Keyring type VSAM

For VSAM type keyrings, an empty string for parameter “keyring” is specified in the `gsk_initialize()` call.

```
char * keyring = "";
gsk_init_data init_data;
...
init_data.keyring = keyring;
rc = gsk_initialize(&init_data);
```

The keyring name (DName) defined in the `gsk_secure_soc_init()` call, specifies the VSAM file label.

```
gsk_soc_data * socdata;
gsk_soc_init_data sock_init_data;
...
sock_init_data.DName = "MYKEY"; // VSAM file label
socdata = gsk_secure_soc_init(&sock_init_data);
```

Using a Password Protected Keyring

When using password-protected keyrings for PEM files, each client or server must specify the PEM passphrase to access the keystore.

If you are using the `gsk-API`, you specify the keyring password in the `gsk_init_data` structure, when calling `gsk_initialize()`.

```
char * keyring = "";
gsk_init_data init_data;
...
init_data.keyring = keyring;
init_data.keyring_pw = "ssltest";
rc = gsk_initialize(&init_data);
```

Note: The `gsk-API` supports password protected PEM files, however IPv6/VSE currently does not support password protected PEM files

OpenSSL on z/VSE assumes the following:

1. The PEM file was created on an ASCII platform and the password is therefore ASCII-encoded.
2. The gsk application running on z/VSE specifies the password in EBCDIC.

Therefore, before calling the related OpenSSL function for opening the PEM file, the password is translated to ASCII.

Note: Depending on the password characters there might be an issue with the EBCDIC to ASCII translation. It is currently not possible to specify code pages.

Supported Cipher Suites

OpenSSL allows more cipher suites than previously available with TCP/IP for VSE/ESA.

In addition to specifying a list of cipher suites, OpenSSL allows keywords like DEFAULT, ALL, HIGH. However, an application that uses the gsk API must specify a list of hex codes as done previously when using the CSI implementation. The list of hex codes is then internally translated into an OpenSSL readable list of cipher suites.

The following table shows the list of usable cipher suites with OpenSSL and the SSL implementation provided by CSI. The table only lists the cipher suites using the RSA algorithm for the SSL handshaking.

Table 11. OpenSSL Cipher Suites

Hex code	OpenSSL notation	TCP/IP for VSE/ESA notation
01	NULL_MD5	SSL_RSA_WITH_NULL_MD5
02	NULL_SHA	SSL_RSA_WITH_NULL_SHA
03	EXP-RC4-MD5	-
04	RC4-MD5	-
05	RC4-SHA	-
06	EXP-RC2-CBC-MD5	-
08	EXP-DES-CBC-SHA	SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
09	DES-CBC-SHA	SSL_RSA_WITH_DES_CBC_SHA
0A	DES-CBC3-SHA	SSL_RSA_WITH_3DES_EDE_CBC_SHA
11	EXP-EDH-DSS-DES-CBC-SHA	-
12	EDH-DSS-CBC-SHA	-
13	EDH-DSS-DES-CBC3-SHA	-
14	EXP-EDH-RSA-DES-CBC-SHA	-
15	EDH-RSA-DES-CBC-SHA	-
16	EDH-RSA-DES-CBC3-SHA	-
17	EXP-ADH-RC4-MD5	-
18	ADH-RC4-MD5	-
19	EXP-ADH-DES-CBC-SHA	-
1A	ADH-DES-CBC-SHA	-
1B	ADH-DES-CBC3-SHA	-
2F	AES128-SHA	TLS_RSA_WITH_AES_128_CBC_SHA

Table 11. OpenSSL Cipher Suites (continued)

Hex code	OpenSSL notation	TCP/IP for VSE/ESA notation
32	DHE-DSS-AES128-SHA	-
33	DHE-RSA-AES128-SHA	-
34	ADH-AES128-SHA	-
35	AES256-SHA	TLS_RSA_WITH_AES_256_CBC_SHA
38	DHE-DSS-AES256-SHA	-
39	DHE-RSA-AES256-SHA	-
3A	ADH-AES256-SHA	-
62	EXP1024-DES-CBC-SHA	- ⁽¹⁾
63	EXP1024-DHE-DSS-DES-CBCSHA	-
64	EXP1024-RC4-SHA	-
65	EXP1024-DHE-DSS-RC4-SHA	-
66	DHE-DSS-RC4-SHA	-

⁽¹⁾ Cipher suite 0x62 was formerly supported by TCP/IP for VSE/ESA as RSA1024_EXPORT_DESCBC_SHA, but was removed in version 1.5F.

Note: Some cipher suites are supported by z/OS (for example "00" = "NULL-NULL", "0C", "0D", "0F"), but not supported by OpenSSL. For compatibility with z/OS, these cipher suites are also not supported by z/VSE.

For details on OpenSSL cipher suites refer to <http://www.openssl.org/docs/apps/ciphers.html>. For a description of the cipher suites refer to RFCs 2246 and 3268.

Specifying Cipher Suites

The gsk clients and gsk servers specify their lists of cipher suites as required by the gsk API.

```
char * ciphers = "2F350A09";
...
sock_init_data.v3cipher_specs = ciphers;
...
socdata = gsk_secure_soc_init(&sock_init_data);
```

The list is then translated into an OpenSSL readable form. In this example, the resulting string to be passed to OpenSSL is:

```
AES128-SHA:AES256-SHA:DES-CBC3-SHA:DES-CBC-SHA
```

Supported RSA Key Lengths

On z/VSE the RSA key length is limited to 4096 bits, because this is currently the upper limit supported by crypto cards.

OpenSSL itself can process RSA keys up to 16384 bits. For z/VSE the following change in include file `rsa.h` has been made:

```
#ifndef OPENSSL_RSA_MAX_MODULUS_BITS
//# define OPENSSL_RSA_MAX_MODULUS_BITS 16384
# define OPENSSL_RSA_MAX_MODULUS_BITS 4096
#endif
```

Removing this restriction, as well as using software-based encryption, would not make any sense, because software performance is very limited compared to using cryptographic hardware.

Debugging

OpenSSL has different built-in debug capabilities.

When using the gsk API

To turn the debug trace on and off in the z/OS gsk API, you can use the JCL variable:

```
SSL$DBG = [YES | NO]
```

The variable is evaluated in the **gsk_initialize()** function and therefore usable only together with the z/OS gsk interface.

When using both APIs

To turn debugging on and off without recompiling any source modules, z/VSE provides two functions via module IJBSLVSE:

```
extern int debug=0;
void ssl_enable_debug(void);
void ssl_disable_debug(void);
```

Enabling debugging causes a static global variable “debug” to be set to 1. Disabling debugging resets the value to zero. The two functions can be used by any z/VSE gsk or OpenSSL application.

Debug output from phase IJBSSL is printed to SYSLST, if debugging is enabled. Consider enabling your gsk or OpenSSL application for debugging by providing a DEBUG parameter in the PARM string, when calling your application:

```
// EXEC MYAPP,PARM='DEBUG'
```

The application code reads the parameter and calls the related function to enable or disable debugging.

Hardware Crypto Support

Hardware crypto support is provided transparently for existing z/VSE SSL applications, because they use the z/OS gsk API.

Depending on which API your application uses, one of the following possibilities applies.

When using the gsk API

To turn the hardware crypto support on and off in the z/OS gsk API, you can use the JCL variable:

```
SSL$ICA = [YES | NO]
```

The variable is evaluated in the **gsk_initialize()** function and therefore usable only together with the z/OS gsk interface.

When using both APIs

To turn debugging on and off without recompiling any source modules, z/VSE provides two functions via modules IJBSLVSE and IJBSLACC:

```
extern int ssl_use_ibmca=1;
void ssl_enable_ibmca(void);
void ssl_disable_ibmca(void);
```

Enabling hardware crypto support causes a static global variable “ssl_use_ibmca” to be set to 1. Calling **ssl_disable_ibmca()** resets the value to zero. The two functions can be used by any z/VSE gsk or OpenSSL application.

Provide a parameter IBMCA in the PARM string when calling your application:

```
// EXEC MYAPP,PARM='IBMCA'
```

The application code reads the parameter and calls the related function to enable or disable hardware crypto support.

Performing the OpenSSL speed test

OpenSSL provides a built-in speed test that allows checking the speed of your system when performing cryptographic algorithms, like RSA, AES, SHA, or DES.

On z/VSE this test is included in phase IJBSSL and can be invoked with JCL as follows:

```
// EXEC SPEEDTST,PARM='OPENSSL'
```

For a list of supported parameters enter the following command on a workstation with OpenSSL installed.

```
#openssl speed ?
```

For command details refer to <http://www.openssl.org/>.

Test Invocation

To invoke the RSA test on z/VSE specify either:

```
// EXEC SPEEDTST,PARM='OPENSSL RSA'
```

or

```
// EXEC SPEEDTST,PARM='RSA'
```

Further examples invoking other algorithms are:

```
// EXEC SPEEDTST,PARM='AES-128-CBC'
// EXEC SPEEDTST,PARM='SHA1'
// EXEC SPEEDTST,PARM='DES-EDE3 SHA256 AES-128-CBC'
```

The following algorithms are supported by crypto hardware:

- RSA (includes RSA-1024, RSA-2048, RSA-4096)
- DES-CBC
- DES-EDE3
- AES-128-CBC
- AES-192-CBC
- AES-256-CBC
- SHA1

- SHA256

z/OS SSL API

The z/OS SSL API is supported by TCP/IP for VSE/ESA via the \$EDCTCPV phase and by OpenSSL via phase IJBSSL.

This API is described in detail in:

- “TCP/IP Callable Functions — Function Descriptions” on page 85
- z/OS Cryptographic Services System SSL Programming

Following is a list of supported API function calls, and the differences that are caused by the implementation of the OpenSSL to GSK layer.

gsk_free_memory

Releases storage allocated by the SSL runtime.

No change for z/VSE compared to z/OS.

gsk_get_cipher_info

Returns the supported cipher specifications.

Changes for z/VSE:

- The lists of returned cipher suites differs from what is documented in the z/OS book, because some of the ciphers used on z/OS are not supported by OpenSSL (for example: "00"). OpenSSL on z/VSE returns the following strings:

```
"09151203060201" // LOW_SECURITY  
"05043538392F32330A161309151203060201" // HIGH_SECURITY
```

- The version field of the **gsk_sec_level** struct returns the currently supported OpenSSL version, (for example: 100 = 1.0.0).

gsk_get_dn_by_label

Gets the distinguished name for a certificate.

Changes for z/VSE:

- The specified key/cert file must either be a Librarian member with member type PEM or a VSAM file.
- On z/OS, NULL is returned, if the key database cannot be accessed. However, on z/VSE there is not enough information here to access the key store.

gsk_initialize

Initializes the System SSL runtime environment.

Changes for z/VSE:

- Read and evaluate the JCL variables SSL\$DBG and SSL\$ICA.

gsk_secure_soc_close

Closes a secure socket connection.

No change for z/VSE.

gsk_secure_soc_init

Initializes a secure socket connection.

No change for z/VSE.

gsk_secure_soc_read

Reads data using a secure socket connection.

Changes for z/VSE:

- The caller can specify `buflen = 0` to check for pending bytes. If `buflen = 0`, `SSL_pending` is called and **`gsk_secure_soc_read`** returns the return code that was returned by `SSL_pending`.

`gsk_secure_soc_reset`

Resets the session keys for a secure connection.

No change for z/VSE.

`gsk_secure_soc_write`

Writes data using a secure socket connection.

No change for z/VSE.

`gsk_uninitialize`

Terminates the SSL environment.

No change for z/VSE.

`gsk_user_set`

Sets an application callback.

Currently not supported on z/VSE.

Part 5. CICS Listener Support

Chapter 16. Setting Up and Configuring CICS Listener Support

Overview

This section describes the steps required to configure the CICS Listener Support. The error messages are included in *z/VSE Messages and Codes, Volume 1*.

Important:

1. The CICS Listener Support requires starting CICS/TS with SIT parameter SVA=YES.
2. The CICS Listener Support as provided by IBM requires the "task-related-user-exit" program EZATRUE. Refer to "CICS Considerations for the EZA Interfaces" on page 83 for further details on how to start this program.

Before you can start the CICS Listener Support, you need to do the following:

Task	Refer to
Define additional files, programs, maps, and transient data to CICS using RDO.	"CICS — Defining CICS Resources"
Use the configuration macro (EZACICD), to build the CICS Listener Configuration dataset.	"Building the Configuration data set with the Configuration Macro (EZACICD)" on page 446
Use the configuration transaction to customize the Configuration dataset.	"Customizing the Configuration Dataset" on page 451
Note: You can modify the dataset while CICS is running by using EZAC. See "Configuration Transaction (EZAC)" on page 452.	

CICS — Defining CICS Resources

The following definitions are required for the CICS Listener Support:

- "Transaction Definitions"
- "Program Definitions" on page 442
- "File Definitions" on page 443
- "Transient Data Definition" on page 443

Note: With z/VSE all these definitions have been activated using member IESCSEZA.Z and IESZDCT.A in IJSYSRS.SYSLIB. This setup includes the definition of TASKDATAKEY(CICS) for transactions and EXECKEY(CICS) for programs. This is required when running with CICS storage protection. These definitions are ignored when running without CICS storage protection.

For information on defining transactions, programs, and files to the CICS Resource Definition Online (RDO) facility, refer to CICS/ESA Resource Definition (Online).

Transaction Definitions

The following four transactions are required to support the CICS Listener:

EZAC Configure the socket interface

Setting Up / Configuring CICS Listener Support

EZAO Enable the socket interface

EZAP Internal transaction that is invoked during termination of the socket interface

EZAL Listener task

Note: This is a single listener. Each listener in the same CICS partition needs a unique transaction ID.

Tip: For transactions EZAL, EZAO, and EZAP a priority of 255 has been defined. This ensures timely transaction dispatching, and in case of EZAL maximizes the connection rate of clients requesting service.

Using Storage Protection

When running with CICS storage protection, the EZAP, EZAO, and EZAL transactions must be defined with TASKDATAKEY(CICS). If this is not done, EZAO fails with an ASRA abend code indicating an incorrect attempt to overwrite the CDSA by EZACIC01.

If the machine does not support storage protection or is not enabled for storage protection, TASKDATAKEY(CICS) is ignored and does not cause an error.

Note:

1. Use of the IBM-supplied Listener is not required.
2. You may use a transaction name other than EZAL.
3. The TASKDATA Loc values for EZAO and EZAP and the TASKDATA Loc value for EZAL must all be the same.

Program Definitions

The following programs and one map set are required:

EZACIC00

Is the connection manager program. It provides the enabling and disabling of CICS TCP/IP through the transactions EZAO and EZAP.

EZACIC01

Is the task-related user exit (TRUE).

EZACIC02

Is the Listener program that is used by the transaction EZAL. This transaction is started when you enable CICS TCP/IP Listener through the EZAO transaction.

Note: While you do not need to use the IBM-supplied Listener, you do need to provide a Listener function.

EZACIC20

Is the initialization/termination front-end module for the CICS Listener Interface.

EZACIC21

Is the initialization module for the CICS Listener Interface.

EZACIC22

Is the termination module for the CICS Listener Interface.

EZACIC23

Is the primary module for the configuration transaction (EZAC).

EZACIC24

Is the message delivery module for transactions EZAC and EZAO.

EZACIC25

Is the Domain Name Server (DNS) cache module.

EZACICME

Is the US English text delivery module.

EZACICM

has all the maps that are used by the transactions.

EZASOH00, EZASOH99

Interface modules for the TCP/IP API used by the CICS Listener.

Using Storage Protection

When running with CICS Storage Protection, all the required CICS Listener programs must have EXECKEY=CICS as part of their CEDA definitions.

If the machine does not support storage protection or is not enabled for storage protection, EXECKEY(CICS) is ignored and does not cause an error.

File Definitions

The updates to CICS include two files: EZACONF, the CICS Listener configuration file, and EZACACH, which is required if you want to use the Domain Name Server Cache function (EZACIC25).

Transient Data Definition

The CICS Listener Support uses a transient data queue for messages. With z/VSE, the EZAM transient data queue is predefined and can be used by the CICS Listener Support as well as by your own socket applications. The name of the transient data queue can be changed.

If so, it must match the name that is specified in the **ERRORTD** parameter of the **EZAC DEFINE CICS** dialog, the **EZACICD TYPE=CICS** macro call, or both. Refer to “Building the Configuration data set with the Configuration Macro (EZACICD)” on page 446.

The Listener transaction can start a server using a transient data queue, as described in “Listener Input Format (Standard Listener only)” on page 502. Following is a DCT entry for an application that is started using the trigger-level mechanism of the DCT.

```

DFHDCT TYPE=INTRA,                                X
          DESTID=TRAA,                              X
          DESTFAC=FILE,                             X
          TRIGLEV=1,                                 X
          TRANSID=TRAA
          ...
          ...
    
```

Figure 40. Addition to the DCT Required by CICS TCP/IP

CICS Monitoring

Optionally, the CICS Listener Interface uses the CICS Monitoring Facility to collect data about its operation. Event Monitoring Points (EMPs) with identifier 'EZA02' are used by the Listener to collect performance class data.

Event Monitoring Points for the Listener

The Listener monitors the activities associated with connection acceptance and server task startup.

The listener counts the following events:

- Number of Connection Requested Accepted
- Number of Transactions Started
- Number of Transactions Rejected Due To Invalid Transaction ID
- Number of Transactions Rejected Due To Disabled Transaction
- Number of Transactions Rejected Due To Disabled Program
- Number of Transactions Rejected Due To Givesocket Failure
- Number of Transactions Rejected Due To Negative Response from Security Exit
- Number of Transactions Not Authorized to Run
- Number of Transactions Rejected Due to I/O Error
- Number of Transactions Rejected Due to No Space
- Number of Transactions Rejected Due to TD Length Error

The following Monitor Control Table (MCT) entries make use of the event-monitoring points in the performance class used by the Listener.

Figure 41. The Monitor Control Table (MCT) for Listener

DFHMCT	TYPE=EMP, ID=(EZA02.01), CLASS=PERFORM, PERFORM=ADDCNT(1,1)	X
DFHMCT	TYPE=EMP, ID=(EZA02.02), CLASS=PERFORM, PERFORM=ADDCNT(2,1)	X
DFHMCT	TYPE=EMP, ID=(EZA02.03), CLASS=PERFORM, PERFORM=ADDCNT(3,1)	X
DFHMCT	TYPE=EMP, ID=(EZA02.04), CLASS=PERFORM, PERFORM=ADDCNT(4,1)	X
DFHMCT	TYPE=EMP, ID=(EZA02.05), CLASS=PERFORM, PERFORM=ADDCNT(5,1)	X
DFHMCT	TYPE=EMP, ID=(EZA02.06), CLASS=PERFORM, PERFORM=ADDCNT(6,1)	X
DFHMCT	TYPE=EMP, ID=(EZA02.07), CLASS=PERFORM, PERFORM=ADDCNT(7,1)	X
DFHMCT	TYPE=EMP, ID=(EZA02.08), CLASS=PERFORM, PERFORM=ADDCNT(8,1)	X
DFHMCT	TYPE=EMP, ID=(EZA02.09), CLASS=PERFORM, PERFORM=ADDCNT(9,1)	X
DFHMCT	TYPE=EMP, ID=(EZA02.10), CLASS=PERFORM, PERFORM=ADDCNT(10,1)	X
DFHMCT	TYPE=EMP, ID=(EZA02.11), CLASS=PERFORM, PERFORM=ADDCNT(11,1)	X
DFHMCT	TYPE=EMP, ID=(EZA02.12), CLASS=PERFORM, PERFORM=(MLTCNT(1,11)), COUNT=(1,CONN,STARTED,INVALID,DISTRAN,DISPROG,GIVESOKT,SECEXIT)	X

The following specifications are used in the ID parameter:

- (EZA02.01)
Completion of ACCEPT call.
- (EZA02.02)
Completion of CICS transaction initiation.
- (EZA02.03)
Detection of Invalid Transaction ID.
- (EZA02.04)
Detection of Disabled Transaction.
- (EZA02.05)
Detection of Disabled Program.
- (EZA02.06)
Detection of Givesocket Failure.
- (EZA02.07)
Transaction Rejection by Security Exit.
- (EZA02.08)
Transaction Not Authorized
- (EZA02.09)
I/O Error on Transaction Start.
- (EZA02.10)
No Space Available for TD Start Message
- (EZA02.11)
TD Length Error
- (EZA02.12)
Program Termination.

CICS Program List Table (PLT)

You can allow automatic startup/shutdown of the CICS Listener Interface through updates to the PLT. This is achieved through placing the EZACIC20 module in the appropriate PLT.

To start the CICS Listener Interface interface automatically, make the following entry in PLTPI *after* the DFHDELIM entry:

```
DFHPLT TYPE=ENTRY,PROGRAM=EZACIC20
```

To shut down the CICS Listener Interface interface automatically, make the following entry in PLTSD *before* the DFHDELIM entry:

```
DFHPLT TYPE=ENTRY,PROGRAM=EZACIC20
```

Configuring the CICS TCP/IP Environment

The CICS Listener Configuration File (EZACONF) contains information about the CICS Listener environment.

The file is organized by two types of objects—CICS instances and listeners within those instances. The creation of this dataset is done in three stages:

1. Create the empty dataset using VSAM IDCAMS (Access Method Services). For z/VSE the configuration file is preallocated, but empty.

These are the preallocated VSAM definition statements:

Setting Up / Configuring CICS Listener Support

```
DEFINE CLUSTER(NAME(VSE.EZACICS.CONFIG) -  
  RECORDS(3000 2000) -  
  SHAREOPTIONS(2) -  
  RECORDSIZE(150,150) -  
  VOLUMES(SYSWK1,DOSRES) -  
  NOREUSE -  
  INDEXED -  
  FREESPACE(15 7) -  
  KEYS (16,0) -  
  NOCOMPRESSED -  
  TO (99366)) -  
  DATA (NAME(VSE.EZACICS.CONFIG.@D@) -  
  CONTROLINTERVALSIZE (4096)) -  
  INDEX (NAME(VSE.EZACICS.CONFIG.@I@)) -  
  CATALOG (VSESP.USER.CATALOG)
```

2. Initialize the dataset using the program generated by the EZACICD macro. See member SKCICSLI in ICCF library 59 for a sample job to initialize the configuration file.
3. Add to or modify the dataset using the configuration transaction EZAC. This step is described in "Customizing the Configuration Dataset" on page 451.¹

Building the Configuration data set with the Configuration Macro (EZACICD)

The configuration macro (EZACICD) is used to build the configuration data set.

This data set can then be incorporated into CICS using RDO and can be modified using the configuration transactions. See "Configuration Transaction (EZAC)" on page 452. The macro is keyword driven. The TYPE keyword controls the specific function request. The data set contains one record for each instance of CICS it supports, and one record for each listener. The following is an example of the macros that are required to create a configuration file for two instances of the CICS Listener interface:

```
EZACICD TYPE=INITIAL,      Start of macro assembly input      X  
  FILNAME=EAZCONF,       Name for configuration file          X  
  PRGNAME=EZAACONFP      Name of batch program to run  
EZACICD TYPE=CICS,       CICS record definition              X  
  APPLID=DBDCCICS,      APPLID of CICS                      X  
  TCPADDR=SOCKET00,     Name of TCP/IP Address              X  
  PLTSDI=YES,           PLT shutdown method is immediately  X  
  CACHMIN=15,           Minimum refresh time for cache      X  
  CACHMAX=30,           Maximum refresh time for cache      X  
  CACHRES=10,           Maximum number of resident resolvers X  
  ERRORTD=CSMT,        Transient data queue for error msgs  X  
  SMSGSUP=NO,          STARTED Messages Suppressed?  
EZACICD TYPE=LISTENER,   Listener record definition          X  
  FORMAT=STANDARD,     Standard Listener                  X  
  APPLID=DBDCCICS,     Applid of CICS region              X  
  TRANID=EZAL,         Transaction name for Listener       X  
  PORT=3010,           Port number for Listener            X  
  AF=INET,             Listener Address Family            X  
  IMMED=YES,           Listener starts up at initialization? X  
  BACKLOG=20,          Backlog value for Listener          X  
  NUMSOCK=50,          # of sockets supported by Listener  X  
  MINMSGSL=4,          Minimum input message length       X  
  ACCTIME=30,          Timeout value for Accept            X  
  GIVTIME=30,          Timeout value for Givesocket       X  
  REATIME=30,          Timeout value for Read              X  
  RTYTIME=10,          Wait 10 seconds for TCP to come back X  
  TRANTRN=YES,         Is TRANUSR=YES conditional?        X
```

1. The EZAC transaction is modeled after the CEDA transaction used by CICS Resource Definition Online (RDO).

TRANUSR=YES,	Translate user data?	X
SECEXIT=EZACICSE	Name of security exit program	
EZACICD TYPE=LISTENER,	Listener record definition	X
FORMAT=ENHANCED,	Enhanced Listener	X
APPLID=DBDCCICS,	Applid of CICS region	X
TRANID=CSKM,	Transaction name for Listener	X
PORT=3011,	Port number for Listener	X
AF=INET,	Listener Address Family	X
IMMED=YES,	Listener starts up at initialization?	X
BACKLOG=20,	Backlog value for Listener	X
NUMSOCK=50,	# of sockets supported by Listener	X
ACCTIME=30,	Timeout value for Accept	X
GIVTIME=30,	Timeout value for Givesocket	X
REETIME=30,	Timeout value for Read	X
RTYTIME=20,	Wait 20 seconds for TCP to come back	X
CSTRAN=TRN1,	Name of child IPv4 server transaction	X
CSSTYP=KC,	Child server startup type	X
CSDELAY=000000,	Child server delay interval	X
MSGLEN=0,	Length of input message	X
MSGFORM=ASCII,	Output message format	X
SECEXIT=EZACICSE	Name of security exit program	
EZACICD TYPE=FINAL	End of assembly input	

TYPE Parameter

The TYPE parameter controls the function requests. It can have the following values:

INITIAL

Initializes the generation environment. Define this value once per generation at the first invocation of the macro. If **TYPE=INITIAL** is specified, the following parameters apply:

PRGNAME=EZACONFP |XXXXXXXX

The name of the generated initialization program. The default is **EZACONFP**.

FILNAME=EZACONF |XXXXXXXX

The file name that is used for the configuration file in the execution of the initialization program. The default is **EZACONF**.

CICS

Identifies a CICS object. The object corresponds to a specific instance of CICS and creates a configuration record. If **TYPE=CICS** is specified, the following parameters apply:

APPLID=xxx

The APPLID of the CICS address space in which this instance of CICS Listener is to run. This parameter is mandatory.

CACHMAX=30 |nnn

The maximum refresh time for the domain name server cache in minutes. This value depends on the stability of your network, that is, the time you would expect a domain name to have the same IP address. Higher values improve performance but increase the risk of getting an incorrect (expired) address when resolving a name. The value must be greater than **CACHMIN**. The default value is 30. Possible values are 0 - 999.

CACHMIN=15 |nnn

The minimum refresh time for the domain name server cache in minutes. This value depends on the stability of your network, that is, the time you would expect a domain name to have the same IP address. Higher values improve performance but increase the risk of getting an incorrect (expired)

Setting Up / Configuring CICS Listener Support

address when resolving a name. The value must be less than **CACHMAX**. The default value is 15. Possible values are 0 - 998.

CACHRES=10 | *nn*

The maximum number of concurrent resolvers wanted. If the number of concurrent resolvers is equal to or greater than this value, refresh of cache records does not occur unless their age is greater than the **CACHMAX** value. The default is 10. Possible values are 0 - 99.

ERRORTD=EZAM | *xxx*

The name of a transient data destination to which error messages are written. The default is **EZAM**.

PLTSDI=NO | **YES**

If the CICS TCP/IP Listener Support is shut down using the EZACIC20 PLT program, the **PLTSDI** parameter specifies whether the interface is shut down immediately or not. The default is **NO** and specifies a deferred shutdown. **YES** specifies an immediate shutdown. If the **PLTSDI** parameter is not specified, a deferred shutdown is performed. A deferred shutdown enables all CICS Listeners to end gracefully. An immediate shutdown directs all CICS listeners to be immediately terminated.

SMSGSUP=NO | **YES**

If **YES** is specified, messages EZY1318E, EZY1325I, and EZY1330I are suppressed. The default is **NO**, and allows these messages to be issued.

TCPADDR=SOCKET00 | *nn*

The name of the z/VSE TCP/IP address space. z/VSE supports **SOCKET***nn* where *nn* is the ID of the TCP/IP stack to connect with.

LISTENER

Identifies the listener object. This creates a listener record. If **TYPE=LISTENER** is specified, the following parameters apply:

ACCTIME=60 | *nn*

The time in seconds the listener waits for a connection request before checking for a CICS Listener interface shutdown or CICS shutdown. The default is 60. Setting this value high minimizes CPU consumption on a lightly loaded system but lengthens shutdown processing. Setting this value low uses more CPU but facilitates shutdown processing.

AF=INET | **INET6**

Specifies, if the defined listener supports IPv6 partners and is able to give an IPv6 socket descriptor to an IPv6 child server program. **INET6** indicates that the listener gives an IPv6 socket to the child server program. **INET**, the default, indicates that the listener gives an IPv4 socket to the child server program. Ensure that the child server program performing the **TAKESOCKET** command matches the domain of the socket being given by the listener.

APPLID=xxx

The **APPLID** of the CICS object for which this listener is being defined. If this parameter is omitted, the **APPLID** from the previous **TYPE=CICS** macro is used.

BACKLOG=20 | *nn*

The number of unaccepted connections that can be queued to this listener. The default value is 20.

Setting Up / Configuring CICS Listener Support

Note: If you use TCP/IP for VSE/ESA stack of CSI International, a value of 1 is always used, because of the stacks implementation of the backlog feature.

CSDELAY=hhmmss

(Enhanced CICS Listener only) This parameter is applicable only if **CSSTYP** is **IC**. It specifies the delay interval to be used on the **EXEC CICS START** command. The format is: *hhmmss* (hours/minutes/seconds).

CSSTYP=IC|KC|TD

(Enhanced CICS Listener only) This parameter specifies the default start method for the child server task. This can be overridden by the security/transaction exit. Possible values are **IC**, **KC**, and **TD**.

IC Indicates that the child server task is started using **EXEC CICS START** with the value specified by **CSDELAY** (or an overriding value from the security/transaction exit) as the delay interval.

KC Indicates that the child server task is started using **EXEC CICS START** with no delay interval. This is the default.

TD Indicates that the child server task is started using the **EXEC CICS WRITEQ TD** command, which uses transient data to trigger the child server task.

CSTRAN=xxx

(Enhanced CICS Listener only) This mandatory parameter specifies the default child server transaction that the listener starts. **CSTRAN** can be overridden by the security/transaction exit. The child server transaction is verified to be defined to CICS and enabled when the listener is started by the **EZAO Operator** transaction.

FORMAT=STANDARD|ENHANCED

STANDARD, which is the default, indicates the original CICS listener that requires the client to send the standard header. **ENHANCED** indicates the enhanced CICS listener that does not expect the standard header from the client.

GIVTIME=nn

The time in seconds the listener waits for a response to a **GIVESOCKET** request. If the time expires, the listener assumes that either the server transaction did not start or the **TAKESOCKET** failed. The listener sends the client a message that indicates the server failed to start and closed the socket (connection). If this parameter is not specified, the **ACCTIME** value is used.

IMMED=YES|NO

YES, which is the default, indicates that the listener is started when the interface starts. **NO** indicates that the listener is started independently using the **EZAO** transaction.

MINMSGL=4|nn

(Standard CICS Listener only) This parameter specifies the minimum length of the Transaction Initial Message from the client to the listener. The default value is 4. The listener continues to read on the connection until this length of data has been received.

MSGFORM=ASCII|EBCDIC

(Enhanced CICS Listener only) This parameter indicates, if an error message returned to the client is in ASCII or EBCDIC. ASCII is the default. **MSGFORM** is displayed as **MSGFORMat** on the **EZAC** screens.

Setting Up / Configuring CICS Listener Support

MSGLEN=0 | *nnn*

(Enhanced CICS Listener only) This parameter specifies the length of the data to be received from the client. The valid range is 0 - 999. If the value is 0, the listener does not read in any data from the client.

NUMSOCK=50 | *nnn*

The number of sockets that are supported by the listener. One socket is the listening socket. The other sockets are used to pass connections to the servers using the GIVESOCKET call. The maximum number of concurrent GIVESOCKET requests that can be active, is this value minus one. The default value is 50.

PORT=nnnnn

The port number the listener uses for accepting connections. This parameter is mandatory. The ports can be shared. Possible values are 1 - 65535.

REETIME=nn

The time in seconds this listener waits for a response to a READ request. If this time expires, the listener assumes that the client has failed and terminates the connection by closing the socket. If this parameter is not specified, no checking for read timeout is done.

RTYTIME=15 | *nnn*

Specifies the length of time, in seconds, that the listener waits after a TCP/IP stack outage occurs before it attempts to connect or reconnect. If you specify 0, the listener cleans up any resources and ends. A value greater than 0 and smaller than 15 results in an RTYTIME value of 15 seconds. The listener task is delayed 15 seconds before it attempts to connect or reconnect. The stack that it tries to connect to is the stack that is specified by the listener's IP CICS socket interface TCPADDR configuration option. If the connection fails, the listener task is delayed for the length of time that is specified by the RTYTIME parameter. After this interval lapses, the listener attempts to connect to its stack. The listener continues to attempt to connect to the stack until either it succeeds or is terminated by the operator. The value of *nnn* can range from 0 - 999. The default setting is 15 seconds. The following table shows a summary of the listener's action that is based on the combination of the RTYTIME value and the state of the listener's TCP stack. This parameter is optional.

Table 12. Listener action based on RTYTIME and stack state

Listener	RTYTIME	TCP down	TCP up
Initially started	0	Listener ends	Listener initializes
	>0	Listener waits	
Previously active	0	Listener ends	
	>0	Listener waits	

SECEXIT

The name of the user written security exit that is used by the listener. The default is no security exit. The listener uses the EXEC CICS LINK command to give control to the security exit. A check is made to ensure the specified security exit program is defined to CICS and enabled for use when the listener is started by the EZAO Operator transaction.

TRANID=EZAL | *xxx*

The transaction name for the listener. The default is EZAL.

TRANTRN=YES|NO

(Standard CICS Listener only) YES indicates that the translation of the user data is based on the character format of the transaction code. That is, with YES specified for TRANTRN, the user data is translated if and only if TRANUSR is YES and the transaction code is not uppercase EBCDIC. If NO is specified for TRANTRN, the user data is translated if and only if TRANUSR is YES. The default value for TRANTRN is YES.

Note: Regardless of how TRANTRN is specified, translation of the transaction code occurs only if the first character is not uppercase EBCDIC.

TRANUSR=YES|NO

(Standard CICS Listener only) NO indicates that the user data from the Transaction Initial Message is not translated from ASCII to EBCDIC. YES indicates that the user data is translated depending on TRANTRN and character format of the transaction code as shown in Table 13. The default for TRANUSR is YES.

Note: Previous implementations functioned as if TRANTRN and TRANUSR were both set to YES. Normally, data on the Internet is ASCII and should be translated. The exceptions are data coming from an EBCDIC client or binary data in the user fields. In those cases, you should set these values accordingly. If you are operating in a mixed environment, use multiple listeners on multiple ports.

Table 13. Conditions for Translation of Tranid and User Data

TRANTRN	TRANUSR	Tranid format	Translate tranid?	Translate user data?
YES	YES	EBCDIC	NO	NO
YES	NO	EBCDIC	NO	NO
NO	YES	EBCDIC	NO	YES
NO	NO	EBCDIC	NO	NO
YES	YES	ASCII	YES	YES
YES	NO	ASCII	YES	NO
NO	YES	ASCII	YES	YES
NO	NO	ASCII	YES	NO

FINAL

Indicates the end of the generation. There are no subparameters.

Customizing the Configuration Dataset

There is a CICS object for each CICS that uses the CICS Listener Support and is controlled by the Configuration File. The CICS object is identified by the APPLID of the CICS it references.

There is a Listener object for each Listener defined for a CICS. It is possible that a CICS does not have a Listener but this is not common practice. A CICS can have multiple listeners which are either multiple instances of the supplied Listener with different specifications, multiple user-written listeners or some combination.

Configuration Transaction (EZAC)

The EZAC transaction is a panel-driven interface that lets you add, delete, or modify the Configuration file. The following table lists and describes the functions supported by the EZAC transaction.

Command	Object	Function
"ALTER Function"	CICS/Listener	Modifies the attributes of an existing resource definition.
"COPY Function" on page 456	CICS/Listener	<ul style="list-style-type: none"> • CICS - Copies the CICS object and its associated listeners to create another CICS object. COPY will fail if the new CICS object already exists. • Listener - Copies the Listener object to create another Listener object. COPY will fail if the new Listener object already exists.
"CONVERT Function" on page 458	Listener	Converts from a Standard Listener to an Enhanced Listener and vice versa.
"DEFINE Function" on page 459	CICS/Listener	Create a new resource definition.
"DELETE Function" on page 463	CICS/Listener	<ul style="list-style-type: none"> • CICS - Deletes the CICS object and all of its associated listeners. • Listener - Deletes the Listener object.
"DISPLAY Function" on page 464	CICS/Listener	Shows the parameters specified for the CICS/Listener object.
"RENAME Function" on page 467	CICS/Listener	Performs a COPY followed by a DELETE of the original object.

If you enter EZAC, the following screen is displayed:

```

EZAC                                     APPLID=DBDCCICS
  ENTER ONE OF THE FOLLOWING
Alter
Convert
COpy
DEFine
DELeTe
DISplay
REName

PF 1 HELP      3 END      6 CRSR      9 MSG      12 CNCL
    
```

ALTER Function

The ALTER function is used to change CICS objects, their Listener objects, or both.

Setting Up / Configuring CICS Listener Support

If you specify ALter on the EZAC Initial Screen or enter EZAC AL on a blank screen, the following screen is displayed:

```
EZAC,ALTER,                                     APPLID=DBDCCICS
Enter One of the Following
CICS
LISTENER

PF 3 END                                         12 CNCL
```

Fast path: You can shortcut this by entering either EZAC ALTER CICS or EZAC ALTER LISTENER.

ALTER CICS

For alteration of a CICS object, the following screen is displayed:

```
EZAC,ALTER,CICS                                 APPLID=DBDCCICS
ENTER ALL FIELDS
APPLID      ===>                               APPLID of CICS System

PF 3 END                                         12 CNCL
```

After the APPLID is entered, the following screen is displayed.

Setting Up / Configuring CICS Listener Support

```
EZAC,ALTER,CICS                                APPLID = xxxxxxxx

OVERTYPE TO ENTER

APPLID      ==> XXXXXXXX      APPLID of CICS System
TCPADDR     ==> xxxxxxxx      Name of TCP Address Space
CACHMIN     ==> xxx           Minimum Refresh Time for Cache
CACHMAX     ==> xxx           Maximum Refresh Time for Cache
CACHRES     ==> xxx           Maximum number of Resolvers
ERRORTD     ==> xxxxx        TD Queue for Error Messages
MSGSUP      ==> xxx           Suppress Task Started Messages
PLTSDI      ==> xxx           CICS PLT Shutdown Immediately

Press enter to confirm function
or PF12 to make more changes

PF 3 END                                         12 CNCL
```

The system requests a confirmation of the values displayed. After the changes are confirmed, the changed values are in effect for the next initialization of the CICS sockets interface.

ALTER LISTENER

For alteration of a Listener, the following screen is displayed:

```
EZAC,ALTER,                                     APPLID=DBDCCICS
ENTER ALL FIELDS

APPLID      ==>                                     APPLID of CICS System
NAME        ==>                                     Transaction Name of Listener

APPLID=DBDCCICS

PF 3 END                                         12 CNCL
```

After the names are entered, the following screen is displayed:

Note: Screen 1 for the Enhanced Listener is similar to screen 1 for the Standard Listener.

Setting Up / Configuring CICS Listener Support

EZAC,ALTER,LISTENER (standard listener. screen 1 of 2) APPLID = xxxxxxxx

OVERTYPE TO ENTER

APPLID	====> xxxxxxxx	APPLID of CICS System
TRANID	====> xxxxxx	Transaction Name of Listener
PORT	====> xxxxxx	Port Number of Listener
AF	====> xxxxxx	Listener Address Family
IMMEDIATE	====> xxx	Immediate Startup Yes No
BACKLOG	====> xxx	Backlog Value for Listener
NUMSOCK	====> xxx	Number of Sockets in Listener
ACCTIME	====> xxx	Timeout Value for ACCEPT
GIVTIME	====> xxx	Timeout Value for GIVESOCKET
RETIME	====> xxx	Timeout Value for READ
RTYTIME	====> xxx	Stack Connection Retry Time

Verify parameters, press PF8 to go to screen 2

PF 3 END

8 NEXT

12 CNCL

The following screen 2 is displayed for the Standard Listener:

EZAC,ALTER,LISTENER (standard listener. screen 2 of 2) APPLID = xxxxxxxx

Overtime to Enter

MINMSGL	====> xxx	Minimum Message Length
TRANTRN	====> xxx	Translate TRNID Yes No
TRANUSR	====> xxx	Translate User Data Yes No
SECEXIT	====> xxxxxxxx	Name of Security Exit

Verify parameters, press PF7 to go back to screen 1
or ENTER if finished making changes

PF 3 END

7 PREV

12 CNCL

The following screen 2 is displayed for the Enhanced Listener:

Setting Up / Configuring CICS Listener Support

```
EZAC,ALTER,LISTENER (enhanced listener.  screen 2 of 2)  APPLID = xxxxxxxx

Overtpe to Enter

CSTRANid  ==> XXXX          Child Server Transaction Name
CSSTYPe   ==> xx           Startup Method (KC|IC|TD)
CSDELAY   ==> xxxxxx       Delay Interval (hhmmss)
MSGLENgth ==> xxx         Message Length (0-999)
MSGFORMat ==> xxxxx       Enter ASCII|EBCDIC
USEREXIT  ==>             Name of User/Security exit

Verify parameters, press PF7 to go back to screen 1
                        or ENTER if finished making changes

PF 3 END                7 PREV                12 CNCL
```

The system requests a confirmation of the values displayed. After the changes are confirmed, the changed values are in effect for the next initialization of the CICS sockets interface.

COPY Function

The COPY function is used to copy an object into a new object.

If you specify COPY on the EZAC initial Screen or enter EZAC CO on a blank screen, the following screen is displayed:

```
EZAC,COPY,                                APPLID=DBDCCICS

Enter One of the Following

CICS
LISTENER

PF 3 END                                12 CNCL
```

Fast path: You can shortcut this by entering either EZAC COPY CICS or EZAC COPY LISTENER.

COPY CICS

If you specify CICS on the previous screen, the following screen is displayed:

```

EZAC,COPY,                                     APPLID=DBDCCICS
ENTER ALL FIELDS
SCICS      ==> .....                          APPLID of Source CICS
TCICS      ==> .....                          APPLID of Target CICS

PF 3 END                                     9 MSG                                     12 CNCL
    
```

After the APPLIDs of the source CICS object and the target CICS object are entered, confirmation is requested. When confirmation is entered, the copy is performed.

COPY LISTENER

If you specify COPY LISTENER, the following screen is displayed:

```

EZAC,COPY,                                     APPLID=DBDCCICS
ENTER ALL FIELDS
SCICS      ==> .....                          APPLID of Source CICS
SLISTener  ==> ....                          Transaction Name of Source Listener
TCICS      ==> .....                          APPLID of Target CICS
TLISTener  ==> ....                          Transaction Name of Target Listener

PF 3 END                                     12 CNCL
    
```

After the APPLIDs of the source and target CICS objects and the names of the source and target listeners are entered, confirmation is requested. When the confirmation is entered, the copy is performed.

Setting Up / Configuring CICS Listener Support

CONVERT Function

The CONVERT function is used to convert a Standard Listener to an Enhanced Listener.

If you specify CONVERT on the EZAC initial screen or enter EZAC CONVERT on a blank screen, the following screen is displayed:

```
EZAC,CONVERT,LISTENER                                APPLID = xxxxxxxx

OVERTYPE TO ENTER

APPLID      ==> xxxxxxxx          APPLID of CICS System
TRANID      ==> xxxx              Transaction Name of Listener
Format      ==> xxxxxxxx          Enter STANDARD|ENHANCED

PF 3 END                                           12 CNCL
```

After the names are entered, the following screen is displayed:

Note: Screen 1 for the Enhanced Listener is similar to screen 1 for the Standard Listener.

```
EZAC,CONVERT,LISTENER (standard listener. screen 1 of 2) APPLID = xxxxxxxx

OVERTYPE TO ENTER

APPLID      ==> xxxxxxxx          APPLID of CICS System
TRANID      ==> xxxxx             Transaction Name of Listener
PORT        ==> xxxxx             Port Number of Listener
AF          ==> xxxxx             Listener Address Family
IMMEDIATE   ==> xxx              Immediate Startup Yes|No
BACKLOG     ==> xxx              Backlog Value for Listener
NUMSOCK     ==> xxx              Number of Sockets in Listener
ACCTIME     ==> xxx              Timeout Value for ACCEPT
GIVTIME     ==> xxx              Timeout Value for GIVESOCKET
REACTIME    ==> xxx              Timeout Value for READ
RTYTIME     ==> xxx              Stack Connection Retry Time

Verify parameters, press PF8 to go to screen 2

PF 3 END                                           8 NEXT                                           12 CNCL
```

The following screen 2 is displayed for the Standard Listener:

Setting Up / Configuring CICS Listener Support

```
EZAC,CONVERT,LISTENER (standard listener. screen 2 of 2) APPLID = xxxxxxxx
```

Overtyp e to Enter

MINMSGL	====> xxx	Minimum Message Length
TRANTRN	====> xxx	Translate TRNID Yes No
TRANUSR	====> xxx	Translate User Data Yes No
SECEXIT	====> xxxxxxxx	Name of Security Exit

Verify parameters, press PF7 to go back to screen 1
or ENTER if finished making changes

PF 3 END

7 PREV

12 CNCL

The following screen 2 is displayed for the Enhanced Listener:

```
EZAC,CONVERT,LISTENER (enhanced listener. screen 2 of 2) APPLID = xxxxxxxx
```

Overtyp e to Enter

CSTRANid	====> XXXX	Child Server Transaction Name
CSSTYPe	====> xx	Startup Method (KC IC TD)
CSDELAY	====> xxxxxx	Delay Interval (hhmmss)
MSGLENgth	====> xxx	Message Length (0-999)
MSGFORMat	====> xxxxx	Enter ASCII EBCDIC
USEREXIT	====>	Name of User/Security exit

Verify parameters, press PF7 to go back to screen 1
or ENTER if finished making changes

PF 3 END

7 PREV

12 CNCL

After the definition is entered, confirmation is requested. When confirmation is entered, the object is created on the configuration file.

DEFINE Function

The DEFINE function is used to create CICS objects and their Listener objects.

If you specify DEFine on the EZAC initial screen or enter EZAC DEF on a blank screen, the following screen is displayed:

Setting Up / Configuring CICS Listener Support

```
EZAC,DEFINE,                                APPLID=DBDCCICS
Enter One of the Following
CICS
LISTENER

PF 3 END                                    12 CNCL
```

Fast path: You can shortcut this by entering either EZAC DEFINE CICS or EZAC DEFINE LISTENER.

DEFINE CICS

For definition of a CICS object, the following screen is displayed:

```
EZAC,DEFINE,CICS                             APPLID=DBDCCICS
ENTER ALL FIELDS
APPLID      ===>                             APPLID of CICS System

PF 3 END                                    12 CNCL
```

After the APPLID is entered, the following screen is displayed.

```

EZAC,DEFINE,CICS                                APPLID = xxxxxxxx

OVERTYPE TO ENTER

APPLID      ==> xxxxxxxx          APPLID of CICS System
TCPADDR     ==> xxxxxxxx          Name of TCP Address Space
CACHMIN     ==> xxx              Minimum Refresh Time for Cache
CACHMAX     ==> xxx              Maximum Refresh Time for Cache
CACHRES     ==> xxx              Maximum number of Resolvers
ERRORTD     ==> xxxxx            TD Queue for Error Messages
SMSGSUP     ==> xxx              Suppress Task Started Messages
PLTSDI      ==> xxx              CICS PLT Shutdown Immediately

Press enter to confirm function function
or PF12 to make more changes

PF 3 END                                         12 CNCL
    
```

After the definition is entered, confirmation is requested. When confirmation is entered, the object is created on the configuration file.

DEFINE LISTENER

For definition of a Listener, the following screen is displayed:

```

EZAC,DEFINE,LISTENER                            APPLID = xxxxxxxx

OVERTYPE TO ENTER

APPLID      ==> xxxxxxxx          APPLID of CICS System
TRANID      ==> xxxxx            Transaction Name of Listener
Format      ==> xxxxxxxx          Enter STANDARD|ENHANCED

PF 3 END                                         12 CNCL
    
```

After the names are entered, the following screen is displayed:

Note: Screen 1 for the Enhanced Listener is similar to screen 1 for the Standard Listener.

Setting Up / Configuring CICS Listener Support

EZAC,DEFINE,LISTENER (standard listener. screen 1 of 2) APPLID = xxxxxxxx

OVERTYPE TO ENTER

APPLID	====> xxxxxxxx	APPLID of CICS System
TRANID	====> xxxxxx	Transaction Name of Listener
PORT	====> xxxxxx	Port Number of Listener
AF	====> xxxxxx	Listener Address Family
IMMEDIATE	====> xxx	Immediate Startup Yes No
BACKLOG	====> xxx	Backlog Value for Listener
NUMSOCK	====> xxx	Number of Sockets in Listener
ACCTIME	====> xxx	Timeout Value for ACCEPT
GIVTIME	====> xxx	Timeout Value for GIVESOCKET
REACTIME	====> xxx	Timeout Value for READ
RTYTIME	====> xxx	Stack Connection Retry Time

Verify parameters, press PF8 to go to screen 2

PF 3 END

8 NEXT

12 CNCL

The following screen 2 is displayed for the Standard Listener:

EZAC,DEFINE,LISTENER (standard listener. screen 2 of 2) APPLID = xxxxxxxx

Overtime to Enter

MINMSGL	====> xxx	Minimum Message Length
TRANTRN	====> xxx	Translate TRNID Yes No
TRANUSR	====> xxx	Translate User Data Yes No
SECEXIT	====> xxxxxxxx	Name of Security Exit

Verify parameters, press PF7 to go back to screen 1
or ENTER if finished making changes

PF 3 END

7 PREV

12 CNCL

The following screen 2 is displayed for the Enhanced Listener:


```

EZAC,DEFINE,LISTENER (enhanced listener.  screen 2 of 2)  APPLID = xxxxxxxx

Overtpe to Enter

CSTRANid   ==> XXXX           Child Server Transaction Name
CSSTYPe    ==> xx             Startup Method (KC|IC|TD)
CSDELAY    ==> xxxxxx        Delay Interval (hhmmss)
MSGLENgth  ==> xxx           Message Length (0-999)
MSGFORMat  ==> xxxxx        Enter ASCII|EBCDIC
USEREXIT   ==>              Name of User/Security exit

Verify parameters, press PF7 to go back to screen 1
                        or ENTER if finished making changes

PF 3 END                7 PREV                12 CNCL
  
```

After the definition is entered, confirmation is requested. When confirmation is entered, the object is created on the configuration file.

DELETE Function

The DELETE function is used to delete a CICS object or a Listener object.

Deleting a CICS object deletes all Listener objects within that CICS object. If you specify DELETE on the EZAC initial screen or enter EZAC DEL on a blank screen, the following screen is displayed:

```

EZAC,DELETE,                APPLID=DBDCCICS

Enter One of the Following

CICS
LISTENER

PF 3 END                12 CNCL
  
```

DELETE CICS

If you specify DELETE CICS, the following screen is displayed:

```
EZAC,DELETE,CICS                                APPLID=DBDCCICS
  ENTER ALL FIELDS

APPLID      ===>                                APPLID of CICS System

PF          3  END                                12  CNCL
```

After the APPLID is entered, confirmation is requested. When the confirmation is entered, the CICS object is deleted.

DELETE LISTENER

If you specify DELETE LISTENER, the following screen is displayed:

```
EZAC,DELETE,LISTENER                            APPLID=DBDCCICS
  ENTER ALL FIELDS

APPLID      ===>                                APPLID of CICS System
NAME        ===>                                Transaction Name of Listener

PF          3  END                                12  CNCL
```

After the APPLID and listener name are entered, confirmation is requested. When confirmation is entered, the Listener object is deleted.

DISPLAY Function

The DISPLAY function is used to display the specification of an object.

Setting Up / Configuring CICS Listener Support

If you specify DISplay on the initial EZAC screen or enter EZAC DIS on a blank screen, the following screen is displayed:

```
EZAC,DISPLAY,                                APPLID=DBDCCICS
Enter One of the Following
CICS
LISTENER

PF 3 END                                     12 CNCL
```

Fast path: You can shortcut this by entering either EZAC DISPLAY CICS or EZAC DISPLAY LISTENER.

DISPLAY CICS

If you specify DISPLAY CICS, the following screen is displayed:

```
EZAC,DISPLAY,                                APPLID=DBDCCICS
  ENTER ALL FIELDS
APPLID      ===>                            APPLID of CICS System

PF          3 END                             12 CNCL
```

After the APPLID is entered, the following screen is displayed:

Setting Up / Configuring CICS Listener Support

```
EZAC,DISplay,CICS                                APPLID = xxxxxxxx

OVERTYPE TO ENTER

APPLID      ==> XXXXXXXX          APPLID of CICS System
TCPADDR     ==> xxxxxxxx          Name of TCP Address Space
CACHMIN     ==> xxx               Minimum Refresh Time for Cache
CACHMAX     ==> xxx               Maximum Refresh Time for Cache
CACHRES     ==> xxx               Maximum number of Resolvers
ERRORTD     ==> xxxxx            TD Queue for Error Messages
MSGSUP      ==> xxx               Suppress Task Started Messages
PLTSDI      ==> xxx               CICS PLT Shutdown Immediately

Press ENTER or PF3 to exit

PF 3 END                                           12 CNCL
```

DISPLAY LISTENER

If you specify DISPLAY LISTENER, the following screen is displayed:

```
EZAC,DISPLAY,                                     APPLID=DBDCCICS
ENTER ALL FIELDS

APPLID      ==>                    APPLID of CICS System
NAME        ==>                    Transaction Name of Listener

PF 3 END                                           12 CNCL
```

After the APPLID and name are entered, the following screen is displayed

```

EZAC,DISPLAY,LISTENER                                APPLID=DBDCCICS

APPLID      ==> .....          APPLID of CICS System
TRaname     ==> .....          Transaction Name of Listener
POrt        ==> .....          Port Number of Listener
IMMediate   ==> Yes            Immediate Startup          Yes|No
BAcklog     ==> 020            Backlog Value for Listener
NUMsock     ==> 50             Number of Sockets in Listener
MINmsgl     ==> 04             Minimum Message Length
ACCTime     ==> 60             Timeout Value for Accept
GIVTime     ==> 60             Timeout Value for Givesocket
REATime     ==> 10            Timeout Value for Read
FASTread    ==> Yes            Read immediately          Yes|No
TRANtrn     ==> Yes            Translate Trans. Name      Yes|No
TRANusr     ==> Yes            Translate User Data         Yes|No
SECexit     ==> .....          Security Exit Name

PF 3 END                                           12 CNCL
  
```

RENAME Function

The RENAME function is used to rename a CICS or Listener object.

It consists of a COPY followed by a DELETE of the source object. For a CICS object, the object and all of its associated listeners are renamed. For a listener object, only that listener is renamed.

If you specify RENAME on the initial EZAC screen or enter EZAC REN on a blank screen, the following screen is displayed:

```

EZAC,RENAME,                                         APPLID=DBDCCICS

Enter One of the Following

CICS
LISTENER

PF 3 END                                           12 CNCL
  
```

Fast path: You can shortcut this by entering either EZAC RENAME CICS or EZAC RENAME LISTENER.

RENAME CICS

If you specify CICS on the previous screen, the following screen is displayed:

```
EZAC,RENAME,                                APPLID=DBDCCICS
  ENTER ALL FIELDS

SCICS      ==> .....          APPLID of Source CICS
TCICS      ==> .....          APPLID of Target CICS

PF 3 END                                9 MSG                                12 CNCL
```

After the APPLIDs of the source CICS object and the target CICS object are entered, confirmation is requested. When confirmation is entered, the rename is performed.

RENAME LISTENER

If you specify RENAME LISTENER, the following screen is displayed:

```
EZAC,RENAME,                                APPLID=DBDCCICS
  ENTER ALL FIELDS

SCICS      ==> .....          APPLID of Source CICS
SLISTener  ==> ....          Transaction Name of Source Listener
TCICS      ==> .....          APPLID of Target CICS
TLISTener  ==> ....          Transaction Name of Target Listener

PF 3 END                                12 CNCL
```

After the APPLIDs of the source and target CICS objects and the names of the source and target listeners are entered, confirmation is requested. When the confirmation is entered, the rename is performed.

Chapter 17. Configuring the CICS Domain Name Server Cache

Overview of the Domain Name Server Cache

The Domain Name Server (DNS) is like a telephone book that contains a person's name, address, and telephone number. The name server maps a host name to an IP address, or an IP address to a host name. For each host, the name server can contain IP addresses, nicknames, mailing information, and available well-known services (for example, SMTP, FTP, or Telnet).

Translating host names into IP addresses is just one way of using the DNS. Other types of information related to hosts may also be stored and queried. The different possible types of information are defined via input data to the name server in the resource records.

While the CICS Domain Name Server Cache function is optional, it is useful in a highly active CICS client environment. It combines the `gethostbyname()` call supported in TCP/IP for VSE and a cache that saves results from the `gethostbyname()` for future reference. If your system gets repeated requests for the same set of domain names, using the DNS will improve performance significantly.

Function Components

The function consists of three parts.

- A VSAM file which is used for the cache.
- A macro, EZACICR which is used to initialize the cache file.
- A CICS application program, EZACIC25, which is invoked by the CICS application in place of the `gethostbyname` socket call.

VSAM Cache File

The cache file is a VSAM KSDS (Key Sequenced Data Set) with a key of the host name padded to the right with binary zeros. The cache records contain a compressed version of the hostent structure returned by the domain name server plus a time of last refresh field. When a record is retrieved, EZACIC25 determines if it is usable based on the difference between the current time and the time of last refresh.

EZACICR macro

The EZACICR macro builds an initialization module for the cache file, because the cache file must start with at least one record to permit updates by the EZACIC25 module. To optimize performance, you can preload 'dummy' records for the hosts names which you expect to be used frequently. This results in a more compact file and minimizes the I/O required to use the cache. If you do not specify at least one dummy record, the macro will build a single record of binary zeros. See "Step 1: Create the Initialization Module" on page 471.

EZACIC25 Module

This module is a normal CICS application program which is invoked by an EXEC CICS LINK command. The COMMAREA passes information between the invoking

Configuring the CICS Domain Name Server Cache

CICS program and the DNS Module. If domain name resolves successfully, EZACIC25 obtains storage from CICS and builds a hostent structure in that storage. When finished with the hostent structure, release this storage using the EXEC CICS FREEMAIN command.

The EZACIC25 module uses four parameters plus the information passed by the invoking application to manage the cache. These parameters are as follows:

Error Destination

The Transient Data destination to which error messages are sent.

Minimum Refresh Time

The minimum time in minutes between refreshes of a cache record. If a cache record is 'younger' than this time, it will be used. This value is set to 15 (minutes).

Maximum Refresh Time

The maximum time in minutes between refreshes of a cache record. If a cache record is 'older' than this time, it will be refreshed. This value is set to 30 (minutes).

Maximum Resolver Requests

The maximum number of concurrent requests to the resolver. It is set at 10. See "How the DNS Cache Handles Requests."

How the DNS Cache Handles Requests

When a request is received where cache retrieval is specified, the following takes place:

1. Attempt to retrieve this entry from the cache. If not successful, issue gethostbyname unless request specifies cache only.
2. If cache retrieval is successful, calculate the 'age' of the record (the difference between the current time and the time this record was created or refreshed).
 - If the age is not greater than minimum cache refresh, use the cache information and build the Hostent structure for the requestor. Then return to the requestor.
 - If the age is greater than the maximum cache refresh, go issue the gethostbyname call and refresh the cache record with the results.
 - If the age is between the minimum and maximum cache refresh values, do the following:
 - a. Calculate the difference between the maximum and minimum cache refresh times and divide it by the maximum number of concurrent resolver requests. The result is called the time increment.
 - b. Multiply the time increment by the number of currently active resolver requests. Add this time to the minimum refresh time giving the adjusted refresh time.
 - c. If the age of the record is less than the adjusted refresh time, use the cache record.
 - d. If the age of the record is greater than the adjusted refresh time, issue the gethostbyname call and refresh the cache record with the results.
 - If the gethostbyname is issued and is successful, the cache is updated and the update time for the entry is changed to the current time.

Using the DNS Cache

Three steps are needed to use the DNS cache.

1. Create the initialization module, which in turn defines and initializes the file and the EZACIC25 module. See “Step 1: Create the Initialization Module.”
2. Define the cache files to CICS. See “Step 2: Define the Cache File to CICS” on page 473.
3. Use EZACIC25 to replace gethostbyname calls in CICS application modules. See “Step 3: Execute EZACIC25” on page 473.

Step 1: Create the Initialization Module

The initialization module is created using the EZACICR macro. A minimum of two invocations of the macro are coded and assembled and the assembly produces the module. An example follows:

```
EZACICR TYPE=INITIAL
EZACICR TYPE=FINAL
```

This produces an initialization module which creates one record of binary zeros. If you wish to preload the file with dummy records for frequently referenced domain names, it would look like this:

```
EZACICR TYPE=INITIAL
EZACICR TYPE=RECORD,NAME=HOSTA
EZACICR TYPE=RECORD,NAME=HOSTB
EZACICR TYPE=RECORD,NAME=HOSTC
EZACICR TYPE=FINAL
```

where HOSTA, HOSTB, AND HOSTC are the host names you want in the dummy records. The names can be specified in any order.

Parameters

The specifications for the EZACICR macro are as follows:

TYPE There are three acceptable values:

INITIAL

Indicates the beginning of the generation input. This value should only appear once and should be the first entry in the input stream.

RECORD

Indicates a dummy record the user wants to generate. There can be from 0 to 4096 dummy records generated and each of them must have a unique name. Generating dummy records for frequently used host names will improve the performance of the cache file. A TYPE=INITIAL must precede a TYPE=RECORD statement.

FINAL

Indicates the end of the generation input. This value should only appear once and should be the last entry in the input stream. A TYPE=INITIAL must precede a TYPE=FINAL.

AVGREC

The length of the average cache record. This value is specified on the TYPE=INITIAL macro and has a default value of 500. It is recommended that you use the default value until you have adequate statistics to determine a better value. This parameter is the same as the first sub-parameter in the RECORDSIZE parameter of the IDCAMS DEFINE

Configuring the CICS Domain Name Server Cache

statement. Accurate definition of this parameter along with use of dummy records will minimize control interval and control area splits in the cache file.

NAME

Specifies the host name for a dummy record. The name must be from 1 to 255 bytes long. The NAME operand is required for TYPE=RECORD entries.

Within z/VSE the DNS cache file is predefined, but empty. It is defined as VSAM cluster VSE.EZACICS.CACHE within catalog VSESP.USER.CATALOG. Its filename is EZACACH. For a minimum initialization of this file, the following JCL may be used:

```
// JOB   CACHCRE
// EXEC  ASSEMBLY,GO
        EZACICR TYPE=INITIAL
        EZACICR TYPE=FINAL
        END
/*
/ &
```

Be aware that file EZACACH must be closed when running this job.

Once the cache file has been created, it has the following layout:

Host Name

A 255-byte character field specifying the host name. This field is the key to the file.

Record Type

A 1-byte binary field specifying the record type. The value is X'00000001'.

Last Refresh Time

A 4-byte field specifying the last refresh time. It is expressed in seconds since 0000 hours on January 1, 1990 and is derived by taking the ABSTIME value obtained from an EXEC CICS ASKTIME and subtracting the value for January 1, 1990.

Number of Alias Entries

A halfword binary field specifying the number of entries in the Alias array.

Offset to Alias Array List

A halfword binary field specifying the offset in the record to the Alias array. The Alias array consists of alias names each followed by a x '00' byte.

Number of INET Addresses

A halfword binary field specifying the number of INET addresses in the record..

INET Addresses

One or more fullword binary fields specifying INET addresses returned from gethostbyname().

Alias Names

An array of variable length character fields specifying the alias name(s) returned from the domain name server cache. These fields are delimited by a byte of binary zeros. Each of these fields have a maximum length of 255 bytes.

Step 2: Define the Cache File to CICS

All CICS definitions required to add this function to a CICS system are already provided within z/VSE.

This includes the definitions for file EZACACH as well as for program EZACIC25.

Step 3: Execute EZACIC25

EZACIC25 replaces the gethostbyname socket call. It is invoked by a EXEC CICS LINK PROGRAM (EZACIC25) COMMAREA(com-area) where com-area is defined as follows:

Return Code

A fullword binary variable specifying the results of the function:

- 1 ERRNO value returned from gethostbyname() call. Check ERRNO field.
- 0 Host name could not be resolved either within the cache or by use of the gethostbyname call.
- 1 Host name was resolved using cache.
- 2 Host name was resolved using gethostbyname call.

ERRNO

A fullword binary field specifying the ERRNO returned from the GETHOSTBYNAME call.

HOSTENT Address

The address of the returned HOSTENT structure.

Command

A 4-byte character field specifying the requested operation.

GHBN

gethostbyname. This is the only function supported.

Namelen

A fullword binary variable specifying the actual length of the host name for the query.

Query_Type

A 1-byte character field specifying the type of query:

- 0 Attempt query using cache. If unsuccessful, attempt using gethostbyname() call.
- 1 Attempt query using gethostbyname() call. This forces a cache refresh for this entry.
- 2 Attempt query using cache only.

Note: If the cache contains a matching record, the contents of that record will be returned regardless of its age.

Name A 256-byte character variable specifying the host name for the query.

Configuring the CICS Domain Name Server Cache

HOSTENT Structure

The returned HOSTENT structure is shown in Figure 42.

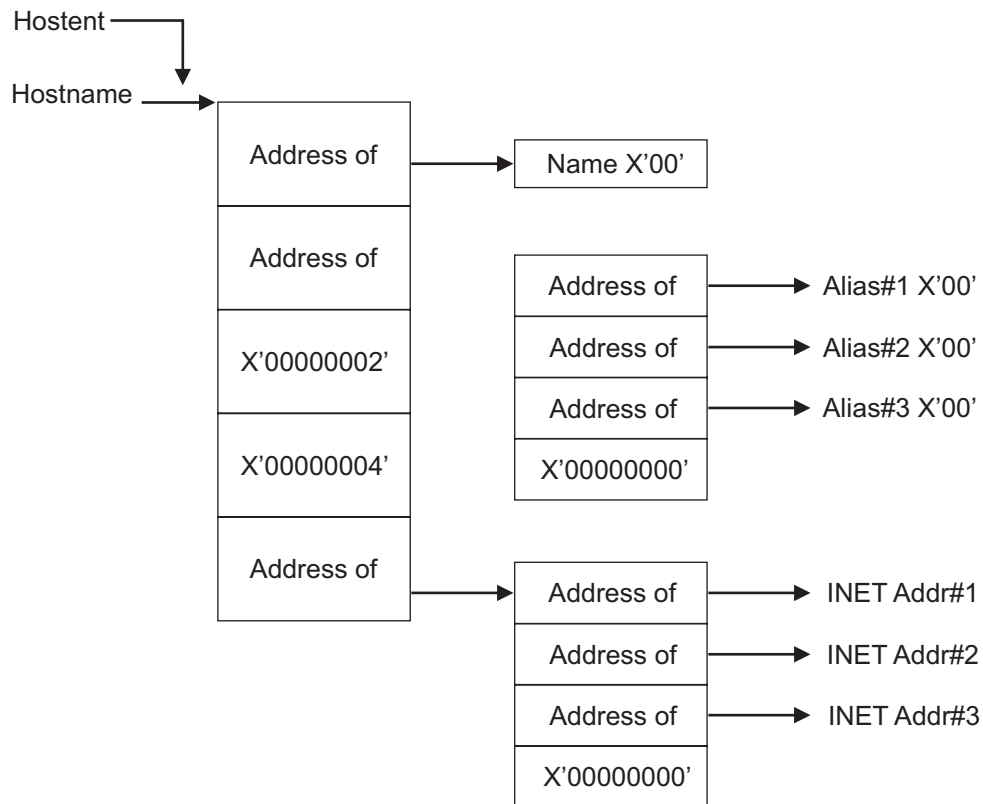


Figure 42. The DNS Hostent

Chapter 18. Starting and Stopping the CICS Listener Support

Overview

This section explains how to start and stop (enable and disable) the CICS Listener Support. It describes:

- You can customize your system so that the CICS Listener Support starts and stops automatically. See “Starting/Stopping CICS Listener Support Automatically.”
- An operator can also start and stop CICS Listener Support manually after CICS has been initialized. See “Starting/Stopping CICS Manually.”
- You can also start and stop CICS Listener Support from a CICS application program. See “Starting/Stopping CICS Listener Support with Program Link” on page 480.

This section describes all three methods.

Note: The listener interface must be started first before any listener is started. Listener transactions should be started only via transaction EZAO or via program EZACIC20.

Starting/Stopping CICS Listener Support Automatically

You can start and stop the CICS Listener Support automatically by modifying the CICS Program List Table (PLT).

- Startup (PLTPI)

To start the CICS Listener Support automatically, make the following entry in the PLTPI **after** the DFHDELIM entry:

```
DFHPLT      TYPE=ENTRY, PROGRAM=EZACIC20
```

- Shutdown (PLTSD)

To shut down the CICS Listener Support automatically, make the following entry in the PLTSD **before** the DFHDELIM entry:

```
DFHPLT      TYPE=ENTRY, PROGRAM=EZACIC20
```

Starting/Stopping CICS Manually

You can start CICS Listener Support manually by using the EZAO transaction. This operational transaction has four functions:

CICS Listener Support Startup

Starts the CICS Listener Support in a CICS address space and starts all listeners which are identified for immediate start.

Note: The EZAO transaction **must** be running on the CICS where you want to start the CICS Listener Support. You may not start a CICS Listener Support from a different CICS.

CICS Listener Support Shutdown

Stops the interface in a CICS address space.

Listener Startup

Starts a Listener in a CICS address space.

Starting and Stopping the CICS Listener Support

Listener Shutdown

Stops a Listener in a CICS address space.

If you enter EZAO, the following screen displays.

```
EZAO                                     APPLID=DBDCCICS
ENTER ONE OF THE FOLLOWING
STArT
STOp

PF 1 HELP      3 END      6 CRSR      9 MSG      12 CNCL
```

START Function

The START function starts either the CICS Listener Support or a single Listener.

When the CICS Listener Support is started, all Listeners marked for immediate start will be started as well. If you enter STA on the previous screen or enter EZAO STA on a blank screen, the following screen displays.

```
EZAO START                               APPLID=DBDCCICS
ENTER ONE OF THE FOLLOWING

CICS      ==> ...      Enter Yes|No
LIStener  ==> ...      Enter Yes|No

PF 1 HELP      3 END      6 CRSR      9 MSG      12 CNCL
```

START CICS Listener Support

If you enter EZAO START CICS, the following screen displays.

```
EZAO START CICS                                APPLID=DBDCCICS

CICS      ==> APPLID          APPLID of CICS

RESULT MESSAGE APPEARS HERE

PF 1 HELP      3 END          6 CRSR          9 MSG          12 CNCL
```

START A LISTENER

If you enter EZAO START LISTENER, the following screen displays.

```
EZAO START LISTENER                            APPLID=DBDCCICS
CICS      ==> APPLID          APPLID of CICS
NAME      ==>                Enter Name of Listener

PF          3 END          9 MSG          12 CNCL
```

After you enter the listener name, the listener is started. The following screen displays; the results appear in the message area.

Starting and Stopping the CICS Listener Support

```
EZAO START LISTENER                                APPLID=DBDCCICS

CICS      ==> APPLID      APPLID of CICS system
NAME      ==> XXXX       Transaction Name of Listener

RESULT MESSAGE APPEARS HERE

PF          3 END                      9 MSG          12 CNCL
```

STOP Function

The STOP function is used to stop either the CICS Listener Support or a single Listener within the interface.

If the CICS Listener Support is stopped, all Listeners will be stopped before the CICS Listener Support is stopped. If you enter STO on the previous screen or enter EZAO STO on a blank screen, the following screen will be displayed:

```
EZAO STOP                                APPLID=DBDCCICS
ENTER ONE OF THE FOLLOWING

CICS      ==> ...           Enter Yes|No
LISstener ==> ...           Enter Yes|No

PF 1 HELP      3 END          6 CRSR          9 MSG          12 CNCL
```


STOP CICS Listener Support

If you specify EZAO STOP CICS, the following screen is displayed

```
EZAO STOP CICS                                     APPLID=DBDCCICS

CICS          ===> ...                               APPLID of CICS
IMMEDIATE     ===> ...                               Enter Yes|No

PF 1 HELP      3 END          6 CRSR          9 MSG          12 CNCL
```

Two options are available to stop CICS Listener Support:

IMMEDIATE=NO

This should be used in most cases, because it causes the controlled termination of the CICS Listener Support. It has the following effects on applications using this API:

- The Listener transaction (EZAL) quiesces after a maximum wait of 3 minutes provided that no other socket applications are active or suspended.
- If there are active or suspended sockets applications, the Listener will allow them to continue processing. When all of these tasks are completed, then the Listener terminates.
- New listeners cannot be started.

IMMEDIATE=YES

This option is reserved for unusual situations and causes the abrupt termination of the CICS Listener Support. It has the following effect on applications using this API:

- It force purges the master server (Listener) EZAL.

After you choose an option, the stop will be attempted. The screen re-displays; the results appear in the message line.

Starting and Stopping the CICS Listener Support

STOP A LISTENER

If you specify STOP LISTENER, the following screen displays.

```
EZAO STOP LISTENER                                APPLID=DBDCCICS
CICS      ==> DBDCCICS                            APPLID of this CICS
LIStener  ==> .....                               Enter Name of Listener

PF 1 HELP      3 END          6 CRSR          9 MSG          12 CNCL
```

If you enter the listener named, that listener will be stopped. The screen re-displays; the results appear in the message line.

Starting/Stopping CICS Listener Support with Program Link

You can start or stop the CICS TCP/IP Listener Support by issuing an EXEC CICS LINK to program EZACIC20. Make sure you include the following steps in the LINKing program:

1. Define the COMMAREA for EZACIC20. This can be done by including the following instruction within your DFHEISTG definition:

```
EZACICA AREA=P20,TYPE=CSECT
```

The length of the area is equated to P20PARML and the name of the structure is P20PARMS.

2. Initialize the COMMAREA values as follows:

P20TYPE

```
      I      Initialization
      T      Immediate Termination
      D      Deferred Termination
```

P20OBJ

```
      C      CICS Listener Support
      L      Listener
```

P20LIST

Name of listener if this is listener initialization/termination.

3. Issue the EXEC CICS LINK to program EZACIC20. EZACIC20 *will not* return until the function is complete.
4. Check the P20RET field for the response from EZACIC20.

Starting and Stopping the CICS Listener Support

Note: The following user abend codes might be issued by EZACIC20:

- E20L is issued if the CICS Listener Support is not in startup or termination and no COMMAREA was provided.
- E20T is issued if CICS Listener Support is not active.

Starting and Stopping the CICS Listener Support

Chapter 19. Writing Your Own Listener

Basic Requirements

The CICS Listener Support provides a structure which supports up to 255 listeners. These listeners can be multiple copies of the IBM-supplied listener, user-written listeners, or a combination of the two. You can choose to run without a listener as well.

For each listener (IBM-Supplied or user-written), there are certain basic requirements that enable the interface to manage the listeners correctly; particularly during initialization and termination. They are:

- Each listener instance must have a unique transaction name, even if you are running multiple copies of the same listener.
- Each listener should have an entry in the CICS Listener Configuration Dataset. Even if you don't use automatic initiation for your listener, the lack of an entry would prevent correct termination processing and could prevent CICS from completing a normal shutdown.

For information on the IBM-supplied Listener, see “The Listener” on page 502.

Prerequisites

Some installations may require a customized, user-written listener. Writing your own listener has the following prerequisites:

1. Determine what capability is required which is not supplied by the IBM-supplied listener. Is this capability a part of the listener or a part of the server?
2. Knowledge of the CICS-Assembler environment is required.
3. Knowledge of multithreading applications is required. A listener must be able to perform multiple functions concurrently to achieve good performance.
4. Knowledge of the CICS Listener Interface is required.

Using IBM's Environmental Support

A user-written listener can use the environmental support supplied and used by the IBM-Supplied Listener. To employ this support, the user-written listener must do the following in addition to the requirements described above (a detailed description of the referenced storage areas is given in Chapter 20, “External Data Structures,” on page 487):

- The user-written listener must be written in Assembler.
- The RDO definitions for the listener transaction and program should be identical to those for the IBM-supplied listener with the exception of the transaction/program names.
- In the program, define an input area for configuration file records. If you are going to read the configuration file using MOVE mode you can define the area as by making the following entry in your DFHEISTG area:

```
EZACICA AREA=CONFIG,TYPE=CSECT
```

Writing Your Own Listener

If you are going to read the configuration file using LOCATE mode, you can define a DSECT for the area as follows:

```
EZACICA AREA=CONFIG,TYPE=DSECT
```

In either case, the length of the area is represented by the EQUATE label CFGLEN. The name of the area/DSECT is CFG0000.

- The CICS TCP/IP Listener Support does not require any LE runtime environment. Therefore, the listener program can be written as an assembler main routine.
- In the program, define a DSECT for mapping the Global Work Area (GWA). This is done by issuing the following macro:

```
EZACICA AREA=GWA,TYPE=DSECT
```

The name of the DSECT is GWA0000.

- In the program define a DSECT for mapping the Listener Control Area (LCA). This is done by issuing the following macro:

```
EZACICA AREA=LCA,TYPE=DSECT
```

The name of the DSECT is LCA0000.

- Obtain address of the Global Work Area (GWA). This can be done using the following CICS command:

```
EXEC CICS EXTRACT EXIT PROGRAM(EZACIC01) GASET(ptr) GALEN(len)
```

where *ptr* is a register and *len* is a halfword binary variable. The address of the GWA is returned in *ptr* and the length of the GWA is returned in *len*.

- Read the configuration file during initialization of the listener. The configuration file is identified as EZACONF in the CICS Configuration file. The record key for the user-written listener is as follows:

- APPLID

An 8 byte character field set to the APPLID value for this CICS. This value can be obtained from the field GWACAPPL in the GWA or using the following CICS command:

```
EXEC CICS ASSIGN APPLID(applid)
```

where *applid* is an 8 byte character field.

- Record Type

A 1 byte character field set to the record type. It must have the value 'L'.

- Reserved Field

A 3 byte hex field set to binary zeros.

- Transaction

A 4 byte character field containing the transaction name for this listener. It can be obtained from the EIBTRNID field in the Execute Interface Block.

The configuration record provides the information entered via either the configuration macro or the EZAC transaction. The user-written listener may use this information selectively but it is highly recommended it uses the port, backlog and number of sockets data.

For shared files: If the user-written listener reads the configuration file, it must first issue an EXEC CICS SET command to enable and open the file. When the

file operation is complete, the user-written listener must issue an EXEC CICS SET command to disable and close the file. Failure to do so will result in file errors in certain shared-file situations.

- The user-written listener should locate its Listener Control Area (LCA). The LCAs are located contiguously in storage with the first one pointed to by the GWALCAAD field in the GWA. The correct LCA has the transaction name of the listener in the field LCATRAN.
- The user-written listener should monitor either the LCASTAT field in the LCA or the GWATSTAT field in the GWA for shutdown status. If either field shows an immediate shutdown in progress, the user-written listener should terminate by issuing an EXEC CICS RETURN and allow the interface to clean up any socket connections. If either field shows a deferred termination in progress, the user-written listener should do the following:
 1. Accept any pending connections and then close the passive (listen) socket.
 2. Complete processing of any sockets involved in transaction initiation, i.e. processing the GIVESOCKET command. When processing is complete, close these sockets.
 3. When all sockets are closed, issue an EXEC CICS RETURN.
- The user-written listener should avoid socket calls which imply blocks dependent on external events such as ACCEPT or READ. These calls should be preceded by a single SELECTEX call which waits on the ECB LCATECB in the LCA. This ECB is posted when an immediate termination is detected and its posting will cause the SELECTEX to complete with a RETCODE of 0 and an ERRNO of 0. The program should check the ECB when the SELECTEX completes in this way as this is identical to the way SELECTEX completes when a timeout happens. The ECB may be checked by looking for a X'80' in the third byte (post bit).

This SELECTEX should specify a timeout value. This provides the listener with a way to periodically check for a deferred termination request. Without this, CICS Listener Deferred Termination or CICS Deferred Termination cannot complete.

Writing Your Own Listener

Chapter 20. External Data Structures

External Data Structures

The data structures available for customer use are as follows:

Configuration Data Set Record Formats

DSECT/Structure Name

CFG0000

Length of Structure

CFGLEN

Macro Expansion

EZACICA AREA=CONFIG,TYPE=DSECT

EZACICA AREA=CONFIG,TYPE=CSECT

Table 14. Configuration File Format

Field Name	Field Type	Description	Default Value												
CFHAPPL	8 byte char	APPLID of CICS object to which this record refers													
CFHRTYPE	1 byte char	Record type <ul style="list-style-type: none"> • C = CICS object record • L = Listener object record 													
	3 bytes	Reserved	00												
<i>CICS Record Format</i>															
CFCTRAN	4 bytes bin	Binary zeros	00000000												
CFCTCPIP	8 bytes char	Address space name for TCP/IP	SOCKET00												
CFCNOTSK	2 bytes bin	Number of reusable tasks ⁽¹⁾	20												
CFCSTIME	2 bytes bin	Cache minimum refresh time	15												
CFCLTIME	2 bytes bin	Cache maximum refresh time	30												
CFCNORES	2 bytes bin	Cache number of concurrent resolvers	10												
CFCDPRTY	2 bytes bin	Limit priority of subtask ⁽¹⁾	0												
CFCENAME	4 bytes char	Name of TD error queue	EZAM												
CFCOPT	1 byte bin	CICS options <table style="margin-left: 20px; border: none;"> <thead> <tr> <th>Value</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>B'00000001'</td> <td>Suppress task standard messages</td> </tr> <tr> <td>B'00000010'</td> <td>Suppress trace ⁽¹⁾</td> </tr> <tr> <td>B'00000100'</td> <td>Use OTE ⁽¹⁾</td> </tr> <tr> <td>B'00001000'</td> <td>PLT immediate shutdown</td> </tr> <tr> <td>B'00010000'</td> <td>Register application data ⁽¹⁾</td> </tr> </tbody> </table>	Value	Meaning	B'00000001'	Suppress task standard messages	B'00000010'	Suppress trace ⁽¹⁾	B'00000100'	Use OTE ⁽¹⁾	B'00001000'	PLT immediate shutdown	B'00010000'	Register application data ⁽¹⁾	
Value	Meaning														
B'00000001'	Suppress task standard messages														
B'00000010'	Suppress trace ⁽¹⁾														
B'00000100'	Use OTE ⁽¹⁾														
B'00001000'	PLT immediate shutdown														
B'00010000'	Register application data ⁽¹⁾														
	1 byte	Reserved													
CFCTERML	2 bytes bin	Subtask termination limit ⁽¹⁾													
CFCTCBLM	4 bytes bin	OPEN API TCP limit ⁽¹⁾													
<i>Listener Record Format</i>															
CFLTRAN	4 bytes char	Listener transaction name	EZAL												

External Data Structures

Table 14. Configuration File Format (continued)

Field Name	Field Type	Description	Default Value																		
CFLPORT	2 bytes bin	Port number for the listener																			
CFLBKLOG	2 bytes bin	BACKLOG value for the listener	20																		
CFLNSOCK	2 bytes bin	Number of sockets used by the listener	50																		
CFLMMIN	2 bytes bin	Minimum length of input message	4																		
CFLLTIM	2 bytes bin	Timeout value (seconds) for ACCEPT	60																		
CFLRTIM	2 bytes bin	Timeout value (seconds) for READ	0																		
CFLGTIM	2 bytes bin	Timeout value (seconds) for GIVESOCKET	same as ACCEPT timeout value																		
CFLOPT	1 byte bin	Listener Options <table border="0"> <tr> <td>Value</td> <td>Meaning</td> </tr> <tr> <td>B'00000001'</td> <td>Immediate startup</td> </tr> <tr> <td>B'00000110'</td> <td>Translate entire message</td> </tr> <tr> <td>B'00000010'</td> <td>Translate transaction code only</td> </tr> <tr> <td>B'00000100'</td> <td>Translate user data only</td> </tr> <tr> <td>B'00010000'</td> <td>Peek at data only</td> </tr> <tr> <td>B'00100000'</td> <td>Outbound messages are in EBCDIC</td> </tr> <tr> <td>B'01000000'</td> <td>This is an Enhanced Listener</td> </tr> <tr> <td>B'10000000'</td> <td>Listeners AF is AF_INET6</td> </tr> </table>	Value	Meaning	B'00000001'	Immediate startup	B'00000110'	Translate entire message	B'00000010'	Translate transaction code only	B'00000100'	Translate user data only	B'00010000'	Peek at data only	B'00100000'	Outbound messages are in EBCDIC	B'01000000'	This is an Enhanced Listener	B'10000000'	Listeners AF is AF_INET6	B'00000111'
Value	Meaning																				
B'00000001'	Immediate startup																				
B'00000110'	Translate entire message																				
B'00000010'	Translate transaction code only																				
B'00000100'	Translate user data only																				
B'00010000'	Peek at data only																				
B'00100000'	Outbound messages are in EBCDIC																				
B'01000000'	This is an Enhanced Listener																				
B'10000000'	Listeners AF is AF_INET6																				
CFLSECXT	8 bytes char	Name of security exit																			
	36 bytes	Reserved																			
CFLCSTRN	4 bytes char	Child server TRANID																			
CFLCSSTT	2 bytes char	Child server startup type																			
CFLCSDLY	6 bytes char	Child server delay interval																			
	1 byte	Reserved																			
CFLMSGLN	2 bytes bin	Length of inbound message																			
CFLOPT2	1 byte bin	Listener options byte 2 ⁽¹⁾																			
CFLUSRID	8 bytes char	Listeners user ID ⁽¹⁾																			
	1 byte	Reserved																			
CFLRTYTIME	2 bytes bin	Stack connection retry time																			

⁽¹⁾ Not used with z/VSE.

Global Work Area

DSECT/Structure Name

GWA0000

Length of Structure

GWALENTH (Length of fixed area)

Macro Expansion

EZACICA AREA=GWA,TYPE=DSECT

EZACICA AREA=GWA,TYPE=CSECT

Table 15. Global Work Area Format

Field Name	Field Type	Description	Default Value
<i>Beginning of Global Work Area Eyecatcher</i>			
GWACMDSC	8 byte char	Identifier	EZACICGW
<i>Beginning of Startup Module Heritage</i>			
GWACMNAM	8 byte char	Startup module name	EZACIC21
	1 byte char	Delimiter	
GWACMVER	3 byte char	Startup service level	
	1 byte char	Delimiter	
GWACMREL	11 byte char	Startup module date or APAR	
	6 byte char	Reserved	
<i>End of Startup Module Heritage</i>			
<i>Beginning of Task-Related User Exit Heritage</i>			
GWATRNAM	8 byte char	Task related user exit module name	EZACIC01
GWATRVER	2 byte char	Task related user exit version number ⁽¹⁾	
GWATRREL	2 byte char	Task related user exit release number ⁽¹⁾	
GWATRMOD	2 byte char	Task related user exit mod number ⁽¹⁾	
GWATRDAT	8 byte char	Task related user exit assembled date ⁽¹⁾	
GWATRTIM	8 byte char	Task related user exit assembled time ⁽¹⁾	
<i>End of Task-Related User Exit Heritage</i>			
<i>Beginning of IBM Listener Heritage</i>			
GWAMSNAM	8 byte char	IBM Listener module name	EZACIC02
GWAMSVR	2 byte char	IBM Listener version number	
GWAMSREL	2 byte char	IBM Listener release number	
GWAMSMOD	2 byte char	IBM Listener mod number	
GWAMSDAT	8 byte char	IBM Listener assembled date	
GWAMSTIM	8 byte char	IBM Listener assembled time	
<i>End of IBM Listener Heritage</i>			
<i>Beginning of Attached Subtask Heritage</i>			
GWASTNAM	8 byte char	Attached subtask module name ⁽¹⁾	
GWASTVER	2 byte char	Attached subtask version number ⁽¹⁾	
GWASTREL	2 byte char	Attached subtask release number ⁽¹⁾	
GWASTMOD	2 byte char	Attached subtask mod number ⁽¹⁾	
GWASTDAT	8 byte char	Attached subtask assembled date ⁽¹⁾	
GWASTTIM	8 byte char	Attached subtask assembled time ⁽¹⁾	
<i>End of Attached Subtask Heritage</i>			
GWACMIBM	154 byte char	Copyright statement	
GWAENQTID	4 byte char	Transaction ID enqueueing GWA ⁽¹⁾	
GWAENQTNO	4 byte bin	Task number of GWAENQTID ⁽¹⁾	
	2 byte	Reserved	
GWAC01TR	4 byte bin	TRU00000 address of EZACIC01 ⁽¹⁾	
GWANCHND	4 byte bin	Number entries on LCA chain ⁽¹⁾	

External Data Structures

Table 15. Global Work Area Format (continued)

Field Name	Field Type	Description	Default Value												
GWATCBLM	4 byte bin	TCB limit for OTE ⁽¹⁾													
GWAOATNO	4 byte bin	OPEN API TCB number active ⁽¹⁾													
GWAOATQD	4 byte bin	TCBLIM queue depth ⁽¹⁾													
GWAOATHW	4 byte bin	Suspend task high water mark ⁽¹⁾													
GWAOPT3	1 byte	Unused option flag													
GWAOPT2	1 byte	Unused option flag													
GWACTL	1 byte	TCBLIM flag ⁽¹⁾													
GWAOPT	1 byte	<table border="0"> <thead> <tr> <th>Value</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>B'00000001'</td> <td>Suppress task standard messages</td> </tr> <tr> <td>B'00000010'</td> <td>Suppress trace ⁽¹⁾</td> </tr> <tr> <td>B'00000100'</td> <td>Use OTE ⁽¹⁾</td> </tr> <tr> <td>B'00001000'</td> <td>PLT immediate shutdown</td> </tr> <tr> <td>B'00010000'</td> <td>Register application data ⁽¹⁾</td> </tr> </tbody> </table>	Value	Meaning	B'00000001'	Suppress task standard messages	B'00000010'	Suppress trace ⁽¹⁾	B'00000100'	Use OTE ⁽¹⁾	B'00001000'	PLT immediate shutdown	B'00010000'	Register application data ⁽¹⁾	
Value	Meaning														
B'00000001'	Suppress task standard messages														
B'00000010'	Suppress trace ⁽¹⁾														
B'00000100'	Use OTE ⁽¹⁾														
B'00001000'	PLT immediate shutdown														
B'00010000'	Register application data ⁽¹⁾														
GWASTGPR	4 byte bin	CICS storage protector indicator													
GWAUSCNT	4 byte bin	Use count for this GWA													
GWABKWRD	4 byte bin	Task chain anchor backward address ⁽¹⁾													
GWAFOWRD	4 byte bin	Task chain anchor forward address ⁽¹⁾													
GWACAPPL	8 byte char	VTAM APPLID of the CICS system													
GWATRUEN	8 byte char	Name of task related user exit load module													
GWASTSKN	8 byte char	Name of attached subtask load module ⁽¹⁾													
GWATCPID	8 byte char	TCP/IP address space name	SOCKET00												
GWALCAAD	4 byte bin	Address of first Listener control area													
GWA03PSA	4 byte bin	Address of EZASOH03 load module ⁽¹⁾													
GWANTASK	2 byte bin	Number of reusable tasks ⁽¹⁾													
GWANLIST	2 byte bin	Number of Listeners													
GWATSTAT	1 byte char	<p>Task related user exit status</p> <table border="0"> <thead> <tr> <th>Value</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>E</td> <td>TRUE is enabled</td> </tr> <tr> <td>I</td> <td>Immediate shutdown requested/processing</td> </tr> <tr> <td>Q</td> <td>Quiescent shutdown requested/processing</td> </tr> <tr> <td>T</td> <td>TRUE is requesting a shutdown</td> </tr> </tbody> </table>	Value	Meaning	E	TRUE is enabled	I	Immediate shutdown requested/processing	Q	Quiescent shutdown requested/processing	T	TRUE is requesting a shutdown			
Value	Meaning														
E	TRUE is enabled														
I	Immediate shutdown requested/processing														
Q	Quiescent shutdown requested/processing														
T	TRUE is requesting a shutdown														
GWARSHUT	1 byte char	<p>EZAO shutdown request indicator</p> <table border="0"> <thead> <tr> <th>Value</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>I</td> <td>Immediate shutdown requested</td> </tr> <tr> <td>Q</td> <td>Quiescent shutdown requested</td> </tr> </tbody> </table>	Value	Meaning	I	Immediate shutdown requested	Q	Quiescent shutdown requested							
Value	Meaning														
I	Immediate shutdown requested														
Q	Quiescent shutdown requested														
GWACSTART	1 byte bin	CICS execution Status ⁽¹⁾													
GWAVOSYS	1 byte bin	MVS version ⁽¹⁾													
GWAOPREL	2 byte bin	MVS release ⁽¹⁾													
GWACIVER	2 byte char	CICS version number ⁽¹⁾													
GWACIREL	1 byte char	CICS release number ⁽¹⁾													
GWACIMOD	1 byte char	CICS modification number ⁽¹⁾													
GWATOKEN	8 byte char	Token for OS/390 registration/deregistration ⁽¹⁾													

Table 15. Global Work Area Format (continued)

Field Name	Field Type	Description	Default Value
GWAMSGMD	8 byte char	Name of message module	EZACICMx
GWATDMSG	4 byte char	TD queue for messages	
GWACSTIM	2 byte bin	Cache refresh short time	
GWACLTIM	2 byte bin	Cache refresh long time	
GWACNRES	2 byte bin	Max number of resolvers	
GWAC01EP	4 byte bin	Entry point address of EZACIC01 ⁽¹⁾	
GWAC02EP	4 byte bin	Entry point address of EZACIC02 ⁽¹⁾	
GWALCACH	4 byte bin	Address of first chained LCA ⁽¹⁾	
GWATERML	2 byte bin	Subtask termination limit ⁽¹⁾	
GWALCACT	2 byte bin	Number of active LCAs	
GWAMONST	4 byte bin	TRUEs EMP disabled status ⁽¹⁾	
GWAMONSTL	4 byte bin	Listeners EMP disabled status ⁽¹⁾	
GWAMONCT	16 byte bin	Counters for system termination EMP ⁽¹⁾	
End of Fixed Part of GWA			

⁽¹⁾ Not used within z/VSE

Parameter List (COMMAREA) for EZACIC20

DSECT/Structure Name
P20PARMS

Length of Structure
P20PARML

Macro Expansion
EZACICA AREA=P20,TYPE=DSECT
EZACICA AREA=P20,TYPE=CSECT

Table 16. COMMAREA Format for EZACIC20

Field Name	Field Type	Description	Default Value										
P20TYPE	1 byte char	Type of function <table> <thead> <tr> <th>Value</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>I</td> <td>Initialization</td> </tr> <tr> <td>T</td> <td>Immediate termination</td> </tr> <tr> <td>D</td> <td>Deferred termination</td> </tr> <tr> <td>Q</td> <td>QUERY PLTSDI</td> </tr> </tbody> </table>	Value	Meaning	I	Initialization	T	Immediate termination	D	Deferred termination	Q	QUERY PLTSDI	
Value	Meaning												
I	Initialization												
T	Immediate termination												
D	Deferred termination												
Q	QUERY PLTSDI												
P20OBJ	1 byte char	Initialization / termination object <table> <thead> <tr> <th>Value</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>C</td> <td>CICS initialization / termination</td> </tr> <tr> <td>L</td> <td>Listener initialization / termination</td> </tr> </tbody> </table>	Value	Meaning	C	CICS initialization / termination	L	Listener initialization / termination					
Value	Meaning												
C	CICS initialization / termination												
L	Listener initialization / termination												
P20LIST	4 byte char	Transaction name of Listener											

External Data Structures

Table 16. COMMAREA Format for EZACIC20 (continued)

Field Name	Field Type	Description	Default Value																
P20RET	1 byte bin	Return flags <table border="0"> <thead> <tr> <th>Value</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>B'00000000</td> <td>No errors encountered'</td> </tr> <tr> <td>B'00000001'</td> <td>Error in CICS sockets I/F initialization</td> </tr> <tr> <td>B'00000010'</td> <td>Error in Listener initialization</td> </tr> <tr> <td>B'00000100'</td> <td>Error in CICS sockets I/F termination</td> </tr> <tr> <td>B'00001000'</td> <td>Error in Listener termination</td> </tr> <tr> <td>B'00010000'</td> <td>Error in COMMAREA contents</td> </tr> <tr> <td>B'00100000'</td> <td>Errors in CICS/TS for VSE</td> </tr> </tbody> </table>	Value	Meaning	B'00000000	No errors encountered'	B'00000001'	Error in CICS sockets I/F initialization	B'00000010'	Error in Listener initialization	B'00000100'	Error in CICS sockets I/F termination	B'00001000'	Error in Listener termination	B'00010000'	Error in COMMAREA contents	B'00100000'	Errors in CICS/TS for VSE	
Value	Meaning																		
B'00000000	No errors encountered'																		
B'00000001'	Error in CICS sockets I/F initialization																		
B'00000010'	Error in Listener initialization																		
B'00000100'	Error in CICS sockets I/F termination																		
B'00001000'	Error in Listener termination																		
B'00010000'	Error in COMMAREA contents																		
B'00100000'	Errors in CICS/TS for VSE																		

Listener Control Area (LCA)

DSECT/Structure Name

LCA0000

Length of Structure

LCALEN

Macro Expansion

EZACICA AREA=LCA,TYPE=DSECT

EZACICA AREA=LCA,TYPE=CSECT

Table 17. Listener Control Area (LCA)

Field Name	Field Type	Description	Default Value																				
LCATECB	Fullword bin	ECB posted by termination manager																					
LCATRAN	4 byte char	Transaction name for this Listener																					
LCASTAT	1 byte bin	Status of this Listener, byte 1 <table border="0"> <thead> <tr> <th>Value</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>B'00000000</td> <td>Listener not in operation'</td> </tr> <tr> <td>B'00000001'</td> <td>Listener in initialization</td> </tr> <tr> <td>B'00000010'</td> <td>Listener in SELECT</td> </tr> <tr> <td>B'00000100'</td> <td>Listener processing</td> </tr> <tr> <td>B'00001000'</td> <td>Listener had initialization error</td> </tr> <tr> <td>B'00010000'</td> <td>Immediate termination in progress</td> </tr> <tr> <td>B'00100000'</td> <td>Deferred termination in progress</td> </tr> <tr> <td>B'01000000'</td> <td>Listener is active</td> </tr> <tr> <td>B'10000000'</td> <td>Listener is CICS delayed retry</td> </tr> </tbody> </table>	Value	Meaning	B'00000000	Listener not in operation'	B'00000001'	Listener in initialization	B'00000010'	Listener in SELECT	B'00000100'	Listener processing	B'00001000'	Listener had initialization error	B'00010000'	Immediate termination in progress	B'00100000'	Deferred termination in progress	B'01000000'	Listener is active	B'10000000'	Listener is CICS delayed retry	
Value	Meaning																						
B'00000000	Listener not in operation'																						
B'00000001'	Listener in initialization																						
B'00000010'	Listener in SELECT																						
B'00000100'	Listener processing																						
B'00001000'	Listener had initialization error																						
B'00010000'	Immediate termination in progress																						
B'00100000'	Deferred termination in progress																						
B'01000000'	Listener is active																						
B'10000000'	Listener is CICS delayed retry																						
LCASTAT2	1 byte bin	Status of this Listener, byte 2 <table border="0"> <thead> <tr> <th>Value</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>B'00000001'</td> <td>Listener can now connect to TCP/IP</td> </tr> <tr> <td>B'00000010'</td> <td>Register application data ⁽¹⁾</td> </tr> <tr> <td>B'00000100'</td> <td>LAPPLD inherits APPLDAT ⁽¹⁾</td> </tr> <tr> <td>B'00100000'</td> <td>This is a standard Listener</td> </tr> <tr> <td>B'01000000'</td> <td>This is an enhanced Listener</td> </tr> <tr> <td>B'10000000'</td> <td>Listeners AF is AF_INET6</td> </tr> </tbody> </table>	Value	Meaning	B'00000001'	Listener can now connect to TCP/IP	B'00000010'	Register application data ⁽¹⁾	B'00000100'	LAPPLD inherits APPLDAT ⁽¹⁾	B'00100000'	This is a standard Listener	B'01000000'	This is an enhanced Listener	B'10000000'	Listeners AF is AF_INET6							
Value	Meaning																						
B'00000001'	Listener can now connect to TCP/IP																						
B'00000010'	Register application data ⁽¹⁾																						
B'00000100'	LAPPLD inherits APPLDAT ⁽¹⁾																						
B'00100000'	This is a standard Listener																						
B'01000000'	This is an enhanced Listener																						
B'10000000'	Listeners AF is AF_INET6																						
	2 bytes	reserved																					
LCAIEAD	4 bytes bin	Address of Listener's TIE																					

Chapter 21. CICS Listener Programming Considerations

Overview

This section describes typical sequences of calls for client, concurrent server (with associated child server processes), and iterative server programs. The contents of the section are:

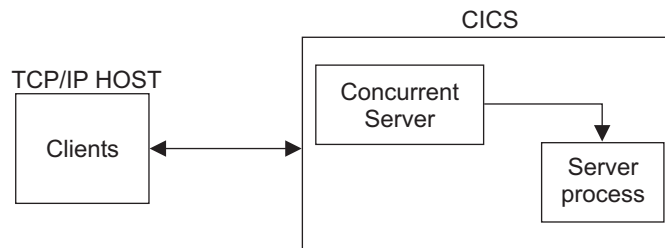
- Four setups for writing CICS TCP/IP applications:
 1. Concurrent server (the supplied Listener transaction) and child server processes run under CICS TCP/IP
 2. The same as 1 but with a user-written concurrent server
 3. An iterative server running under CICS TCP/IP
 4. A client application running under CICS TCP/IP
- Socket addresses
- GETCLIENTID, GIVESOCKET, and TAKESOCKET commands
- The Listener program

Writing CICS TCP/IP Applications

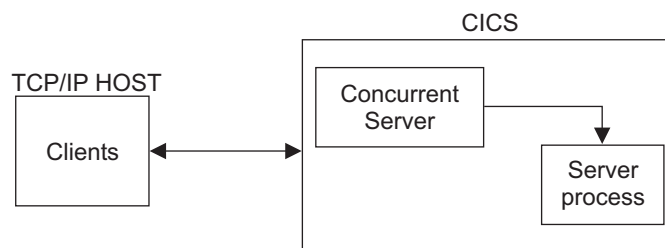
In this section, four TCP/IP setups in which CICS TCP/IP applications are used in various parts of the client/server system are discussed in detail.

The setups are:

1. **The Client-Listener-Child Server Application Set.** The concurrent server and child server processes run under CICS TCP/IP. The concurrent server is the supplied **Listener** transaction. The client might be running TCP/IP under the OS/2 operating system or one of the various UNIX operating systems such as AIX.

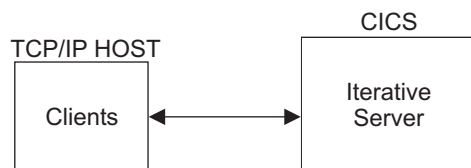


2. **Writing Your Own Concurrent Server.** This is the same setup as the first except that a user-written concurrent server is being used instead of the IBM Listener.

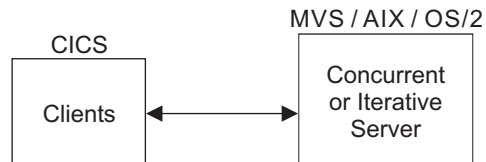


CICS Listener Programming Considerations

3. **The Iterative Server CICS TCP/IP Application.** This setup is designed to process one socket at a time.



4. **The Client CICS TCP/IP Application.** In this setup, the CICS application is the client and the server is the remote TCP/IP process.



1. The Client-Listener-Child-Server Application Set

Figure 43 on page 495 shows the sequence of CICS commands and socket calls involved in this setup. CICS commands are prefixed by EXEC CICS; all other numbered items in the figure are CICS TCP/IP calls.

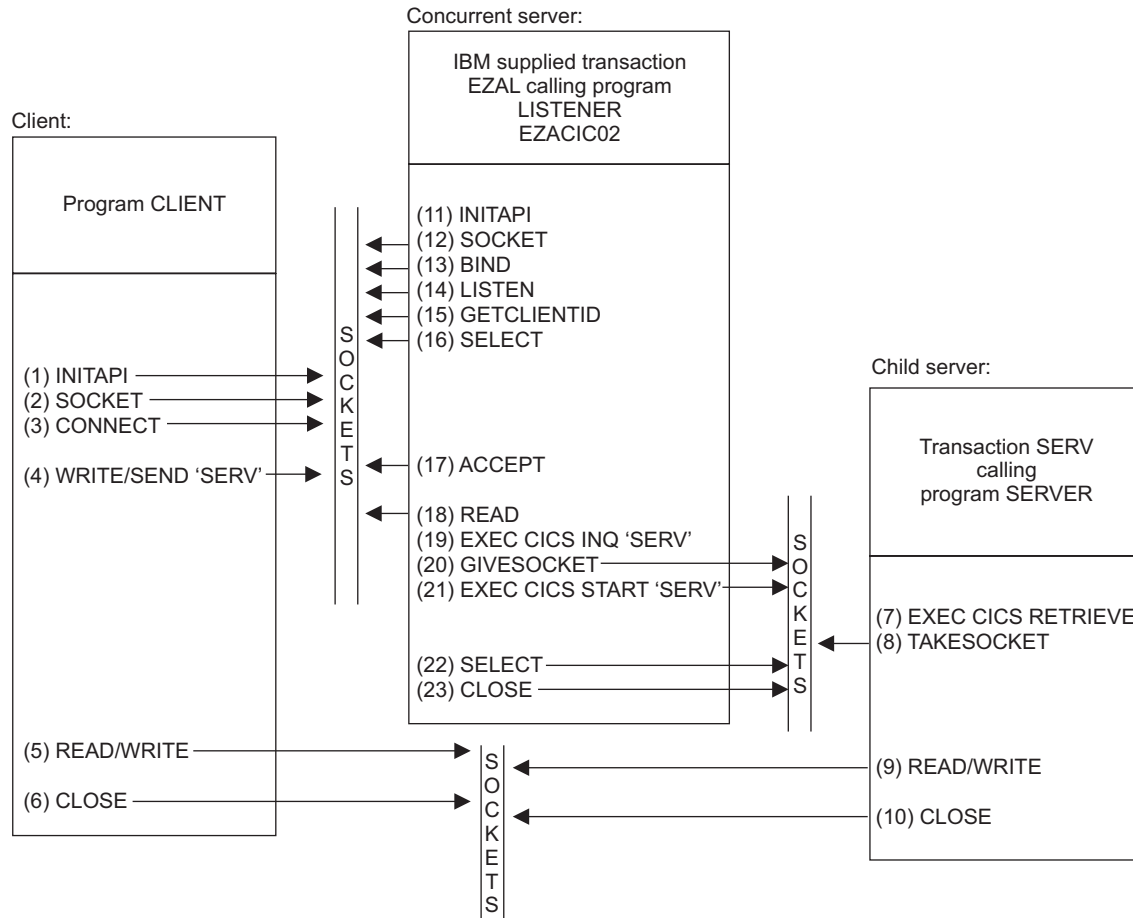


Figure 43. The Sequence of Sockets Calls

Client Call Sequence

Table 18 explains the functions of each of the calls listed in Figure 43.

Table 18. Calls for the Client Application

(1) INITAPI	Connect the CICS application to the TCP/IP interface. Use the MAX-SOC parameter to specify the maximum number of sockets to be used by the application.
(2) SOCKET	This obtains a socket. You define a socket with 3 parameters: <ul style="list-style-type: none"> • The domain, or addressing family • The type of socket • The protocol <p>For VSE TCP/IP, the domain can only be the TCP/IP internet domain (2 in COBOL, AF_INET in C). The type can be stream sockets (1 in COBOL, SOCK_STREAM in C), or datagram sockets (2 in COBOL, SOCK_DGRAM in C). The protocol can be either TCP or UDP. Passing 0 for the protocol selects the default protocol.</p> <p>If successful, the SOCKET call returns a socket descriptor, <i>s</i>, which is always a small integer. Notice that the socket obtained is not yet attached to any local or destination address.</p>

CICS Listener Programming Considerations

Table 18. Calls for the Client Application (continued)

(3) CONNECT	Client applications use this to establish a connection with a remote server. You must define the local socket s (obtained above) to be used in this connection and the address and port number of the remote socket. The system supplies the local address, so on successful return from CONNECT, the socket is completely defined, and is associated with a TCP connection (if stream) or UDP connection (if datagram).
(4) WRITE	This sends the first message to the Listener. The message contains the CICS transaction code as its first 4 bytes of data. You must also specify the buffer address and length of the data to be sent.
(5) READ/WRITE	These calls continue the conversation with the server until it is complete.
(6) CLOSE	This closes a specified socket and so ends the connection. The socket resources are released for other applications.

Listener Call Sequence

The Listener transaction EZAL is provided as part of CICS TCP/IP. These are the calls issued by the CICS Listener. Your client and server call sequences must be prepared to work with this sequence. These calls are documented in “2. Writing Your Own Concurrent Server,” where the Listener calls in Figure 43 on page 495 are explained.

Child Server Call Sequence

Table 19 explains the functions of each of the calls listed in Figure 43 on page 495.

Table 19. Calls for the Server Application

(7) EXEC CICS RETRIEVE	This retrieves the data passed by the EXEC CICS START command in the concurrent server program. This data includes the socket descriptor and the concurrent server client ID as well as optional additional data from the client.
(8) TAKESOCKET	This acquires the newly created socket from the concurrent server. The TAKESOCKET parameters must specify the socket descriptor to be acquired and the client id of the concurrent server. This information was obtained by the EXEC CICS RETRIEVE command. Note: If TAKESOCKET is the first call, it issues an implicit INITAPI with default values.
(9) READ/WRITE	The conversation with the client continues until complete.
(10) CLOSE	Terminates the connection and releases the socket resources when finished.

2. Writing Your Own Concurrent Server

The overall setup is the same as the first scenario, but your concurrent server application performs many of the functions performed by the Listener.

The client and child server applications have the same functions.

Concurrent Server Call Sequence

Table 20 on page 497 explains the functions of each of the steps listed in Figure 43 on page 495.

Table 20. Calls for the Concurrent Server Application

(11) INITAPI	Connects the application to TCP/IP, as in table Table 18 on page 495.
(12) SOCKET	This obtains a socket, as in table Table 18 on page 495.
(13) BIND	Once a socket has been obtained, a concurrent server uses this call to attach itself to a specific port at a specific address so that the clients can connect to it. The socket descriptor and a local address and port number are passed as arguments. On successful return of the BIND call, the socket is <i>bound</i> to a port at the local address, but not (yet) to any remote address.
(14) LISTEN	After binding an address to a socket, a concurrent server uses the LISTEN call to indicate its readiness to accept connections from clients. LISTEN tells TCP/IP that all incoming connection requests should be held in a queue until the concurrent server can deal with them. The BACKLOG parameter in this call sets the maximum queue size.
(15) GETCLIENTID	This command returns the identifiers (z/VSE partition name and subtask name) by which the concurrent server is known by TCP/IP. This information will be needed by the EXEC CICS START call.
(16) SELECT	The SELECT call monitors activity on a set of sockets. In this case, it is used to interrogate the queue (created by the LISTEN call) for connections. It will return when an incoming CONNECT call is received, or else will time out after an interval specified by one of the SELECT parameters.
(17) ACCEPT	The concurrent server uses this call to accept the first incoming connection request in the queue. ACCEPT obtains a new socket descriptor with the same properties as the original. The original socket remains available to accept more connection requests. The new socket is associated with the client that initiated the connection.
(18) READ	This reads one message from the client to determine what service is required. This message contains, at a minimum, the CICS transaction ID of the server.
(19) CICS INQ	This checks that the SERV transaction is defined to CICS (else the TRANSIDERR exceptional condition is raised), and, if so, that its status is ENABLED. If either check fails, the Listener does not attempt to start the SERV transaction.
(20) GIVESOCKET	This makes the socket obtained by the ACCEPT call available to a child server program.
(21) CICS START	This initiates the CICS transaction for the child server application and passes the ID of the concurrent server, obtained with GETCLIENTID, to the server. For example, in "Listener Output Format" on page 504, the parameter LSTN-CLIENTID defines the Listener.
(22) SELECT	Again, the SELECT call is used to monitor TCP/IP activity. This time, SELECT returns when the child server issues a TAKESOCKET call. This SELECT is the same as the SELECT call in Step 16. They are shown as two calls to clarify the functions being performed.
(23) CLOSE	This releases the new socket to avoid conflicts with the child server.

Passing Sockets

Sockets can be passed between programs within the same task, by passing the descriptor number. However, passing a socket between CICS tasks does require a

CICS Listener Programming Considerations

GIVESOCKET/TAKESOCKET sequence of calls.

3. The Iterative Server CICS TCP/IP Application

Figure 44 shows the sequence of socket calls involved in a simple client-iterative server setup.

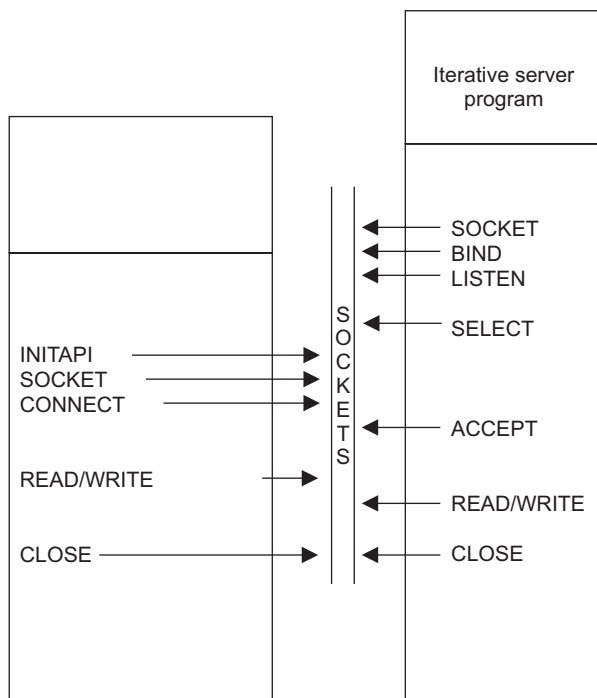


Figure 44. Sequence of Socket Calls with an Iterative Server

The setup with an iterative server is much simpler than the previous cases with concurrent servers.

Iterative Server Use of Sockets

The iterative server need only obtain 2 socket descriptors. The iterative server makes the following calls:

1. As with the concurrent servers, SOCKET, BIND, and LISTEN calls are made to inform TCP/IP that the server is ready for incoming requests, and is listening on socket 0.
2. The SELECT call then returns when a connection request is received. This prompts the issuing of an ACCEPT call.
3. The ACCEPT call obtains a new socket (1). Socket 1 is used to handle the transaction. Once this completed, socket 1 closes.
4. Control returns to the SELECT call, which then waits for the next connection request.

The disadvantage of an iterative server is that it remains blocked for the duration of a transaction.

4. The Client CICS TCP/IP Application

Figure 45 shows the sequence of calls in a CICS client-remote server setup. The calls are similar to the previous examples.

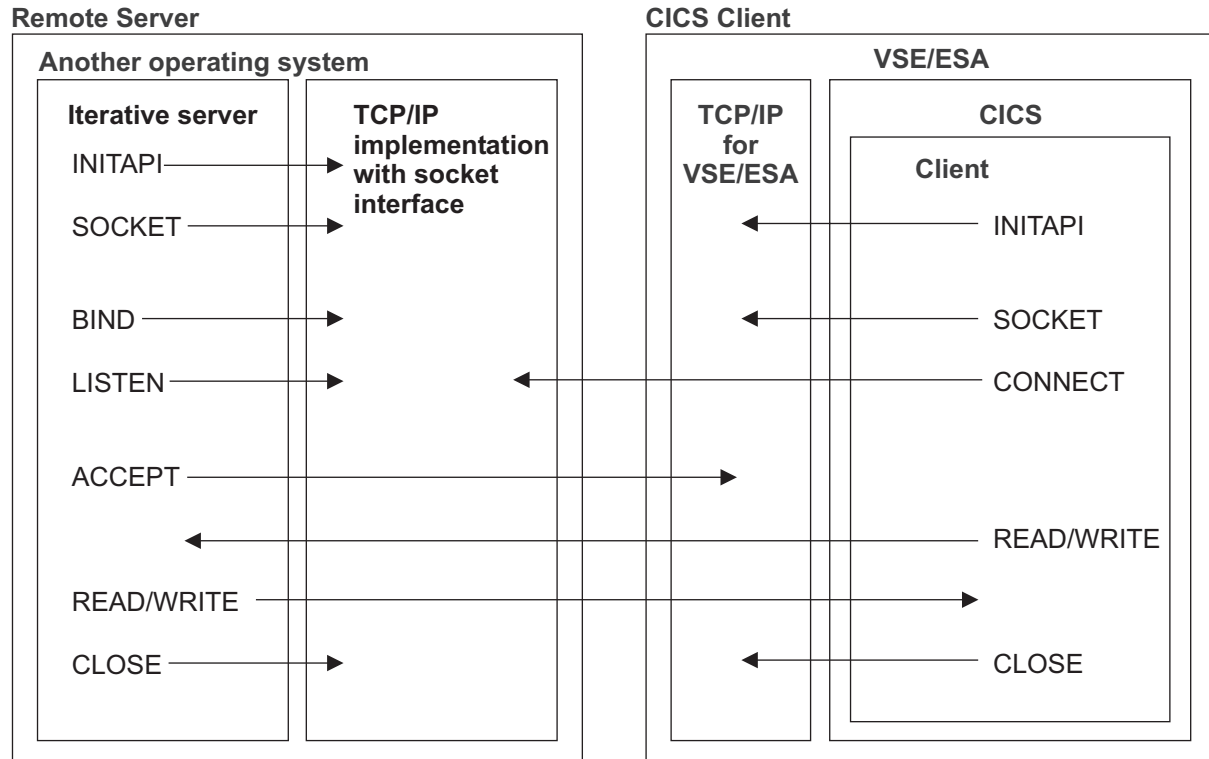


Figure 45. Sequence of Socket Calls between a CICS Client and a Remote Iterative Server

Figure 45 shows that the server can be on any processor and can run under any operating system, provided that the combined software-hardware configuration supports a TCP/IP server.

For simplicity, the figure shows an iterative server. A concurrent server would need a child server in the remote processor and an adjustment to the calls according to the model in Figure 43 on page 495.

A CICS server issues a READ call to read the client's first message, which contains the CICS transaction name of the required child server. When the server is in a non-CICS system, application design must specify how the first message from the CICS client indicates the service required (in Figure 45, the first message is sent by a WRITE call).

If the server is a concurrent server, this indication is typically the name of the child server. If the server is iterative as in Figure 45, and all client calls require the same service, this indication might not be necessary.

Socket Addresses

Socket addresses are defined by specifying the address family and the address of the socket in the internet.

CICS Listener Programming Considerations

In VSE TCP/IP, the address is specified by the IP address and port number of the socket.

Address Family (Domain)

VSE TCP/IP supports only one TCP/IP addressing family (or domain, as it is called in the UNIX system). This is the internet domain, denoted by AF_INET in C. Many of the socket calls require you to define the domain as one of their parameters.

A socket address is defined by the IP address of the socket and the port number allocated to the socket.

IP Addresses

IP addresses are allocated to each TCP/IP address on a TCP/IP internet. Each address is a unique 32-bit quantity defining the host's network and the particular host. A host can have more than one IP address if it is connected to more than one network (a so-called multi-homed host).

Ports

A host can maintain several TCP/IP connections at a time. One or more applications using TCP/IP on the same host are identified by a port number. The port number is an additional qualifier used by the system software to get data to the correct application. Port numbers are 16-bit integers; some numbers are reserved for particular applications and are called well-known ports (for example, 23 is for TELNET).

Address Structures

A socket address in an IP addressing family comprises 4 fields: the address family, an IP address, a port, and a character array (zeros), set as follows:

- The family field is set to AF_INET in C, or to 2 in other languages.
- The port field is the port used by the application, in network byte order (which is explained on page "Network Byte Order").
- The address field is the IP address of the network interface used by the application. It is also in network byte order.
- The character array field should always be set to all zeros.

Network Byte Order

Ports and addresses are specified using the TCP/IP network byte ordering convention, which is known as *big endian*.

In a big endian system, the most significant byte comes first. By contrast, in a *little endian* system, the least significant byte comes first. z/VSE uses the big endian convention; because this is the same as the network convention, CICS TCP/IP applications do not need to use any conversion routines, such as `htonl`, `htons`, `ntohl`, and `ntohs`.

Note: The socket interface does not handle differences in data byte ordering within application data. Sockets application writers must handle these differences themselves.

GETCLIENTID, GIVESOCKET, and TAKESOCKET

In CICS the socket calls GETCLIENTID, GIVESOCKET, and TAKESOCKET are used with the EXEC CICS START and EXEC CICS RETRIEVE commands to make a socket available to a new process. This is shown in Figure 46.

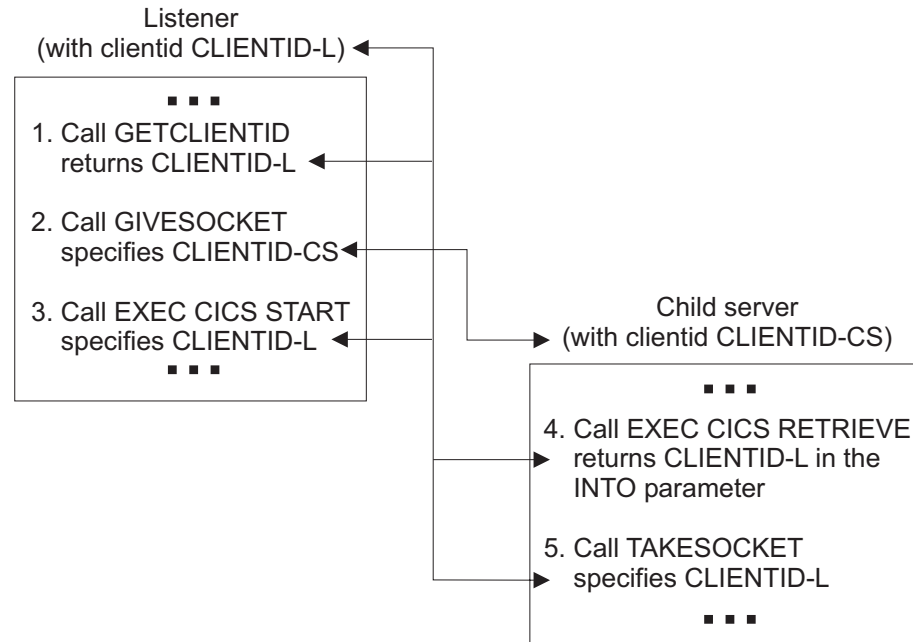


Figure 46. Transfer of CLIENTID Information

Figure 46 shows the calls used to make a Listener socket available to a child server process. It shows the following steps:

1. The Listener calls GETCLIENTID. This returns the Listener's own CLIENTID (CLIENTID-L), which comprises the z/VSE partition name and subtask identifier of the Listener. The Listener transaction needs access to its own CLIENTID for step 3.
2. The Listener calls GIVESOCKET, specifying a socket descriptor and the CLIENTID of the child server.

If the Listener and child server processes are in the same CICS region (and so in the same address space), the z/VSE partition name identifier in CLIENTID can be set to blanks. This means that the Listener's address space is also the child's address space.

If the Listener and child server processes are in different CICS regions, enter the new address space and subtask.

In the CLIENTID structure, the supplied listener enters its own z/VSE partition name and sets the subtask identifier to blanks. This makes the socket available to a TAKESOCKET command from any task in the Listener's address space, but only the child server receives the socket descriptor number, so the exposure is minimal. For total integrity, the child server's subtask identifier should be entered.

3. The Listener performs an EXEC CICS START. In the FROM parameter, the CLIENTID-L, obtained by the previous GETCLIENTID, is specified. The Listener is telling the new child server where it will get its socket from in step 5 on page 502.

CICS Listener Programming Considerations

4. The child server performs an EXEC CICS RETRIEVE. In the INTO parameter, CLIENTID-L is retrieved.
5. The child server calls TAKESOCKET, specifying CLIENTID-L as the process from which it wants to take a socket.

The Listener

In a CICS system based on SNA terminals, the CICS terminal management modules perform the functions of a concurrent server. Because the TCP/IP interface does not use CICS terminal management, CICS provides these functions in form of a CICS application transaction, the Listener. The CICS transaction ID of the Listener is EZAL.

The Listener performs the following functions:

1. It issues appropriate TCP/IP calls to “listen” on the port specified in the configuration file and waits for incoming connection requests issued by clients.
2. When an incoming connection request arrives, the Listener accepts it and obtains a new socket to pass to the CICS child server application program.
3. The standard Listener starts the CICS child server transaction based on information in the first message on the new connection. The format of this information is given in “Listener Input Format (Standard Listener only).” For the enhanced listener, the CICS child server transaction is started based on information in the TCP/IP CICS configuration file, EZACONF.
4. It waits for the child server transaction to take the new socket and then issues the close call. When this occurs, the receiving application assumes ownership of the socket and the Listener has no more interest in it.

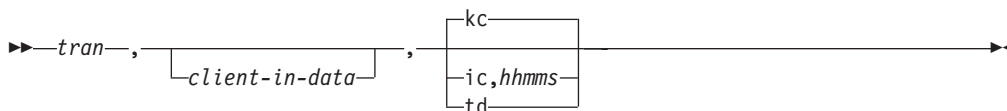
The Listener program is written so that some of this activity goes on in parallel. For example, while the program is waiting for a new server to accept a new socket, it listens for more incoming connections. The program can be in the process of starting 49 child servers simultaneously. The starting process begins when the Listener accepts the connection and ends when the Listener closes the socket it has given to the child server.

Listener Input Format (Standard Listener only)

The Listener requires the following input format from the client in its first transmission. The client should then wait for a response before sending any subsequent transmissions. Input can be in uppercase or lowercase. The commas are required.

Note: The Listener cannot distinguish between a comma that is used as a delimiter in the Listener's initial message and a comma that is part of the client-in-data format. Therefore, the client-in-data format should not contain a comma. In text such as x'2C' in ASCII data or such as '6B' in EBCDIC data, the single quotation mark might be interpreted as a comma.

Format



Parameters

tran

The CICS transaction ID (in uppercase) that the Listener is going to start. This field can be 1 - 4 characters.

client-in-data

This optional parameter specifies application data, which is used by the optional security exit (See "Writing Your Own Security Link Module for the Listener" on page 505) or the server transaction. The maximum length of this field is 40 characters (35 bytes, plus one byte filler and 4 bytes for startup).

kc|ic|td

This optional parameter specifies the startup type that can be either kc for CICS startup control, ic for CICS interval control or td for CICS transient data. They types are not case-sensitive. If this field is left blank, startup is immediate under CICS task control (kc) .kc indicates that the child server task is started using **EXEC CICS START** with no delay interval. This is the same as specifying **IC,000000**.

hhmms

If the transaction is started using interval control (ic), this parameter is mandatory. It specifies hours, minutes, and seconds for interval time. All 6 digits must be given.

Note: TD ignores the time field.

Examples

The following are examples of client input and the Listener processing that results from them. The data fields that are referenced can be found in "Listener Output Format" on page 504. Parameters are separated by commas.

Example	Listener Response
TRN1,userdataishere	It starts the CICS transaction TRN1 using task control, and passes to it the data userdataishere in the field CLIENT-IN-DATA.
TRN2,,IC,000003	It starts the CICS transaction TRN2 using interval control, without user data. There is a 3-second delay between the initiation request from the Listener and the transaction startup in CICS.
TRN3,userdataishere,TD	It writes a message to the transient data queue named TRN3 in the format that is described by the structure TCPSOCKET-PARM, described in "Listener Output Format" on page 504. The data that is contained in userdataishere is passed to the field CLIENT-IN-DATA. This queue must be an intrapartition queue with trigger-level set to 1. It causes the initiation of transaction TRN3 if it is not already active. This transaction should be written to read the transient data queue and process requests until the queue is empty. This mechanism is provided for those server transactions that are used frequently and for which the overhead of initiating a separate CICS transaction for each server request could be a performance concern.

CICS Listener Programming Considerations

Example	Listener Response
TRN3,,TD	It causes data to be placed on transient data queue TRN3, which in turn causes the start or continued processing of the CICS transaction TRN3, as described in the TRN3 previous example. There is no user data passed.
TRN4	It starts the CICS transaction TRN4 using task control. There is no user data that is passed to the new transaction.

Listener Output Format

Table 21 shows the format of the Listener output data area passed to the child server. This output data area has a total length of 96 bytes. The Listener program uses the following COBOL definition:

```

01 TCPSOCKET-PARM.
   05 GIVE-TAKE-SOCKET    PIC 9(8) COMP.
   05 LSTN-CLIENTID.
      15 LSTN-CID-DOMAIN  PIC 9(8) COMP.
      15 LSTN-CID-NAME    PIC X(8)
      15 LSTN-CID-TASK    PIC X(8)
      15 LSTN-CID-RSVD    PIC X(20)
   05 CLIENT-IN-DATA      PIC X(35).
   05 FILLER              PIC X(1).
   05 SOCKADDR-IN-PARM.
      15 SIN-FAMILY       PIC 9(4) COMP.
      15 SIN-PORT         PIC 9(4) COMP.
      15 SIN-ADDRESS      PIC 9(8) COMP.
      15 SIN-ZERO         PIC X(8).

```

Table 21. Listener Output Format - using the IPv4 protocol

Description	Format	Value
Socket descriptor	Fullword binary	The socket descriptor to be used by the child server in the TAKESOCKET command
Listener Client ID	40 bytes	Client ID of Listener
Data area	35-byte character plus 1-byte filler	Client-in-data from Listener input received from the client
Socket address	Structure containing remaining 4 fields	See each field
TCP/IP addressing family	Halfword binary	2, indicating AF-INET
Port descriptor	Halfword binary	Descriptor of the port bound to the socket (Listener's port number from the configuration file).
32-bit IP address	Fullword binary	IP address of the socket's host machine in network byte order
Unused	Doubleword	Binary zeros

Table 22. Output from EZA Listener to the Child Server - Enhanced mode - using the IPv6 protocol

Description	Format	Value
Socket descriptor	Fullword binary	Socket number to be specified in the TAKESOCKET command
Client ID	40 bytes	Client ID of Listener

Table 22. Output from EZA Listener to the Child Server - Enhanced mode - using the IPv6 protocol (continued)

Description	Format	Value
Client data	35 bytes	The first 35 bytes that is read by the Listener (if any)
Filler	1 byte	Filler
Client's SOCKADDR	2 bytes family 2 bytes port 4 bytes flow info 16 bytes IPv6 address 4 bytes scope	Flow info and scope are not used on z/VSE
	68 bytes	reserved
Data length	Halfword binary	Length of data received from the client
Data area	Length defined by previous field	Data received from the client starting at position 1

Writing Your Own Security Link Module for the Listener

The Listener process provides an exit point for those users who want to write and include a module that performs a security check before a CICS transaction is initiated. The exit point is implemented so that if a module is not provided, all valid transactions are initiated.

If you write a security module, you can name it anything you want, as long as you define it in the configuration dataset. You can write this program in COBOL, PL/I, or assembler language and must provide an appropriate entry in the CICS program processing table (PPT).

Specifying in EZAC: Specify the name of the security module in the SECexit field in Alter or Define. If you don't name the module, CICS will assume you don't have one. See 455 for more information.

Just before the task creation process, the Listener invokes the security module by a conditional CICS LINK passing a COMMAREA. The Listener passes a data area to the module that contains information for the module to use for security checking and a 1-byte switch. Your security module should perform a security check and set the switch accordingly.

When the security module returns, the Listener checks the state of the switch and initiates the transaction if the switch indicates security clearance. The module can perform any function that is valid in the CICS environment. Excessive processing, however, could cause performance degradation.

Table 23 shows the data area used by the security module.

Table 23. Output from EZA Listener to the Security / Transaction Exit

Description	Offset	Format	Value
CICS transaction ID	+000 (x00)	4 byte char	CICS transaction requested by the client
Data area	+004 (x04)	35 byte char	User data received by the client
Format switch	+039 (x27)	1 byte char	Format switch (1= expanded, 0= not expanded)

CICS Listener Programming Considerations

Table 23. Output from EZA Listener to the Security / Transaction Exit (continued)

Description	Offset	Format	Value
	+040 (x28)	4 bytes	Reserved
Action	+044 (x2C)	2 byte char	Method of starting task (IC= interval control, KC= task control, TD= transient data)
Interval control time	+046 (x2E)	6 byte char	Interval requested for IC start (hhmmss)
Address family	+052 (x34)	Halfword binary	Address family (2= AF_INET, 19= AF_INET6)
Client's port	+054 (x36)	Halfword binary	Port number of client's port
Client's IPv4 address	+056 (x38)	Fullword binary	IPv4 address of client
Switch	+060 (x3C)	1 byte char	Switch to be set by exit (1= permit transaction, not 1= prohibit transaction)
Switch-2	+061 (x3D)	1 byte char	Switch to be set by exit (1= Listener sends msg to client not 1= exit sends msg to client)
Terminal identification	+062 (x3E)	4 byte char	To be set by exit: Return binary zeros, if no terminal is to be associated with the new task. Otherwise return the CICS terminal ID to be associated with the new task.
Socket descriptor	*066 (x42)	Halfword binary	Current socket descriptor
User ID	+068 (x44)	8 byte char	A user ID value, which is used in starting the server transaction. This is mutually exclusive from terminal ID
Listener's IPv4 address	+076 (x4C)	Fullword binary	Local IPv4 address
Listener's port	+080 (x50)	Halfword binary	Listener's port number
Listener's IPv4 address	+082 (x52)	16 bytes binary	Local IPv6 address
Listener's scope ID	+098 (x62)	Fullword binary	Scope ID of requestor's IPv6 address (not used by VSE)
Client's IPv6 address	+102 (x66)	16 bytes binary	IPv6 address of requestor's host
Client's scope ID	+118 (x76)	Fullword binary	Scope ID of requestor's IPv6 address (not used by VSE)
Client's certificate length	+118 (x7A)	Halfword binary	Not supported by VSE
Client's certificate address	+124 (x7C)	Fullword binary	Not supported by VSE
	+128 (x80)	34 bytes	reserved
Data length	+162 (xA2)	Halfword binary	Length of data received from client

Table 23. Output from EZA Listener to the Security / Transaction Exit (continued)

Description	Offset	Format	Value
Data area	+164 (xA4)	Length defined by previous field	Data received from the client starting at position 1 (for the enhanced Listener the first 35 bytes are the same as in the first data area)

Data Conversion Routines

CICS uses the EBCDIC data format, whereas TCP/IP networks use ASCII. When moving data between CICS and the TCP/IP network, your application programs must initiate the necessary data conversion. CICS programs can use routines provided by z/VSE for:

- Converting data from EBCDIC to ASCII and back, when sending and receiving data to and from the TCP/IP network, with the SEND, RECEIVE, READ, and WRITE calls.
- Converting between bit arrays and character strings when using the SELECT call.

For details, refer to “Using Data Translation Programs for Socket Call Interface” on page 280.

Part 6. Appendixes

Appendix A. Examples to be used with TCP/IP for VSE/ESA

Autonomous FTP

Under normal circumstances, the VSE FTP Daemon performs all file transfers. For this reason, all files must be defined to the TCP/IP for VSE/ESA partition.

Under some circumstances, this can be inconvenient. For example, when a batch process creates a new file which is to be sent to a remote workstation, several interactions with TCP/IP for VSE/ESA are required to define the new file. Rather than force operator intervention in this manner, an extension to the FTP commands is provided that will permit specification of a locally-defined DLBL, without the TCP/IP partition having any advance knowledge of the data set. This operation mode can be considered as 'Autonomous FTP'.

To transfer a file in this mode, use a command of the following format:

```
PUT %dlbl,type,recfm,lrecl,blksize filespec
GET filespec %dlbl,type,recfm,lrecl,blksize
```

The percent sign (%) indicates that a DLBL has been supplied rather than a file name. The other parameters are as follows:

filespec

The file name on the remote system.

type The file's type.

recfm The file's record format.

lrecl The file's logical record length.

blksize

The file's block size.

A detailed discussion of all the parameters to be used for Autonomous FTP can be found in the *TCP/IP for VSE 1.5 User's Guide*.

Example

In the following example a SAM-ESDS working file 'A.KRUS.X1', ('X1' results from the partition-id) is defined indirectly, via IDCAMS REPRO. This file is transferred via Autonomous FTP to a workstation and after successfully processing it is deleted via IDCAMS. The advantage is that you don't have to define the actual file explicitly and to remember its file name.

```
* $$ JOB JNM=FTP AUTNP,CLASS=X,DISP=D
// JOB FTPAUTNP TEST AUTONOMIOUS FTP BATCH
// DLBL TESTNKD,'%A.KRUS',0,VSAM,CAT=ESCAT1,RECSIZE=120, X
// DLBL TESTNKD,DISP=(NEW,KEEP,DELETE),RECORDS=(150,100)
// DLBL TEST,'%A.KRUS',0,VSAM,CAT=ESCAT1
// DLBL COPYIN,'KRUS.SAMF',,VSAM,CAT=ESCAT1
// LOG
*
// EXEC IDCAMS,SIZE=AUTO
REPRO INFILE (COPYIN) -
      OUTFILE (TESTNKD ENV(BLKSZ(120) RECFM(F))) -
      NOREUSE
/*
*
```

Examples TCP/IP for VSE/ESA

```
// EXEC FTP,PARM='IP=KRUSE'  
KRUS  
DAGI  
DD  
DD  
LCD ESCAT1  
CD VSE230/TEMP  
PUT %TEST,SAM,F,120 FTTPUT.X1  
QUIT  
/*  
IF $RC > 0 THEN  
GOTO $EOJ  
// EXEC IDCAMS,SIZE=AUTO  
DELETE (%A.KRUS ) -  
    CLUSTER -  
    PURGE -  
    CATALOG (ESCAT1.USER.CATALOG )  
/*  
/&  
* $$ EOJ
```

Note:

1. In the job listing the workfile has a dynamic name, here
PUT %X1SAM,SAM,F,120 FTTPUT.X1
2. Following is not possible with autonomous FTP, but with the FTPBATCH program:
 - a DLBL statement with the DISP option
 - SAM-ESDS '%%working' files

AUTOLPR – Printing with the CICS Report Controller Feature (RCF)

This section shows an example how to use the CICS Report Controller Feature (RCF) along with the TCP/IP for VSE/ESA AUTOLPR feature.

In addition to printing from batch using the LPR client application or the AUTOLPR feature, TCP/IP for VSE/ESA also supports automatically printing files generated with CICS RCF. Similar to printing from batch, you must specify the name of a Script file within the VSE/POWER *user-information* field or the HOSTNAME parameter of the DEFINE EVENT definition. This Script file must specify the remote IP address of the system hosting the LPD (Line Printer Daemon) and the name of the printer to print on the specified host.

With this information in place, TCP/IP for VSE/ESA will send the print output off the VSE/POWER list queue to the specified destination, assuming an **EVENT** (see example below in this section) was defined to TCP/IP for VSE/ESA covering the specified VSE/POWER class.

A detailed discussion of AUTOLPR can be found in the *TCP/IP for VSE 1.5 User's Guide*.

Specification in the CICS RCF Program

In the CICS RCF Program you need to specify the VSE/POWER class, and the name of the Script file in the user-information field. These required values will be passed to VSE/POWER.

In the following example those values are

- **CLASS('T')** for the VSE/POWER class

- **USERDATA(SCRIPTNM)** for the user information

but they may also contain other values matching your requirements.

```

...
DFHEISTG DSECT
SCRIPTNM DS    CL16
...
TESTLPR CSECT
* Open output spoolfile
    MVC  SCRIPTNM,=CL16'SCRIPT2'    Set Script Name
*   Script-Name-Field should be 16 characters long
*   Script-Name-Field should be padded with blanks
    EXEC CICS SPOOLOPEN REPORT('LPRTST') USERDATA(SCRIPTNM) *
        TOKEN(OUTTOKEN) NOCC CLASS('T') NOSEP *
        RESP(RESPFLD) RESP2(RESP2FLD)
...

```

TCP/IP Definitions

Your TCP/IP for VSE/ESA configuration file IPINITxx.L should contain the following (or similar) definitions. If you have not defined them in your startup configuration, you can also specify those definitions interactively to TCP/IP for VSE/ESA.

- Definition of AUTOLPR for VSE/POWER LST Queue, **CLASS T**
DEFINE EVENT,ID=LPR,TYPE=POWER,CLASS=T,QUEUE=LST
- Symbolic name **REMHOST** for IP address **9.1.2.3**
DEFINE NAME,NAME=REMHOST,IPADDR=9.1.2.3
- Script File Definition for Script **SCRIPT2**, backed by VSE library member **PRTLOCAL.L**
DEFINE NAME,NAME=SCRIPT2,SCRIPT=PRTLOCAL

Script File Definition

The Script file needs to be catalogued as L source book in a VSE library, accessible through the // **LIBDEF SOURCE,SEARCH** chain. In the preceding example the member name is **PRTLOCAL.L**. The Script file contains the required host and printer definitions.

```

* $$ JOB JNM=CATAL,CLASS=A,DISP=D
// JOB CATAL CATALOG SCRIPT MEMBER PRTLOCAL.L
// EXEC LIBR
ACC S=PRD2.CONFIG
CAT PRTLOCAL.L R=Y
SET HOST=REMHOST          SYMBOLIC HOST NAME
SET PRINTER=PRINTER1
/+
/*
/&
* $$ E0J

```

GPS and RCF

The following example shows the definitions to be done for TCP/IP-GPS, VTAM and CICS for use of the GPS by the Report Controller Feature (RCF) of CICS.

A detailed description of all parameters to define a GPS daemon can be found in *TCP/IP for VSE 1.5 Optional Features*.

Examples TCP/IP for VSE/ESA

Defining to VTAM

```
TCPVRT  VBUILD TYPE=APPL
GPS1    APPL  AUTH=(ACQ),DLOGMOD=DSC2K
GPS2    APPL  AUTH=(ACQ),DLOGMOD=DSC2K
```

Note: If a VTAM printer should be shared between different CICS applications it has to be released first from one CICS before it can be used with another CICS. This is accomplished by defining RELREQ=YES in the CICS TYPETERM definition of that printer. But to activate RELREQ=YES the following must be coded in the VTAM APPL statement: SESSLIM=YES. For more details refer to the *VTAM Programming Guide*.

Defining to CICS

```
CEDA DEFine TYPeterm:  GPSRPT  Group:  VSETERM1

CEDA DEFine TERminal:  GPS1     Group:  VSETERM1
CEDA DEFine TERminal:  GPS2     Group:  VSETERM1
```

Defining to TCP/IP

```
DEFINE FILE,PUBLIC='PRD2.GPSWORK',DLBL=PRD2,TYPE=LIBRARY
*
* GPS1 is a IBM4248
DEFINE GPSD,ID=GPS001,STORAGE='PRD2.GPSWORK',TERMNAME=GPS1,-
IPADDR=nnn.nnn.nnn.nnn,PRINTER=LOCAL
*
* GPS2 is a IBM3130
DEFINE GPSD,ID=GPS002,STORAGE='PRD2.GPSWORK',TERMNAME=GPS2,-
IPADDR=nnn.nnn.nnn.nnn,PRINTER=printername
```

Note that the 'printername' is case sensitive.

Defining to RCF

```
PRINTER          DESTINATION
GPS1              GPS1
GPS2              GPS2
```

TELNET and Subnetting in a Class-C Network

The following example shows how a Class-C network can be divided to provide different subnets for Telnet usage. This is done by using different subnet masks for the different subnets.

Requirement/Question

```
CICS Terminal Id = TA31xx  -> IPaddress 9.222.66.1   - 27
                  = TA03xx  -> IPaddress 9.222.66.65  - 91
                  = TA06xx  -> IPaddress 9.222.66.129 - 155
```

How can I differ between the different terminal-ids so that each user is identifiable ?

Answer

```
DEFINE MASK,ID=net1mask,NETWORK=9.222.66.0,MASK=255.255.255.224
DEFINE MASK,ID=net2mask,NETWORK=9.222.66.64,MASK=255.255.255.224
DEFINE MASK,ID=net3mask,NETWORK=9.222.66.128,MASK=255.255.255.224
DEFINE TELNETD,ID=teln1,MENU=MENU3,COUNT=30,TERMNAME=TA31, -
                  IPADDR=9.222.66.0
DEFINE TELNETD,ID=teln2,MENU=MENU4,COUNT=30,TERMNAME=TA03, -
```

```

IPADDR=9.222.66.64
DEFINE TELNETD, ID=teln3, MENU=MENU5, COUNT=30, TERMNAME=TA06, -
IPADDR=9.222.66.128

```

TELNET daemons and logmode

This example shows how to make TELNET sessions queryable.

If the TELNET daemon definitions are made as follows

```

DEFINE TEL, ID=MYTEL, TAR=DBDCCICS, TERM=T1000, CO=20, LOGMODE=SP3272QN, -
LOGMODE3=SP3272QN, LOGMODE4=SP3272QN, LOGMODE5=SP3272QN

```

then all types of terminals (model 2, 3, 4, and 5) will be queryable. If only LOGMODE=SP3272QN

is set, a model 3 will not have SP3272QN but the default value D4B32783 which does not have an Extended Datastream. This is why the above definitions are recommended. In case queryable sessions are not desired, the IUI default logmodes with EXTDS are as follows

```

DEFINE TEL, ID=MYTEL, TAR=DBDCCICS, TERM=T1000, CO=20, LOGMODE=SP3272EN, -
LOGMODE3=SP3273EN, LOGMODE4=NSX32704, LOGMODE5=NSX32705

```

Since there is no explicit logmode for model 4 and 5 in IUI, the VTAM default is used.

VSAMCAT Usage

Instead of defining every VSAM file that you want to access via FTP, NFS, or HTTP, you can define the VSAM catalog to TCP/IP for VSE/ESA and let it dynamically build DLBL and file control block information for every cluster in the catalog.

A detailed description of the VSAMCAT parameter of the DEFINE FILE command can be found in *TCP/IP for VSE 1.5 Commands*.

1. Defining the catalog to VSE

The first step in using a VSAM catalog is to have a DLBL defining the catalog. The VSAMCAT fileIO driver will read the catalog sequentially in order to acquire cluster attribute information. Because of that, the DLBL must have a ",CAT=" parameter pointing back to itself. For example, IJSYSUC:

```
// DLBL IJSYSUC, 'VSAM.USER.CATALOG', , VSAM,CAT=IJSYSUC
```

You can either modify the entry in standard labels, or create a new one and put it in the TCP/IP startup JCL. In either case, it is important that TCP/IP find the DLBL for the catalog and the catalog entry has a ",CAT=" pointing back to itself.

2. Defining the catalog to TCP/IP

Now that VSE knows about the catalog, let's tell TCP/IP. Here is a sample definition for that same catalog:

```
DEFINE FILE,PUBLIC='IJSYSUC',DLBL=IJSYSUC,TYPE=VSAMCAT
```

Of course, the public name can be anything you want, but for this example, we'll make it the same as the DLBL name.

3. Using the catalog

Now that you have the VSE and TCP/IP systems know about the catalog and if it actually exists, you can access it with FTP by issuing a "ChDir" into (in this case) IJSYSUC. The first time that you do this, you will see a message on

Examples TCP/IP for VSE/ESA

SYSLOG indicating that the fileIO module, IPNFVCAT has been loaded into partition storage. When you perform a DirList, IPNFVCAT will read the catalog information and return a listing of information. When you issue a RETRIEve against a specific entry, IPNFVCAT will check the partition to see if a DLBL already exists. If it does not, then one will be dynamically added to the partition for you. After that, the file is transferred for you.

The only exception to this are VSAM-controlled-SAM files. Because the VSAM catalog is not updated with information such as the number of records, you will not be able to retrieve these files using VSAMCAT. In this case you will need to define each of these files individually to TCP/IP as "TYPE=SAM" and retrieve these using the DTFSD methodology.

Performing a PUT to the VSAM catalog is different. For FTP, you need either have the cluster already defined or the cluster can be dynamically defined or a REXX program can be run from FTP to define the file. For NFS, a dynamic DEFINE CLUSTER is automatically performed for you.

Finally, you can perform a DELETE against the VSAM files, and IDCAMS will be dynamically invoked to perform a DELETE CLUSTER for you. However, a RENAME will not work for VSAMCAT files.

Using the Command preprocessor

EXEC TCP based programs require the TCP/IP for VSE/ESA preprocessor program IPNETPRE to generate language specific code constructs.

When you execute IPNETPRE, you specify two options by way of the PARM field of the EXEC statement. for example

```
// EXEC IPNETPRE,SIZE=IPNETPRE,PARM='LANG=COBOL,ENV=CICS'  
* $$ SLI MEM=COBSRC.C,S=PRD3.INGO  
/*
```

LANG

The **LANG=xxx** parameter tells the preprocessor the language being processed. Supported values for xxx are:

ASSEMBLER

High-Level Assembler

COBOL

COBOL for VSE

PL1 PL/I for VSE

ENV The **ENV=xxx** parameter indicates the environment that the finished program will execute in. There are two acceptable values for xxx.

BATCH

The program will execute in batch mode.

CICS The program will be executed under CICS.

Note:

1. For ENV=CICS programs always run the TCP/IP preprocessor before the CICS preprocessor. You must execute them in this order because the TCP/IP preprocessor will generate EXEC CICS statements that must be replaced by the CICS preprocessor.

Refer to *TCP/IP for VSE 1.5 Programmer's Reference* for a detailed description of the TCP/IP for VSE/ESA preprocessor.

Sample Programs

The following sample programs provide the same functionality presented in a variety of languages. In each case, note any "special" techniques shown for manipulating data.

COBOL Example

```

IDENTIFICATION DIVISION.

PROGRAM-ID.          COBSRC.
AUTHOR.              JOHN DOE.
INSTALLATION.        WORTHINGTON OHIO.
DATE-WRITTEN.        AUGUST 2, 1995.
DATE-COMPILED.

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER.     IBM-370.
OBJECT-COMPUTER.     IBM-370.

DATA DIVISION.

EXEC TCP CONTROL DOUBLE(NO)
END-EXEC.

WORKING-STORAGE SECTION.
01 WORK-AREA-ONE.
   05 PART1          PICTURE 9(4) COMP.
   05 PART2          PICTURE 9(4) COMP.
   05 PART3          PICTURE 9(4) COMP.
   05 PART4          PICTURE 9(4) COMP.
   05 IPADDRESS.
      10 IPAD1       PICTURE X.
      10 IPAD2       PICTURE X.
      10 IPAD3       PICTURE X.
      10 IPAD4       PICTURE X.
   05 HALFWORD      PICTURE 9(4) COMP.
   05 HALFWORD-X    REDEFINES HALFWORD.
      10 BYTEX1      PICTURE X.
      10 BYTEX2      PICTURE X.
   05 RESULTS.
      10 RECB        PICTURE X(4).
      10 RLOPORT     PICTURE 9(4) COMP.
      10 RFOPORT     PICTURE 9(4) COMP.
      10 RFOIP       PICTURE X(4).
      10 RCOUNT     PICTURE 9(4) COMP.
      10 RFLAGS      PICTURE X.
      10 RCODE       PICTURE X.
      10 RTERMTY     PICTURE X(40).
   05 MY-DESC       PICTURE X(4).
01 LOCAL-PORT      PICTURE 9(4) COMP.
01 BUFFER.
   05 WORKAREA      PICTURE X(512).

PROCEDURE DIVISION.

BEGIN.
-----*
*                               *
*           First Test          *
*                               *
*-----*
*
*   Setup IPADDRESS to hold 172.20.10.10 in binary
*

```

Examples TCP/IP for VSE/ESA

```
MOVE 172 TO HALFWORD.
MOVE BYTEX2 TO IPAD1.
MOVE 20 TO HALFWORD.
MOVE BYTEX2 TO IPAD2.
MOVE 10 TO HALFWORD.
MOVE BYTEX2 TO IPAD3.
MOVE 10 TO HALFWORD.
MOVE BYTEX2 TO IPAD4.
*
* Attempt to open a connection at 172.20.10.10 port 2000
*
EXEC TCP OPEN FOREIGNPORT(2000)
                FOREIGNIP(IPADDRESS)
                LOCALPORT(0)
                RESULTAREA(RESULTS)
                DESCRIPTOR(MY-DESC)
                ACTIVE
                WAIT(YES)
                ERROR(SECOND-TEST)
END-EXEC.
    DISPLAY 'Open has completed'.
*
* Receive a piece of data
*
EXEC TCP RECEIVE
                TO(BUFFER)
                LENGTH(512)
                RESULTAREA(RESULTS)
                DESCRIPTOR(MY-DESC)
                WAIT(YES)
                ERROR(SECOND-TEST)
END-EXEC.
    DISPLAY 'Receive has completed'.
*
* Close the connection
*
EXEC TCP CLOSE
                RESULTAREA(RESULTS)
                DESCRIPTOR(MY-DESC)
                WAIT(YES)
                ERROR(SECOND-TEST)
END-EXEC.
    DISPLAY 'Close has completed'.
*-----*
*                                     *
*           Second Test                *
*                                     *
*-----*
SECOND-TEST.
*
* Attempt to open another connection
*
MOVE 2000 TO LOCAL-PORT.
EXEC TCP OPEN FOREIGNPORT(0)
                FOREIGNIP(0)
                LOCALPORT(LOCAL-PORT)
                RESULTAREA(RESULTS)
                DESCRIPTOR(MY-DESC)
                PASSIVE
                WAIT(YES)
                ERROR(ERROR-SPOT)
END-EXEC.
    DISPLAY 'Second Open has completed'.
*
* Display the foreign IP address
*
MOVE RFOIP TO IPADDRESS.
```



```

MOVE IPAD1 TO BYTE2.
MOVE HALFWORD TO PART1.
MOVE IPAD2 TO BYTE2.
MOVE HALFWORD TO PART2.
MOVE IPAD3 TO BYTE2.
MOVE HALFWORD TO PART3.
MOVE IPAD4 TO BYTE2.
MOVE HALFWORD TO PART4.
DISPLAY PART1 '.' PART2 '.' PART3 '.' PART4
*
*   Send another piece of data
*
EXEC TCP SEND
        FROM(BUFFER)
        LENGTH(512)
        RESULTAREA(RESULTS)
        DESCRIPTOR(MY-DESC)
        WAIT(YES)
        ERROR(ERROR-SPOT)
END-EXEC.
        DISPLAY 'Second Send has completed'.
*
*   Close the second connection
*
EXEC TCP CLOSE
        RESULTAREA(RESULTS)
        DESCRIPTOR(MY-DESC)
        WAIT(YES)
        ERROR(ERROR-SPOT)
END-EXEC.
        DISPLAY 'Second Close has completed'.

        STOP RUN.

ERROR-SPOT.

        STOP RUN.

```

PL/I Example

SAMPLE4: PROCEDURE OPTIONS(MAIN);

```

DCL IPADDRESS      BINARY FIXED(31,0);
DCL MY-DESC        CHAR(4);
DCL 1 RESULTS,
    2 RECB          CHAR(4),
    2 RLOPORT       BINARY FIXED(15,0),
    2 RFOPORT       BINARY FIXED(15,0),
    2 RFOIP         CHAR(4),
    2 RCOUNT       BINARY FIXED(15,0),
    2 RFLAGS        CHAR(1),
    2 RCODE         CHAR(1),
    2 RTERMTY       CHAR(40);
DCL MY-DESC        CHAR(4);
DCL LOCAL-PORT     BINARY FIXED(15,0);
DCL BUFFER         CHAR(512);

```

```

/*-----*
*
*           First Test
*
*-----*/
/*
*   Attempt to open a connection at 172.20.10.10 port 2000
*/
EXEC TCP OPEN FOREIGNPORT(2000)
             FOREIGNIP(IPADDRESS)

```

Examples TCP/IP for VSE/ESA

```
LOCALPORT(0)
RESULTAREA(RESULTS)
DESCRIPTOR(MY-DESC)
ACTIVE
WAIT(YES)
ERROR(SECOND-TEST);

/*
 * Receive a piece of data
 */
EXEC TCP RECEIVE
    TO(BUFFER)
    LENGTH(512)
    RESULTAREA(RESULTS)
    DESCRIPTOR(MY-DESC)
    WAIT(YES)
    ERROR(SECOND-TEST);

/*
 * Close the connection
 */
EXEC TCP CLOSE
    RESULTAREA(RESULTS)
    DESCRIPTOR(MY-DESC)
    WAIT(YES)
    ERROR(SECOND-TEST);

SECOND-TEST:
/*-----*
 *                               *
 *           Second Test         *
 *                               *
 *-----*
 *
 * Attempt to open a connection
 */
LOCAL-PORT = 2000;
EXEC TCP OPEN FOREIGNPORT(0)
    FOREIGNIP(0)
    LOCALPORT(LOCAL-PORT)
    RESULTAREA(RESULTS)
    DESCRIPTOR(MY-DESC)
    PASSIVE
    WAIT(YES)
    ERROR(ERROR-SPOT);

/*
 * Display the foreign IP address
 */

/* Need code here..... */

/*
 * Receive a piece of data
 */
EXEC TCP SEND
    FROM(BUFFER)
    LENGTH(512)
    RESULTAREA(RESULTS)
    DESCRIPTOR(MY-DESC)
    WAIT(YES)
    ERROR(ERROR-SPOT);

/*
 * Close the connection
 */
EXEC TCP CLOSE
    RESULTAREA(RESULTS)
    DESCRIPTOR(MY-DESC)
    WAIT(YES)
```

```
ERROR(ERROR-SPOT);
```

```
RETURN;
END SAMPLE4;
```

Compiling Your Program

Once you have coded your application program and it has passed through the preprocessor, it must then be submitted to the appropriate compiler. In many instances, you will also need to pass the output from our pre-compiler through one or more pre-compilers. The following examples shows one method for doing this. They use the COBOL example COBSRC.C as shown in "COBOL Example" on page 517.

Compiling a COBOL Program for Batch

The first example shows how to compile source COBSRC.C (see "COBOL Example" on page 517) stored in library PRD3.INGO and generating phase SAMPLEB.

Step 1 - Main Job

The main job frame work will be the same for BATCH as well as CICS runtime environment. It will call procedure COMSTP1.PROC stored in PRD3.INGO.

```
* $$ JOB JNM=COMPILE,CLASS=4,DISP=D
* $$ LST CLASS=W,DISP=D
* $$ PUN CLASS=4,DISP=I
// JOB COMPILE TCPIP PROGRAM
// LIBDEF *,SEARCH=(PRD3.INGO,PRD2.TCPIPC,PRD1.BASE,PRD2.PROD,PRD2.SCEEBASE)
// EXEC PROC=COMSTP1
/&
* $$ EOJ
```

Step 2 - Processing EXEC TCP Statements

Procedure COMSTP1.PROC calls the TCP/IP preprocessor IPNETPRE and generates a new JOB using utility program IESINSRT. This new JOB is named CATAL1 and is aimed to catalog the TCP/IP preprocessor output as PRETCP.DAT into PRD3.INGO. Then it calls procedure COMSTP2.PROC for further processing.

```
// ASSGN SYSIPT,SYSRDR
// EXEC IESINSRT
$ $$ JOB JNM=CATAL1,CLASS=4,DISP=D
$ $$ LST CLASS=W,DISP=D
$ $$ PUN CLASS=4,DISP=I
// JOB CATAL1 CATALOG OUTPUT OF THE TCPIP preprocessor
// LIBDEF *,SEARCH=(PRD3.INGO,PRD2.TCPIPC,PRD1.BASE,PRD2.PROD,PRD2.SCEEBASE)
// LIBDEF *,CATALOG=PRD3.INGO
// EXEC LIBR
  ACC S=PRD3.INGO
  CATALOG PRETCP.DAT EOD=/( REPLACE=YES
* $$ END
// OPTION DECK
*
* Process EXEC TCP source for CICS
*
// EXEC IPNETPRE,SIZE=IPNETPRE,PARM='LANG=COBOL,ENV=BATCH'
* $$ SLI MEM=COBSRC.C,S=PRD3.INGO
/*
// EXEC IESINSRT
/(
/*
```

Examples TCP/IP for VSE/ESA

```
// EXEC PROC=COMSTP2
#&
$ $$ E0J
* $$ END
```

Step 3 - Compiling and Link-Editing

Procedure COMSTP2.PROC invokes the **COBOL for VSE** Compiler and calls the Linkage Editor to link the OBJ deck generated by the compiler. The resulting phase SAMPLEB is stored into PRD3.INGO as

```
// LIBDEF *,CATALOG=PRD3.INGO
```

is still active.

```
*
* Compile and link phase SAMPLEB for Batch
*
// OPTION CATAL
  PHASE SAMPLEB,*
// EXEC IGYCRCTL,SIZE=IGYCRCTL
  CBL TEST APOST
* $$ SLI MEM=PRETCP.DAT,S=PRD3.INGO
/*
// EXEC LNKEDT
```

Compiling a COBOL Program for CICS

The second example shows how to compile source COBSRC.C (see "COBOL Example" on page 517) stored in library PRD3.INGO and generating phase SAMPLEC.

Step 1 - Main Job

The main job frame work is the same as already shown for the BATCH environment. It will call procedure COMSTP1.PROC stored in PRD3.INGO.

```
* $$ JOB JNM=COMPILE,CLASS=4,DISP=D
* $$ LST CLASS=W,DISP=D
* $$ PUN CLASS=4,DISP=I
// JOB COMPILE TCPIP PROGRAM
// LIBDEF *,SEARCH=(PRD3.INGO,PRD2.TCPIPC,PRD1.BASE,PRD2.PROD,PRD2.SCEEBASE)
// EXEC PROC=COMSTP1
/&
* $$ E0J
```

Step 2 - Processing EXEC TCP Statements

Procedure COMSTP1.PROC calls the TCP/IP preprocessor **IPNETPRE** and generates a new JOB using utility program **IESINSRT**. This new JOB is named CATAL1 and is aimed to catalog the TCP/IP preprocessor output as PRETCP.DAT into PRD3.INGO. Then it calls procedure COMSTP2.PROC for further processing.

```
// ASSGN SYSIPT,SYSRDR
// EXEC IESINSRT
$ $$ JOB JNM=CATAL1,CLASS=4,DISP=D
$ $$ LST CLASS=W,DISP=D
$ $$ PUN CLASS=4,DISP=I
// JOB CATAL1 CATALOG OUTPUT OF THE TCPIP preprocessor
// LIBDEF *,SEARCH=(PRD3.INGO,PRD2.TCPIPC,PRD1.BASE,PRD2.PROD,PRD2.SCEEBASE)
// LIBDEF *,CATALOG=PRD3.INGO
// EXEC LIBR
  ACC S=PRD3.INGO
  CATALOG PRETCP.DAT EOD=/( REPLACE=YES
* $$ END
// OPTION DECK
```

```

*
* Process EXEC TCP source for CICS
*
// EXEC IPNETPRE,SIZE=IPNETPRE,PARM='LANG=COBOL,ENV=CICS'
* $$ SLI MEM=COBSRC.C,S=PRD3.INGO
/*
// EXEC IESINSRT
/(
/*
// EXEC PROC=COMSTP2
#&
$ $$ EOJ
* $$ END

```

Step 3 - Processing EXEC CICS Statements

As the TCP/IP preprocessor has generated EXEC CICS statements where appropriate, for example to allocate storage or to WAIT according to the CICS programming model we have to invoke the CICS preprocessor before calling the COBOL compiler.

As shown in step 2 already, COMSTP2.PROC again dynamically generates a new job named CATAL2. It is aimed to store the output from the CICS preprocessor as PRECICS.DAT before calling COMSTP3.PROC for the final compile and link-edit steps.

```

// ASSGN SYSIPT,SYSRDR
// EXEC IESINSRT
$ $$ JOB JNM=CATAL2,CLASS=4,DISP=D
$ $$ LST CLASS=W,DISP=D
$ $$ PUN CLASS=4,DISP=I
// JOB CATALOG OUTPUT OF THE CICS preprocessor
// LIBDEF *,SEARCH=(PRD3.INGO,PRD2.TCPIPC,PRD1.BASE,PRD2.PROD,PRD2.SCEEBASE)
// LIBDEF *,CATALOG=PRD3.INGO
// EXEC LIBR
    ACC S=PRD3.INGO
    CATALOG PRECICS.DAT EOD=/(    REPLACE=YES
* $$ END
*
* Starting CICS command preprocessor
*
// EXEC DFHECP1$,PARM='CICS'
* $$ SLI MEM=PRETCP.DAT,S=PRD3.INGO
/*
// EXEC IESINSRT
/(
/*
// EXEC PROC=COMSTP3
#&
$ $$ EOJ
* $$ END

```

Step 4 - Compiling and Link-Editing

Procedure COMSTP3.PROC invokes the **COBOL for VSE** Compiler and calls the Linkage Editor to link the OBJ deck generated by the compiler. The resulting phase SAMPLEC is stored into PRD3.INGO as

```
// LIBDEF *,CATALOG=PRD3.INGO
```

is still active.

```

*
* Compile and link phase SAMPLEC for CICS
*

```

Examples TCP/IP for VSE/ESA

```
// OPTION CATAL
  PHASE SAMPLEC
  INCLUDE DFHELII
// EXEC IGYCRCTL,SIZE=IGYCRCTL
  CBL TEST APOST
* $$ SLI MEM=PRECICS.DAT,S=PRD3.INGO
/*
// EXEC LNKEDT
```

Appendix B. Debugging Facility for EZASMI and EZASOKET Interfaces (EZAAPI Trace)

The EZA TCP/IP HLL APIs (that is the EZASMI macro and the EZASOKET call interface) have a trace facility integrated.

This trace facility is available with TCP/IP for VSE/ESA and with Linux Fast Path. It generates one (or more) trace messages for each EZASMI or EZASOKET socket call. It allows to trace these calls either for all partitions in the system or for selected partitions and dynamic classes. Trace messages may be directed to SYSLOG or to SYSLST.

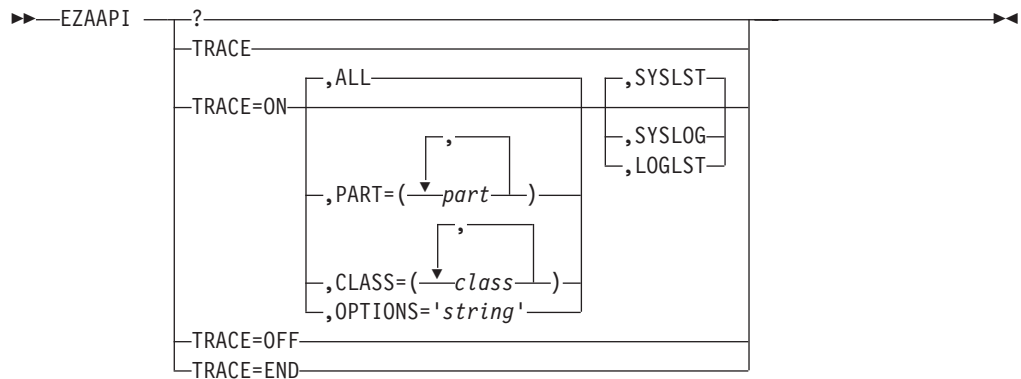
This trace facility is simply referred to as "EZAAPI trace".

Requirements for Usage

The EZAAPI trace needs module EZASOHTR loaded into the SVA (which is the default with z/VSE).

Format

The EZAAPI trace can be activated and controlled with the AR command EZAAPI.



Parameters

EZAAPI ?

Display the command syntax

EZAAPI TRACE

Display current trace settings

EZAAPI TRACE=ON

Define and start or resume starting

(default with no trace defined yet)

Define and start trace with defaults ALL and SYSLST

(default after EZAAPI TRACE=OFF)

Resume trace

All Define and start trace for all partitions in the system

Debugging Facility for EZASMI / EZASOKET Interfaces

PART=(part,..)

Define and start trace for selected partitions

CLASS=(class,..)

Define and start trace for selected dynamic classes

OPTIONS='string'

Set special trace options. Can be a hex character *string* with a maximum length of 8 bytes. OPTIONS=' ' resets and clears an existing OPTION specification. It is up to the TCP/IP interface routines whether the OPTIONS string is being used. Refer to the appropriate product documentation.

EZAAPI TRACE=OFF

Suspend current trace

EZAAPI TRACE=END

End tracing and clear all trace definitions

SYSLST

trace output is send to SYSLST (if SYSLST is assigned)

SYSLOG

trace output is send to SYSLOG

LOGLST

trace output is send to SYSLOG and SYSLST.

Output

The EZAAPI trace generates self-explanatory messages.

When the EZAAPI trace is started, the next call to the EZA interface (EZASMI or EZASOKET) will trigger message:

```
EZA001I EZAS0H99 (Level *date*) started
```

Every call to the EZA interface will then produce

1. one start message, like
EZA002I >>> SOCKET processing starts ...
2. eventually some more messages showing additional input.
EZA033I .. with AF/SOCTYPE/PROTO=02/01/00
3. and one (or more) completion messages.
EZA003I SOCKET returns with RC/ERRNO=00000/00000

If TCP/IP for VSE/ESA is being used with your EZA socket application, activation of the EZAAPI trace automatically activates the BSD-C trace of TCP/IP for VSE/ESA (called \$SOCKDBG trace).

Appendix C. Advanced OSAX Device Driver Configuration

This appendix provides an overview of advanced OSAX device driver configuration with z/VSE.

Configurable QDIO Buffers

The number of QDIO input queue buffers can be configured.

You might need this, if you want to extend z/VSE solutions by exploiting Linux on System z using a HiperSockets network for high-speed TCP/IP connectivity between z/VSE and Linux on System z. The best performance can be achieved, if data is always delivered successfully without the need of resending. Because HiperSockets transfers data synchronously, successful delivery of data depends on free QDIO (Queued Direct I/O) input buffers of the target system. z/VSE uses a default of eight QDIO input buffers. This might not always be sufficient, especially if you have increased the number of QDIO input buffers on the Linux on System z system.

To configure the number of QDIO input buffers for HiperSockets (CHPID type IQD) and OSA-Express devices (CHPID types OSD and OSX), the configuration skeleton SKOSACFG in ICCF library 59 can be used. Additional QDIO input buffers might require to increase the size of the TCP/IP partition.

Note: z/VSE needs 1 MB of 31-bit partition getvis space per link (for 8 input queue buffers). For each additional input queue buffer, z/VSE needs 64 K (OSA-Express) and up to 64 K (HiperSockets - depending on your IOCDs definition) additional 31-bit partition getvis space. Therefore, if you increase the count of input queue buffers, you might also have to increase the partition getvis space. Because the input queue buffers are PFIxed, you also have to increase the above value of the JCL SETPFIX LIMIT statement in your TCP/IP startup job accordingly.

VLAN Support

Virtual LAN (VLAN) support allows a TCP/IP stack to register a specific VLAN identifier for a Layer 2 or Layer 3 link for both IPv4 or IPv6. There are two ways to configure your system to use VLAN:

- **With** TCP/IP use the IPv6/VSE LINK command. For details refer to the IPv6/VSE Installation Guide at:
<http://www.ibm.com/systems/z/os/zvse/documentation/#tcpip>
- **Without** TCP/IP use phase IJBOCONF containing the VLANs to be used with your OSAX devices.

A *virtual LAN* allows a physical network to be divided administratively into separate logical networks. These logical networks operate as if they are physically independent of each other.

z/VSE provides VLAN support for OSA Express (CHPID type OSD and OSX) and HiperSockets™ devices.

- In a Layer 3 configuration, VLANs can be transparently used by IPv6/VSE and TCP/IP for VSE/ESA.

- If you wish to configure VLANs for OSA-Express (CHPID type OSD and OSX) devices in a Layer 2 configuration that carries IPv6 traffic, you require IPv6/VSE.

Global VLAN Support:

- One Global VLAN can be defined per link.
- If you define a Global VLAN, you cannot define other VLANs for the same link.

Index

A

abend codes
 E20L 480
 E20T 480
ACCEPT (call) 201
ACCEPT (macro) 296
accept()
 use in server 496
accept() library function 85
accessibility xv
address
 family (domain) 500
 structures 500
address and name information error
 description
 TCP/IP call function
 gai_strerror() 106
address, host 111
ADDRINFO parameter on call socket
 interface
 on FREEADDRINFO 211
ADDRINFO parameter on macro socket
 interface
 on FREEADDRINFO 306
AF parameter on call socket interface
 on SOCKET 275
AF parameter on macro socket interface
 on NTOP 351
 on PTON 353
 on SOCKET 376
AF_INET domain
 example 96
 socket descriptor created in 96
AF_INET domain parameter 500
AF_INET6 parameter on call socket
 interface
 on PTON 253
aio_cancel() library function 88
aio_error() library function 89
aio_read() library function 90
aio_return() library function 92
aio_suspend() library function 93
aio_write() library function 94
ALTER 453
APITYPE parameter on macro socket
 interface
 on INITAPI 346
AREA parameter on macro socket
 interface
 on GSKFREEMEM 335
ASCII data format 507
ASYNC parameter on macro socket
 interface
 on INITAPI 346
asynch I/O op., retrieve error status 89
asynch I/O op., retrieve status 92
asynch I/O request, cancel 88
asynch I/O request, wait for 93
asynch read from socket 90
asynch write to a socket 94
asynchronous ECB routine 294

asynchronous exit routine 294
asynchronous macro, coding
 example 294
AUTHTYPE parameter on macro socket
 interface
 on GSKINIT 337
AUTOLPR and CICS RCF 512
automatic startup 475
autonomous FTP 511

B

BACKLOG parameter on call socket
 interface
 on LISTEN 250
BACKLOG parameter on macro socket
 interface
 on LISTEN 350
Basic Security Manager (BSM) 23, 43
big endian 500
bind ()
 use in server 496
BIND (call) 204
BIND (macro) 298
bind() library function 96
BIT-MASK parameter on call socket
 interface
 on EZACIC06 282
BIT-MASK-LENGTH parameter on call
 socket interface
 on EZACIC06 282
BLANKING.HTML 13
BSM (Basic Security Manager) 23, 43
BUF parameter on call socket interface
 on READ 254
 on RECV 257
 on RECVFROM 258
 on SEND 267
 on SENDTO 268
 on WRITE 278
BUF parameter on macro socket interface
 on GETIBMOPT 321
 on GSKSSOCREAD 343
 on GSKSSOCWRITE 345
 on READ 354
 on RECV 357
 on RECVFROM 358
 on SEND 367
 on SENDTO 369
 on WRITE 382
buffers
 data stored in 164
 receive data and store in 165
 receive messages and store in 167,
 168

C

cache file, VSAM 469

CALAREA parameter on macro socket
 interface
 on CANCEL 300
CALL Instruction Interface for Assembler,
 PL/1, and COBOL 199
Call Instructions for Assembler, PL/1,
 and COBOL Programs
 ACCEPT 201
 BIND 204
 CLOSE 206
 CONNECT 207
 EZACIC04 281
 EZACIC05 281
 EZACIC06 282
 EZACIC08 284
 EZACIC09 286
 FCNTL 209
 FREEADDRINFO 211
 GETADDRINFO 211
 GETCLIENTID 218
 GETHOSTBYADDR 219
 GETHOSTBYNAME 221
 GETHOSTID 223
 GETHOSTNAME 223
 GETIBMOPT 224
 GETNAMEINFO 225
 GETPEERNAME 229
 GETSOCKNAME 230
 GETSOCKOPT 232
 GIVESOCKET 234
 GSKFREEMEM 236
 GSKGETCIPHINF 237
 GSKGETDNBYLAB 238
 GSKINIT 239
 GSKSSOCLOSE 240
 GSKSSOCINIT 241
 GSKSSOCREAD 244
 GSKSSOCRESET 245
 GSKSSOCWRITE 245
 GSKUNINIT 246
 INITAPI 247
 IOCTL 249
 LISTEN 250
 NTOP 251
 PTON 253
 READ 254
 READV 255
 RECV 257
 RECVFROM 258
 SELECT 261
 SELECTEX 265
 SEND 267
 SENDTO 268
 SETSOCKOPT 270
 SHUTDOWN 274
 SOCKET 275
 TAKESOCKET 276
 TERMAPI 278
 WRITE 278
 WRITEV 279
callable functions 85

- callback routines 429
- CANCEL (macro) 300
- CANNLEN parameter on call socket interface
 - on GETADDRINFO 211
- CANNLEN parameter on macro socket interface
 - on GETADDRINFO 307
- CARROOTS parameter on macro socket interface
 - on GSKINIT 337
- CERTINFO parameter on macro socket interface
 - on GSKSSOCINIT 339
- CHAR-MASK parameter on call socket interface
 - on EZACIC06 282
- CHAR-STRING-LENGTH parameter on call socket interface
 - on EZACIC06 282
- child server 494
- CICS 475
 - starting automatically 475
 - starting manually 475
 - starting with program link 480
- CICS Report Controller Feature (RCF) and AUTOLPR 512
- CICS Report Controller Feature (RCF) and GPS 513
- CIPHLEVEL parameter on macro socket interface
 - on GSKGETCIPHINF 335
- client
 - incoming requests, preparing server for 158
 - socket calls used in 494
- CLIENT parameter on call socket interface
 - on GETCLIENTID 218
 - on GIVESOCKET 234
 - on TAKESOCKET 276
- CLIENT parameter on macro socket interface
 - on GETCLIENTID 314
 - on GIVESOCKET 333
 - on TAKESOCKET 379
- CLOSE (call) 206
- CLOSE (macro) 301
 - use in child server 494
 - use in client 494
 - use in server 496
- close() library function 99
- COMMAND parameter on call socket interface
 - on EZACIC06 282
 - on GETIBMOPT 224
 - on IOCTL 249
- COMMAND parameter on macro interface
 - on FCNTL 209, 305
 - on GETIBMOPT 321
 - on IOCTL 348
- COMMAREA 491
- compile cobol for batch 521
- compile cobol for cics 522
- concurrent server 493
 - writing your own 496

- Configurable QDIO buffers 527
- configuration macro 446
- configuration transaction 451
- configure openssl code 424, 425
- configuring CICS TCP/IP 441
- configuring TCP/IP for VSE/ESA
 - configuring CICS 10
 - configuring TCP/IP using the IUI-based configuration dialog 16
 - supplying the product key 5
- CONNECT (call) 207
- CONNECT (macro) 302
- connect()
 - use in client 494
- connect() library function 100
- connection
 - duplex, shutting down 189
- connection between sockets 100
- connection request 85
- conversion routines 507
- convert Internet address format from binary to text
 - TCP/IP call function
 - inet_ntop() 154
- convert Internet address format from text to binary
 - TCP/IP call function
 - inet_pton() 155
- COPY 456
- creating
 - socket 190
 - socket pair 193
- CSKL transaction 502
- current host address 115
- CUSTDEF phase 6

D

- data
 - buffers, stored in 164
 - receiving 165
 - sending on socket 179
 - store in buffers 165
- data conversion 507
- data set
 - host information 181
 - network information 182
 - network services, opening 183
 - protocol, opening 182
- data structures, external
 - configuration data set 487
 - global work area 488
 - listener control area 492
 - parameter list for EZACIC20 491
- data translation, socket interface 280
 - ASCII to EBCDIC 281
 - bit-mask to character 282
 - character to bit-mask 282
 - EBCDIC to ASCII 281
- datagram
 - sending on socket 176
- Debugging Facility for EZASMI / EZASOKET Interfaces 525
- decimal host address
 - from network number 153
- DEFINE 458, 459
- defining customer information 6

- DELETE 463
- demonstration mode 4
- descriptor, socket 96
- Destination Control Table 443
- Device Driver Configuration 527
- DFHPCTIP 10
- DFHPPTIP 10
- disability xv
- DISPLAY 465
- DNAME parameter on macro socket interface
 - on GSKSSOCINIT 339
- DNS
 - EZACIC25, adding to RDO 442
- domain
 - address family 500
- Domain Name Server Cache 469
 - cache file 469
 - EZACICR macro 469
 - initialization module, creating 471
- DOMAIN parameter on call socket interface
 - on GETCLIENTID 218
 - on GIVESOCKET 234
 - on TAKESOCKET 276
- DOMAIN parameter on macro socket interface
 - on GIVESOCKET 333
 - on TAKESOCKET 379
- DSTADDR parameter on macro socket interface
 - on NTOP 351
 - on PTON 353
- DSTLEN parameter on macro socket interface
 - on NTOP 351
 - on PTON 353
- dual stack support 41
- duplex connection 189

E

- EBCDIC data format 507
- ECB parameter on macro socket interface
 - on ACCEPT 296
 - on BIND 298
 - on CANCEL 300
 - on CLOSE 301
 - on CONNECT 302
 - on FCNTL 305
 - on GETCLIENTID 314
 - on GETHOSTID 319
 - on GETHOSTNAME 320
 - on GETPEERNAME 326
 - on GETSOCKNAME 328
 - on GETSOCKOPT 330
 - on GIVESOCKET 333
 - on IOCTL 348
 - on LISTEN 350
 - on READ 354
 - on RECV 357
 - on SELECT 360
 - on SEND 367
 - on SETSOCKOPT 371
 - on SHUTDOWN 375
 - on SOCKET 376
 - on TAKESOCKET 379

ECB parameter on macro socket interface (*continued*)
on WRITE 382

EDCT001I message 70

EDCT002I message 70

EDCT003I message 70

EDCV001I message 70, 85

EDCV002I message 70

endhostent() TCP/IP function 103

endnetent() TCP/IP function 103

endprotoent() TCP/IP function 103

endservent() TCP/IP function 104

environmental support 483

ERETMSK parameter on call interface, on SELECT 261

ERETMSK parameter on call socket interface
on SELECT 261
on SELECTEX 265

ERETMSK parameter on macro socket interface
on SELECT 360
on SELECTEX 365

ERRNO parameter on call socket interface
on ACCEPT 201
on BIND 204
on CLOSE 206
on CONNECT 207
on FCNTL 209
on FREEADDRINFO 211
on GETADDRINFO 211
on GETCLIENTID 218
on GETHOSTNAME 223
on GETNAMEINFO 225
on GETPEERNAME 229
on GETSOCKNAME 230
on GETSOCKOPT 232
on GIVESOCKET 234
on INITAPI 247
on IOCTL 249
on LISTEN 250
on NTOP 251
on PTON 253
on READ 254
on READV 255
on RECV 257
on RECVFROM 258
on SELECT 261
on SELECTEX 265
on SEND 267
on SENDTO 268
on SETSOCKOPT 270
on SHUTDOWN 274
on SOCKET 275
on TAKESOCKET 276
on WRITE 278
on WRITEV 279

ERRNO parameter on macro socket interface
on ACCEPT 296
on BIND 298
on CANCEL 300
on CLOSE 301
on CONNECT 302
on FCNTL 305
on FREEADDRINFO 306

ERRNO parameter on macro socket interface (*continued*)
on GETADDRINFO 307
on GETCLIENTID 314
on GETHOSTNAME 320
on GETIBMOPT 321
on GETNAMEINFO 322
on GETPEERNAME 326
on GETSOCKNAME 328
on GETSOCKOPT 330
on GIVESOCKET 333
on GSKFREEMEM 335
on GSKGETCIPHINF 335
on GSKGETDNBYLAB 336
on GSKINIT 337
on GSKSSOCLOSE 338
on GSKSSOCINIT 339
on GSKSSOCREAD 343
on GSKSSOCRESET 344
on GSKSSOCWRITE 345
on GSKUNINIT 345
on INITAPI 346
on IOCTL 348
on LISTEN 350
on NTOP 351
on PTON 353
on READ 354
on READV 355
on RECV 357
on RECVFROM 358
on SELECT 360
on SELECTEX 365
on SEND 367
on SENDTO 369
on SETSOCKOPT 371
on SHUTDOWN 375
on SOCKET 376
on TAKESOCKET 379
on WRITE 382
on WRITEV 383

ERRNO values
sorted by name 74
sorted by value 74

ERROR parameter on macro socket interface
on ACCEPT 296
on BIND 298
on CANCEL 300
on CLOSE 301
on CONNECT 302
on FCNTL 305
on FREEADDRINFO 306
on GETADDRINFO 307
on GETCLIENTID 314
on GETHOSTBYADDR 315
on GETHOSTBYNAME 317
on GETHOSTID 319
on GETHOSTNAME 320
on GETIBMOPT 321
on GETNAMEINFO 322
on GETPEERNAME 326
on GETSOCKNAME 328
on GETSOCKOPT 330
on GIVESOCKET 333
on INITAPI 346
on IOCTL 348
on LISTEN 350

ERROR parameter on macro socket interface (*continued*)
on NTOP 351
on PTON 353
on READ 354
on READV 355
on RECV 357
on RECVFROM 358
on SELECT 360
on SELECTEX 365
on SEND 367
on SENDTO 369
on SETSOCKOPT 371
on SHUTDOWN 375
on SOCKET 376
on TAKESOCKET 379, 381
on WRITE 382
on WRITEV 383

ESNDMSK parameter on call socket interface
on SELECT 261
on SELECTEX 265

ESNDMSK parameter on macro socket interface
on SELECT 360
on SELECTEX 365

EWOULDBLOCK error return, call interface calls
RECV 257
RECVFROM 258

EXEC CICS LINK 480

EXEC CICS RETRIEVE 501

EXEC CICS START 501

Executing C Programs 84

EZAAPI trace 525

EZAC (configuration transaction) 451

EZAC start screen 476

EZACIC04, call interface, EBCDIC to ASCII translation 281

EZACIC05, call interface, ASCII to EBCDIC translation 281

EZACIC06, call interface, bit-mask translation 282

EZACIC08, HOSTENT structure interpreter utility 284

EZACIC09, call interface 286

EZACIC20, parameter list 491

EZACIC25 473

EZACICD (configuration macro) 446

EZACICR macro 469, 471

EZACICSE program 505

EZACICxx programs
defining in CICS 442
EZACIC25
Domain Name Server Cache 469

EZAO transaction
manual startup/shutdown 475

EZASMI, programming interface 60

EZASMI, debugging facility 525

EZASOKET, debugging facility 525

EZASOKET, programming interface 60

EZASTRUE program 83

EZAT transaction 83

EZATRUE task-related-user-exit 83

EZBREHST macro
use of 211, 307

F

Fast Path to Linux on System z 387
Fast Path to Linux 387
FCNTL (call) 209
FCNTL (macro) 305
fcntl() library function 104
files, defining to RDO 443
FLAGS parameter on call socket interface
 on GETNAMEINFO 225
 on RECV 257
 on RECVFROM 258
 on SEND 267
 on SENDTO 268
FLAGS parameter on macro socket interface
 on GETNAMEINFO 322
FNDELAY flag on call interface, on FCNTL 209
free addrinfo storage
 TCP/IP call function
 freeaddrinfo() 105
free the memory allocated by if_nameindex()
 TCP/IP call function
 if_freenameindex() 148
FREEADDRINFO (call) 211
FREEADDRINFO (macro) 306
freeaddrinfo() TCP/IP function 105
Functions
 ALTER 453
 COPY 456
 DEFINE 458, 459
 DELETE 463

G

gai_strerror() TCP/IP function 106
get address information
 TCP/IP call function
 getaddrinfo() 106
get IBM TCP/IP image
 TCP/IP call function
 getibmopt() 116
get identifier for calling application 110
get name information
 TCP/IP call function
 getnameinfo() 117
get socket numbers to extend beyond the default range
 TCP/IP call function
 maxdesc() 159
get the name of the peer connected to a socket
 TCP/IP call function
 getpeername() 121
GETADDRINFO (call) 211
GETADDRINFO (macro) 307
getaddrinfo() TCP/IP function 106
GETCLIENTID (call) 218
GETCLIENTID (macro) 314
getclientid()
 use in server 496, 501
getclientid() library function 110
GETHOSTBYADDR (call) 219
GETHOSTBYADDR (macro) 315
gethostbyaddr() library function 111
GETHOSTBYNAME (call) 221
GETHOSTBYNAME (macro) 317
gethostbyname() library function 113
gethostent() TCP/IP function 114
GETHOSTID (call) 223
GETHOSTID (macro) 319
gethostid() library function 115
GETHOSTNAME (call) 223
GETHOSTNAME (macro) 320
gethostname() library function 115
GETIBMOPT (call) 224
GETIBMOPT (macro) 321
getibmopt() TCP/IP function 116
GETNAMEINFO (call) 225
GETNAMEINFO (macro) 322
getnameinfo() TCP/IP function 117
getnetbyaddr() TCP/IP function 119
getnetbyname() TCP/IP function 120
getnetent() TCP/IP function 121
GETPEERNAME (call) 229
GETPEERNAME (macro) 326
getpeername() TCP/IP function 121
getprotobyname() TCP/IP function 123
getprotobynumber() TCP/IP function 123
getprotoent() TCP/IP function 124
getservbyname() TCP/IP function 125
getservbyport() TCP/IP function 125
getservent() TCP/IP function 126
GETSOCKNAME (call) 230
GETSOCKNAME (macro) 328
getsockname() library function 127
GETSOCKOPT (call) 232
GETSOCKOPT (macro) 330
getsockopt() library function 128
GIVESOCKET (call) 234
GIVESOCKET (macro) 333
givesocket()
 use in server 496, 501
givesocket() library function 133
Global VLAN 527
GPS and CICS RCF 513
gsk_free_memory() SSL function 135
gsk_get_cipher_info() SSL function 135
gsk_get_dn_by_label() SSL function 137
gsk_initialize() SSL function 137
gsk_secure_soc_close() SSL function 139
gsk_secure_soc_init() SSL function 140
gsk_secure_soc_read() SSL function 143
gsk_secure_soc_reset() SSL function 144
gsk_secure_soc_write() SSL function 145
gsk_uninitialize() SSL function 146
gsk_user_set() SSL function 146
GSKFREEMEM (call) 236
GSKFREEMEM (macro) 335
GSKGETCIPHINF (call) 237
GSKGETCIPHINF (macro) 335
GSKGETDNBYLAB (call) 238
GSKGETDNBYLAB (macro) 336
GSKINIT (call) 239
GSKINIT (macro) 337
GSKSSOCLOSE (call) 240
GSKSSOCLOSE (macro) 338
GSKSSOCINIT (call) 241
GSKSSOCINIT (macro) 339
GSKSSOCREAD 343
GSKSSOCREAD (call) 244

GSKSSOCRESET (call) 245
GSKSSOCRESET (macro) 344
GSKSSOCWRITE (call) 245
GSKSSOCWRITE (macro) 345
GSKUNINIT (call) 246
GSKUNINIT (macro) 345

H

HANDSHAKE parameter on macro socket interface
 on GSKSSOCINIT 339
hardware crypto support 33, 434, 435
HINTS parameter on call socket interface
 on GETADDRINFO 211
HINTS parameter on macro socket interface
 on GETADDRINFO 307
HiperSockets 33
HiperSockets Completion Queue 387
host address 111
host byte order
 short integer translated to 160
 translating long integer to 160
host information data set
 opening 181
host information data sets 103
host name 113
host name entry 114
HOST parameter on call socket interface
 on GETNAMEINFO 225
HOST parameter on macro socket interface
 on GETNAMEINFO 322
HOSTADDR parameter on call socket interface
 on GETHOSTBYADDR 219
HOSTADDR-SEQ parameter on call socket interface
 on EZACIC08 284
HOSTADR parameter on macro socket interface
 on GETHOSTBYADDR 315
HOSTALIAS-SEQ parameter on call socket interface
 on EZACIC08 284
 on EZACIC09 286
HOSTENT parameter on call socket interface
 on GETHOSTBYADDR 219
 on GETHOSTBYNAME 221
HOSTENT parameter on macro socket interface
 on GETHOSTBYADDR 315
 on GETHOSTBYNAME 317
HOSTENT structure interpreter parameters
 on EZACIC08 284
HOSTLEN parameter on call socket interface
 on GETNAMEINFO 225
HOSTLEN parameter on macro socket interface
 on GETNAMEINFO 322
HOW parameter on call socket interface
 on SHUTDOWN 274

HOW parameter on macro socket interface
 on SHUTDOWN 375
HTMLINST.Z 13
htonl() library function 147
htons() library function 147

I

IDENT parameter on call interface,
 INITAPI call 247
IDENT parameter on macro socket interface
 on INITAPI 346
if_freenameindex() TCP/IP function 148
if_indeXToname() TCP/IP function 148
if_nameindex() TCP/IP function 149
if_nametoindex() TCP/IP function 150
immediate=no 478
immediate=yes 478
IN-BUFFER parameter on call socket interface
 on EZACIC05 281
inet_addr() library function 150
inet_lnaof() library function 151
inet_makeaddr() library function 152
inet_netof() library function 152
inet_network() library function 153
inet_ntoa() library function 153
inet_ntop() TCP/IP function 154
inet_pton() TCP/IP function 155
InfoPrint Manager
 customizing 31
InfoPrint Manager support 29
INITAPI (call) 247
INITAPI (macro) 346
initapi() TCP/IP library function 156
initialize socket API for subtask 156
installing CICS TCP/IP 441
installing product keys 6
integer
 short translated to host byte order 160
 translating 147
 translating long to host byte order 160
 unsigned short 147
Internet address
 host 152, 153
 into network byte order 150
interval control 502
introducing socket programming 47
IOCTL (call) 249
IOCTL (macro) 348
ioctl() library function 156
IOV parameter on macro socket interface
 on READV 355
IOVCNT parameter on macro socket interface
 on READV 355
IP-ADDRESS parameter on call socket interface
 on NTOP 251
 on PTON 253
IPNCSD.Z 12
IPNCSDUP.Z 12
IPv6/VSE 41

IPv6/VSE (*continued*)
 user access key 43
IPv6/VSE Documentation 43
IPv6/VSE Installation Requirements 43
IPv6/VSE product key 43
iterative server
 socket calls in 498

K

KEYLABEL parameter on macro socket interface
 on GSKGETDNBYLAB 336
KEYRING parameter on macro socket interface
 on GSKINIT 337

L

LE enabling, definition of 49
LENGTH parameter on call socket interface
 on EZACIC04 281
 on EZACIC05 281
link, program 480
Linux Fast Path
 administrative tasks 411
 communication flow 389
 configuring LFP 397
 Linux Fast Path overview 387
 preparing to Use 393
 prerequisites 388
 restrictions 389
 starting and stopping LFP 407
 typical configuration in a z/VM environment (example of) 390
 typical configuration via HS CQ (example of) 391
Linux Fast Path (LPF) 387
listen ()
 use in server 496
LISTEN (call) 250
LISTEN (macro) 350
listen() library function 158
Listener
 control area 492
 input format 502
 monitor control table 444
 output format 504
 security module 505
 starting and stopping 502, 505
 user-written 483
listener/server call sequence 494
listener/server, socket call (general) 496
little endian 500
local network address
 into host byte order 151
long integer, translating 147

M

macro instructions for assembler programs
 ACCEPT 296
 BIND 298

macro instructions for assembler programs (*continued*)
 CANCEL 300
 CLOSE 301
 CONNECT 302
 FCNTL 305
 FREEADDRINFO 306
 GETADDRINFO 307
 GETCLIENTID 314
 GETHOSTBYADDR 315
 GETHOSTBYNAME 317
 GETHOSTID 319
 GETHOSTNAME 320
 GETIBMOPT 321
 GETNAMEINFO 322
 GETPEERNAME 326
 GETSOCKNAME 328
 GETSOCKOPT 330
 GIVESOCKET 333
 GSKFREEMEM 335
 GSKGETCIPHINF 335
 GSKGETDNBYLAB 336
 GSKINIT 337
 GSKSSOCCLOSE 338
 GSKSSOCINIT 339
 GSKSSOCREAD 343
 GSKSSOCRESET 344
 GSKSSOCWRITE 345
 GSKUNINIT 345
 INITAPI 346
 IOCTL 348
 LISTEN 350
 NTOP 351
 PTON 353
 READ 354
 READV 355
 RECV 357
 RECVFROM 358
 SELECT 360
 SELECTEX 365
 SEND 367
 SENDTO 369
 SETSOCKOPT 371
 SHUTDOWN 375
 SOCKET 376
 TAKESOCKET 379
 TASK 381
 TERMAPI 381
 WRITE 382
 WRITEV 383
macro, EZACICR 469
manual startup 475
map a network interface index to its corresponding name
 TCP/IP call function
 if_indeXToname() 148
map a network interface name to its corresponding index
 TCP/IP call function
 if_nametoindex() 150
maxdesc() TCP/IP function 159
MAXSNO parameter on call socket interface
 on INITAPI 247
MAXSNO parameter on macro socket interface
 on INITAPI 346

MAXSOC parameter on call socket interface

- on INITAPI 247
- on SELECT 261
- on SELECTEX 265

MAXSOC parameter on macro socket interface

- on INITAPI 346
- on SELECT 360
- on SELECTEX 365

messages

- receive and store in buffers 167, 168
- sending on socket 177

migration considerations 6

monitor activity on socket descriptors 161

MQSeries 84

N

NAME parameter on call socket interface

- on ACCEPT 201
- on BIND 204
- on CONNECT 207
- on GETCLIENTID 218
- on GETHOSTBYNAME 221
- on GETHOSTNAME 223
- on GETNAMEINFO 225
- on GETPEERNAME 229
- on GETSOCKNAME 230
- on GIVESOCKET 234
- on RECVFROM 258
- on SENDTO 268
- on TAKESOCKET 276

NAME parameter on macro socket interface

- on ACCEPT 296
- on BIND 298
- on CONNECT 302
- on GETHOSTBYNAME 317
- on GETHOSTNAME 320
- on GETNAMEINFO 322
- on GETPEERNAME 326
- on GETSOCKNAME 328
- on RECVFROM 358
- on SENDTO 369

name, binding to a socket 96

NAMELEN parameter on call socket interface

- on GETHOSTBYNAME 221
- on GETHOSTNAME 223
- on GETNAMEINFO 225

NAMELEN parameter on macro socket interface

- on GETHOSTBYNAME 317
- on GETHOSTNAME 320
- on GETNAMEINFO 322

NBYTE parameter on call socket interface

- on READ 254
- on RECV 257
- on RECVFROM 258
- on SEND 267
- on SENDTO 268
- on WRITE 278

NBYTE parameter on macro socket interface

- on GSKSSOCREAD 343

NBYTE parameter on macro socket interface (*continued*)

- on GSKSSOCWRITE 345
- on READ 354
- on RECV 357
- on RECVFROM 358
- on SEND 367
- on SENDTO 369
- on WRITE 382

network

- services information data set, opening 183

network byte order 147, 150, 500

network entry 120

network information data set 103

- opening 182

network name 119

network number

- getting decimal host address 153
- getting Internet host address 152

network protocol information data sets 103

network service

- by name 125
- by port 125

network services information data sets

- TCP/IP call function endservent() 104

NODE parameter on call socket interface

- on GETADDRINFO 211

NODE parameter on macro socket interface

- on GETADDRINFO 307

NODELEN parameter on call socket interface

- on GETADDRINFO 211

NODELEN parameter on macro socket interface

- on GETADDRINFO 307

NS parameter on macro socket interface

- on ACCEPT 296

ntohl() library function 160

ntohs() library function 160

NTOP (call) 251

NTOP (macro) 351

NTOP-FAMILY parameter on call socket interface

- on NTOP 251

O

OpenSSL 423, 424, 425, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436

OpenSSL include files 427

OpenSSL passed socket number 428, 430

OpenSSL Programming 427, 428, 429, 430, 431, 432, 433, 434, 435, 436

OpenSSL runtime variables 423

OpenSSL unique features 423

OPTION SYSPARM 84

options, socket 183

OPTLEN parameter on call socket interface

- on GETSOCKOPT 232
- on SETSOCKOPT 270

OPTLEN parameter on macro socket interface

- on GETSOCKOPT 330
- on SETSOCKOPT 371

OPTNAME parameter on call socket interface

- on GETSOCKOPT 232
- on SETSOCKOPT 270

OPTNAME parameter on macro socket interface

- on GETSOCKOPT 330
- on SETSOCKOPT 371

OPTVAL parameter on call socket interface

- on GETSOCKOPT 232
- on SETSOCKOPT 270

OPTVAL parameter on macro socket interface

- on GETSOCKOPT 330
- on SETSOCKOPT 371

OSA Express support 33

OSAX 527

OUT-BUFFER parameter on call socket interface

- on EZACIC04 281

P

partition startup 9

passing sockets 496

password protected key ring 431

PASSWORD.HTML 13

performance and tuning considerations 35

phase

- CUSTDEF 6
- PRODKEYS 6

PL/I programs, required statement 200

PLT 475

PLT entry 445

poll() TCP/IP function 161

port numbers

- definition 500

ports

- numbers 500

Preparation and Setup for SSL 84

PRESENTABLE-ADDRESS parameter on call socket interface

- on NTOP 251
- on PTON 253

PRESENTABLE-ADDRESS-LEN parameter on call socket interface

- on NTOP 251
- on PTON 253

PRODKEYS phase 6

product key 4, 5

program link 480

Program List Table 475

PROTO parameter on call socket interface

- on SOCKET 275

PROTO parameter on macro socket interface

- on SOCKET 376

protocol

- getting name by name 123
- getting name by number 123
- getting next entry 124

protocol (*continued*)
information data set, opening 182
PTON (call) 253
PTON (macro) 353

Q

QDIO Buffers 527

R

RDO
configure the socket interface
(EZAC) 441
READ (call) 254
READ (macro) 354
read()
use in child server 494
use in client 494
read() library function 162
reading
buffers, data stored in 164
data, and store in buffers 164
READV (call) 255
READV (macro) 355
readv() TCP/IP function 164
REASCODE parameter on macro socket
interface
on GSKSSOCINIT 339
receiving
data and store in buffers 165
messages and store in buffers 167,
168
RECV (call) 257
RECV (macro) 357
recv() library function 165
RECVFROM (call) 258
RECVFROM (macro) 358
recvfrom()
use in server 496
recvfrom() library function 167
recvmsg() TCP/IP function 168
RENAME 467
REQARG parameter on call socket
interface
on FCNTL 209
on IOCTL 249
REQARG parameter on macro socket
interface
on FCNTL 305
on IOCTL 348
RES parameter on call socket interface
on EZACIC09 286
on GETADDRINFO 211
RES parameter on macro socket interface
on GETADDRINFO 307
RES-CANONICAL-NAME parameter on
call socket interface
on EZACIC09 286
RES-NAME parameter on call socket
interface
on EZACIC09 286
RES-NEXT parameter on call socket
interface
on EZACIC09 286
resource definition in CICS 441

RETARG parameter on call socket
interface
on IOCTL 249
RETARG parameter on macro socket
interface
on IOCTL 348
RETCODE parameter on call socket
interface
on ACCEPT 201
on BIND 204, 298
on CLOSE 206, 301
on CONNECT 207
on EZACIC06 282
on EZACIC09 286
on FCNTL 209
on FREEADDRINFO 211
on GETADDRINFO 211
on GETCLIENTID 218
on GETHOSTBYADDR 219
on GETHOSTBYNAME 221
on GETHOSTID 223
on GETHOSTNAME 223
on GETNAMEINFO 225
on GETPEERNAME 229
on GETSOCKNAME 230
on GETSOCKOPT 232
on GIVESOCKET 234
on INITAPI 247
on IOCTL 249
on LISTEN 250
on NTOP 251, 351
on PTON 253, 353
on READ 254
on READV 255
on RECV 257
on RECVFROM 258
on SELECT 261
on SELECTEX 265
on SEND 267
on SENDTO 268
on SETSOCKOPT 270
on SHUTDOWN 274
on SOCKET 275
on TAKESOCKET 276
on WRITE 278
on WRITEV 279
RETCODE parameter on macro socket
interface
on ACCEPT 296
on CANCEL 300
on CONNECT 302
on FCNTL 305
on FREEADDRINFO 306
on GETADDRINFO 307
on GETCLIENTID 314
on GETHOSTBYADDR 315
on GETHOSTBYNAME 317
on GETHOSTID 319
on GETHOSTNAME 320
on GETIBMOPT 321
on GETNAMEINFO 322
on GETPEERNAME 326
on GETSOCKNAME 328
on GETSOCKOPT 330
on GIVESOCKET 333
on GSKFREEMEM 335
on GSKGETCIPHINF 335

RETCODE parameter on macro socket
interface (*continued*)
on GSKGETDNBYLAB 336
on GSKINIT 337
on GSKSSOC_CLOSE 338
on GSKSSOCINIT 339
on GSKSSOCREAD 343
on GSKSSOCRESET 344
on GSKSSOCWRITE 345
on GSKUNINIT 345
on INITAPI 346
on IOCTL 348
on LISTEN 350
on READ 354
on READV 355
on RECV 357
on RECVFROM 358
on SELECT 360
on SELECTEX 365
on SEND 367
on SENDTO 369
on SETSOCKOPT 371
on SHUTDOWN 375
on SOCKET 376
on TAKESOCKET 379
on WRITE 382
on WRITEV 383
return all network interface names and
indexes
TCP/IP call function
if_nameindex() 149
RRETMASK parameter on call socket
interface
on SELECT 261
on SELECTEX 265
RRETMASK parameter on macro interface
on SELECT 360
on SELECTEX 365
RSNDMSK parameter on call socket
interface
on SELECT 261
on SELECTEX 265
RSNDMSK parameter on macro interface
on SELECT 360
on SELECTEX 365

S

S, defines socket descriptor on call socket
interface
on ACCEPT 201
on BIND 204
on CLOSE 206
on CONNECT 207
on FCNTL 209
on GETPEERNAME 229
on GETSOCKNAME 230
on GETSOCKOPT 232
on GIVESOCKET 234
on IOCTL 249
on LISTEN 250
on READ 254
on READV 255
on RECV 257
on RECVFROM 258
on SEND 267
on SENDTO 268

- S, defines socket descriptor on call socket interface *(continued)*
 - on SETSOCKOPT 270
 - on SHUTDOWN 274
 - on WRITE 278
 - on WRITEV 279
- S, defines socket descriptor on macro socket interface
 - on ACCEPT 296
 - on CLOSE 301
 - on CONNECT 302
 - on FCNTL 305
 - on GETPEERNAME 326
 - on GETSOCKNAME 328
 - on GETSOCKOPT 330
 - on GIVESOCKET 333
 - on GSKSSOCINIT 339
 - on IOCTL 348
 - on LISTEN 350
 - on READ 354
 - on READV 355
 - on RECV 357
 - on RECVFROM 358
 - on SEND 367
 - on SENDTO 369
 - on SETSOCKOPT 371
 - on SHUTDOWN 375
 - on WRITE 382
 - onWRITEV 383
- SECLEVEL parameter on macro socket interface
 - on GSKGETCIPHINF 335
- SECTYPE parameter on macro socket interface
 - on GSKINIT 337
 - on GSKSSOCINIT 339
- secure socket
 - close connection 139
 - initialize connection 140
 - initialize environment 137
 - provide callback routines 146
 - query information 135
 - receive data 143
 - refresh parameters 144
 - remove current settings 146
 - send data 145
- Security Manager 23, 43
- security module 505
- SELECB parameter on call socket interface
 - on SELECTEX 265
- SELECB parameter on macro socket interface
 - on SELECTEX 365
- SELECT (call) 261
- SELECT (macro) 360
- select()
 - use in server 496
- select() library function 170
- select() TCP/IP function 170
- SELECTEX (call) 265
- SELECTEX (macro) 365
- selectex() library function 174
- selectex() TCP/IP function 174
- SEND (call) 267
- SEND (macro) 367
- send() library function 176
- sendmsg() TCP/IP function 177
- SENDTO (call) 268
- SENDTO (macro) 369
- sendto() TCP/IP function 179
- server
 - incoming client requests 158
 - socket calls in child server 494
 - socket calls in concurrent server 496
 - socket calls in iterative server 498
- SERVIC parameter on call socket interface
 - on GETADDRINFO 211
 - on GETNAMEINFO 225
- SERVICE parameter on macro socket interface
 - on GETADDRINFO 307
 - on GETNAMEINFO 322
- SERVLN parameter on call socket interface
 - on GETADDRINFO 211
 - on GETNAMEINFO 225
- SERVLN parameter on macro socket interface
 - on GETADDRINFO 307
 - on GETNAMEINFO 322
- set IBM TCP/IP image
 - TCP/IP call function
 - setibmopt() 181
- sethostent() TCP/IP function 181
- setibmopt() TCP/IP function 181
- setnetent() TCP/IP function 182
- setprotoent() TCP/IP function 182
- setservent() TCP/IP function 183
- SETSOCKOPT (call) 270
- SETSOCKOPT (macro) 371
- setsockopt() TCP/IP function 183
- shutdown
 - duplex connection 189
- SHUTDOWN (call) 274
- SHUTDOWN (macro) 375
- shutdown, immediate 478
- shutdown, manual 475
- shutdown() TCP/IP function 189
- SKREAD parameter on macro socket interface
 - on GSKSSOCINIT 339
- SKWRITE parameter on macro socket interface
 - on GSKSSOCINIT 339
- socket
 - acquire from another program 193
 - creating 190
 - creating a pair 193
 - data, sending on 179
 - data, writing 196
 - datagrams, sending on 176
 - descriptor in AF_INET domain 96
 - getting name 127
 - make available 133
 - messages, sending on 177
 - operating characteristics,
 - specifying 156
 - options, getting 128
 - options, setting 183
 - pairs, creating 193
 - peer connected to 121
 - send data on 176, 179
 - send messages on 177
- socket *(continued)*
 - shutdown 189
 - writing data on 196
- SOCKET (call) 275
- SOCKET (macro) 376
- socket()
 - use in client 494
 - use in server 496
- socket() TCP/IP function 190
- socketpair() TCP/IP function 193
- sockets
 - passing 496
- SOCRECV parameter on call socket interface
 - on TAKESOCKET 276
- SOCRECV parameter on macro socket interface
 - on TAKESOCKET 379
- SOCTYPE parameter on call socket interface
 - on SOCKET 275
- SOCTYPE parameter on macro interface
 - on SOCKET 376
- specify cipher suites 433
- specify the key ring 430
- SRCADDR parameter on macro socket interface
 - on NTOP 351
 - on PTON 353
- SSL for VSE
 - preparation andsSetup 84
 - usable / not usable with 5
- SSL function
 - gsk_free_memory() 135
 - gsk_get_cipher_info() 135
 - gsk_get_dn_by_label() 137
 - gsk_initialize() 137
 - gsk_secure_soc_close() 139
 - gsk_secure_soc_init() 140
 - gsk_secure_soc_read() 143
 - gsk_secure_soc_reset() 144
 - gsk_secure_soc_write() 145
 - gsk_uninitialize() 146
 - gsk_user_set() 146
- SSL implementation 423
- SSOCDATA parameter on macro socket interface
 - on GSKSSOCCLOSE 338
 - on GSKSSOCREAD 343
 - on GSKSSOCRESET 344
 - on GSKSSOCWRITE 345
- startup
 - automatic 475
 - manually 475
 - program link 480
- startup member TCPSTART.Z 9
- STORAGE parameter on macro socket interface
 - on TASK 381
- storage protection machines 441, 442
- SUBTASK parameter on call socket interface
 - on INITAPI 247
- SUBTASK parameter on macro socket interface
 - on INITAPI 346
- support, environmental 483

supported cipher suites 432
supported RSA key lengths 433, 434
switching between GSK and OpenSSL
 socket calls 430
SYSPARM 84

T

TAKESOCKET (call) 276
TAKESOCKET (macro) 379
takesocket()
 use in child server 494, 501
takesocket() TCP/IP function 193
TASK (macro) 381
task control 502
TASK parameter on call socket interface
 on GETCLIENTID 218
 on GIVESOCKET 234
 on TAKESOCKET 276
TASK parameter on macro socket
 interface
 on ACCEPT 296
 on BIND 298
 on CANCEL 300
 on CLOSE 301
 on CONNECT 302
 on FCNTL 305
 on GETCLIENTID 314
 on GETHOSTBYADDR 315
 on GETHOSTBYNAME 317
 on GETHOSTID 319
 on GETHOSTNAME 320
 on GETIBMOPT 321
 on GETPEERNAME 326
 on GETSOCKNAME 328
 on GETSOCKOPT 330
 on GIVESOCKET 333
 on INITAPI 346
 on IOCTL 348
 on LISTEN 350
 on READ 354
 on READV 355
 on RECV 357
 on RECVMFROM 358
 on SELECT 360
 on SEND 367
 on SENDTO 369
 on SETSOCKOPT 371
 on SOCKET 376
 on TAKESOCKET 379
 on WRITE 382
 on WRITEV 383
TCP/IP call function
 endhostent() 103
 endnetent() 103
 endprotoent() 103
 fcntl() 104
 gethostent() 114
 getnetbyaddr() 119
 getnetbyname() 120
 getnetent() 121
 getprotobyname() 123
 getprotobynumber() 123
 getprotoent() 124
 getservbyname() 125
 getservbyport() 125
 getservent() 126

TCP/IP call function (*continued*)

 initapi() 156
 poll() 161
 readv() 164
 recvmsg() 168
 select() 170
 selectex() 174
 sendmsg() 177
 sendto() 179
 sethostent() 181
 setnetent() 182
 setprotoent() 182
 setservent() 183
 setsockopt() 183
 shutdown() 189
 socket() 190
 socketpair() 193
 takesocket() 193
 termapi() 194
 write() 195
 writev() 196
TCP/IP for VSE/ESA
 callable functions 85
 demonstration mode 4
 migration considerations 6
 partition startup 9
TCPM td queue 443
TCPSTART.Z, startup member 9
TERMAPI (call) 278
TERMAPI (macro) 381
termapi() TCP/IP function 194
TIMEOUT parameter on call socket
 interface
 on SELECT 261
 on SELECTEX 265
TIMEOUT parameter on macro socket
 interface
 on SELECT 360
 on SELECTEX 365
TOKEN parameter on call interface
 on EZACIC06 282
transaction identifier 502
transactions, defining in CICS 441
transferring print files 29
transient data 443
type parameter 446
 TYPE=CICS 446
 TYPE=INITIAL 446
 TYPE=LISTENER 446

U

UEEXIT parameter on macro socket
 interface
 on INITAPI 346
unsigned short integer 147
use of HOSTENT structure interpreter,
 EZACIC08 284
utility programs 280
 EZACIC04 281
 EZACIC05 281
 EZACIC06 282
 EZACIC08 284
 EZACIC09 286

V

V3CIPHER parameter on macro socket
 interface
 on GSKSSOCINIT 339
V3CIPHSEL parameter on macro socket
 interface
 on GSKSSOCINIT 339
V3TIMEOUT parameter on macro socket
 interface
 on GSKINIT 337
VIOLATED.HTML 13
vlan support 527
VSAM cache file 469
VSAMCAT usage 515

W

WRETMASK parameter on call socket
 interface
 on SELECT 261
 on SELECTEX 265
WRETMASK parameter on macro socket
 interface
 on SELECT 360
 on SELECTEX 365
WRITE (call) 278
WRITE (macro) 382
write()
 use in child server 494
 use in client 494
write() TCP/IP function 195
WRITEV (call) 279
WRITEV (macro) 383
writev() TCP/IP function 196
writing
 data on sockets 196
WSNDMSK parameter on call socket
 interface
 on SELECT 261
 on SELECTEX 265
WSNDMSK parameter on macro socket
 interface
 on SELECT 360
 on SELECTEX 365

Z

z/OS ssl api 436
z/VM guest configuration 418

Glossary

This glossary includes terms and definitions for IBM z/VSE.

The following cross-references are used in this glossary:

1. See refers the reader from a term to a preferred synonym, or from an acronym or abbreviation to the defined full form.
2. See also refers the reader to a related or contrasting term.

To view glossaries for other IBM products, go to www.ibm.com/software/globalization/terminology.

A

Access Control Logging and Reporting. An IBM licensed program to log all attempts of access to protected data and to print selected formatted reports on such attempts.

access control table (DTSECTAB). A table that is used by the system to verify a user's right to access a certain resource.

access list. A table in which each entry specifies an address space or data space that a program can reference.

access method. A program, that is, a set of commands (macros) to define files or addresses and to move data to and from them; for example VSE/VSAM or VTAM.

account file. A disk file that is maintained by VSE/POWER containing accounting information that is generated by VSE/POWER and the programs running under VSE/POWER.

addressing mode (AMODE). A program attribute that refers to the address length that a program is prepared to handle on entry. Addresses can be either 24 bits, 31 bits, or 64 bits in length. In 24 bit addressing mode, the processor treats all virtual addresses as 24-bit values; in 31 bit addressing mode, the processor treats all virtual addresses as 31-bit values and in 64-bit addressing mode, the processor treats all virtual addresses as 64-bit values. Programs with an addressing mode of ANY can receive control in either 24 bit or 31 bit addressing mode. 64 bit addressing mode cannot be used as program attribute.

administration console. In z/VSE, one or more consoles that receive all system messages, except for those that are directed to one particular console.

Contrast this with the user console, which receives only those messages that are directed to it, for example messages that are issued from a job that was submitted with the request to echo its messages to that console. The operator of an administration console can reply to all outstanding messages and enter all system commands.

alternate block. On an FBA disk, a block that is designated to contain data in place of a defective block.

alternate index. In systems with VSE/VSAM, the index entries of a given base cluster that is organized by an alternate key, that is, a key other than the prime key of the base cluster. For example, a personnel file preliminary ordered by names can be indexed also by department number.

alternate library. An interactively accessible library that can be accessed from a terminal when the user of that terminal issues a connect or switch library request.

alternate track. A library, which becomes accessible from a terminal when the user of that terminal issues a connect or switch (library) request.

AMODE. Addressing mode.

APA. All points addressable.

APAR. Authorized Program Analysis Report.

appendage routine. A piece of code that is physically located in a program or subsystem, but logically and extension of a supervisor routine.

application profile. A control block in which the system stores the characteristics of one or more application programs.

application program. A program that is written for or by a user that applies directly to the user's work, such as a program that does inventory control or payroll. See also batch program and online application program.

AR/GPR. Access register and general-purpose register pair.

ASC mode. Address space control mode.

ASI (automated system initialization) procedure. A set of control statements, which specifies values for an automatic system initialization.

attention routine (AR). A routine of the system that receives control when the operator presses the Attention key. The routine sets up the console for the

input of a command, reads the command, and initiates the system service that is requested by the command.

automated system initialization (ASI). A function that allows control information for system startup to be cataloged for automatic retrieval during system startup.

autostart. A facility that starts VSE/POWER with little or no operator involvement.

auxiliary storage. Addressable storage that is not part of the processor, for example storage on a disk unit. Synonymous with external storage.

B

B-transient. A phase with a name beginning with \$\$B and running in the Logical Transient Area (LTA). Such a phase is activated by special supervisor calls.

bar. 2 GigaByte (GB) line

basic telecommunications access method (BTAM). An access method that permits read and write communication with remote devices. BTAM is not supported on z/VSE.

BIG-DASD. A subtype of Large DASD that has a capacity of more than 64 K tracks and uses up to 10017 cylinders of the disk.

block. Usually, a block consists of several records of a file that are transmitted as a unit. But if records are very large, a block can also be part of a record only. On an FBA disk, a block is a string of 512 bytes of data. See also a control block.

block group. In VSE/POWER, the basic organizational unit for fixed-block architecture (FBA) devices. Each block group consists of a number of 'units of transfer' or blocks.

C

CA splitting. Is the host part of the VSE JavaBeans, and is started using the job STARTVCS, which is placed in the reader queue during installation of z/VSE. Runs by default in dynamic class R. In VSE/VSAM, to double a control area dynamically and distribute its CIs evenly when the specified minimum of free space get used up by more data.

carriage control character. The first character of an output record (line) that is to be printed; it determines how many lines should be skipped before the next line is printed.

catalog. A directory of files and libraries, with reference to their locations. A catalog may contain other information such as the types of devices in which the files are stored, passwords, blocking factors. To store a

library member such as a phase, module, or book in a sublibrary. See also VSE/VSAM catalog.

cell pool. An area of virtual storage that is obtained by an application program and managed by the callable cell pool services. A cell pool is located in an address space or a data space and contains an anchor, at least one extent, and any number of cells of the same size.

central location. The place at which a computer system's control device, normally the systems console in the computer room, is installed.

chained sublibraries. A facility that allows sublibraries to be chained by specifying the sequence in which they must be searched for a certain library member.

chaining. A logical connection of sublibraries to be searched by the system for members of the same type (phases or object modules, for example).

channel command word (CCW). A doubleword at the location in main storage that is specified by the channel address word. One or more CCWs make up the channel program that directs data channel operations.

channel program. One or more channel command words that control a sequence of data channel operations. Execution of this sequence is initiated by a start subchannel instruction.

channel scheduler. The part of the supervisor that controls all input/output operations.

channel subsystem. A feature of z/Architecture that provides extensive additional channel (I/O) capabilities over the System z.

channel to channel attachment (CTCA). A function that allows data to be exchanged

1. Under the control of VSE/POWER between two virtual VSE machines running under VM or
2. Under the control of VTAM between two processors.

character-coded request. A request that is encoded and transmitted as a character string. Contrast with *field-formatted request*.

checkpoint.

1. A point at which information about the status of a job and the system can be recorded so that the job step can be restarted later.
2. To record such information.

CICS (Customer Information Control System). An IBM program that controls online communication between terminal users and a database. Transactions that are entered at remote terminals are processed concurrently by user-written application programs. The program includes facilities for building, using, and servicing databases.

CICS ECI. The CICS External Call Interface (ECI) is one possible requester type of the *CICS business logic interface* that is provided by the CICS Transaction Server for VSE/ESA. It is part of the CICS client and allows workstation programs to CICS function on the z/VSE host.

CICS EXCI. The EXternal CICS Interface (EXCI) is one possible requester type of the *CICS business logic interface* that is provided by the CICS Transaction Server for VSE/ESA. It allows any BSE batch application to call CICS functions.

CICS system definition data set (CSD). A VSAM KSDS cluster that contains a resource definition record for every record defined to CICS using resource definition online (RDO).

CICS Transaction Server for VSE/ESA. A z/VSE base program that controls online communication between terminal users and a database. This is the successor system to CICS/VSE.

CICS TS. CICS Transaction Server

CICS/VSE. Customer Information Control System/VSE. No longer shipped on the Extended Base Tape and no longer supported, cannot run on z/VSE 5.1 or later.

class. In VSE/POWER, a group of jobs that either come from the same input device or go to the same output device.

CMS. Conversational monitor system running on z/VM.

common library. A library that can be interactively accessed by any user of the (sub)system that owns the library.

communication adapter. A circuit card with associated software that enables a processor, controller, or other device to be connected to a network.

communication region. An area of the supervisor that is set aside for transfer of information within and between programs.

component.

1. Hardware or software that is part of a computer system.
2. A functional part of a product, which is identified by a component identifier.
3. In z/VSE, a component program such as VSE/POWER or VTAM.
4. In VSE/VSAM, a named, cataloged group of stored records, such as the data component or index component of a key-sequenced file or alternate index.

component identifier. A 12-byte alphanumeric string, uniquely defining a component to MSHP.

conditional job control. The capability of the job control program to process or to skip one or more statements that are based on a condition that is tested by the program.

connect. To authorize library access on the lowest level. A modifier such as "read" or "write" is required for the specified use of a sublibrary.

connection pooling. Introduced with an z/VSE 5.1 update to manage (reuse) connections of the z/VSE database connector in CICS TS.

connector. In the context of z/VSE, a connector provides the middleware to connect two platforms: Web Client and z/VSE host, middle-tier and z/VSE host, or Web Client and middle-tier.

connector (e-business connector). A piece of software that is provided to connect to heterogeneous environments. Most connectors communicate to non-z/VSE Java-capable platforms.

container. Is part of the JVM of application servers such as the IBM WebSphere Application Server, and facilitates the implementation of servlets, EJBs, and JSPs, by providing resource and transaction management resources. For example, an EJB developer must not code against the JVM of the application server, but instead against the interface that is provided by the container. The main role of a container is to act as an intermediary between EJBs and clients. Is the host part of the VSE JavaBeans, and is started using the job STARTVCS, which is placed in the reader queue during the installation of z/VSE. Runs by default in dynamic class R. and also to manage multiple EJB instances. After EJBs have been written, they must be stored in a container residing on an application server. The container then manages all threading and client-interactions with the EJBs, and co-ordinate connection- and instance pooling.

control interval (CI). A fixed-length area of disk storage where VSE/VSAM stores records and distributes free space. It is the unit of information that VSE/VSAM transfers to or from disk storage. For FBA it must be an integral multiple to be defined at cluster definition, of the block size.

control program. A program to schedule and supervise the running of programs in a system.

conversational monitor system (CMS). A virtual machine operating system that provides general interactive time sharing, problem solving, and program development capabilities and operates under the control of z/VM.

count-key-data (CKD) device. A disk device that store data in the record format: count field, key field, data field. The count field contains, among others, the address of the record in the format: cylinder, head (track), record number, and the length of the data field.

The key field, if present, contains the record's key or search argument. CKD disk space is allocated by tracks and cylinders. Contrast with *FBA disk device*. See also *extended count-key-data device*.

cross-partition communication control. A facility that enables VSE subsystems and user programs to communicate with each other; for example, with VSE/POWER.

cryptographic token. Usually referred to simply as a *token*, this is a device, which provides an interface for performing cryptographic functions like generating digital signatures or encrypting data.

cryptology.

1. A method for protecting information by transforming it (encrypting it) into an unreadable format, called ciphertext. Only users who possess a secret key can decipher (or decrypt) the message into plaintext.
2. The transformation of data to conceal its information content and to prevent its unauthorized use or undetected modification .

D

data block group. The smallest unit of space that can be allocated to a VSE/POWER job on the data file. This allocation is independent of any device characteristics.

data conversion descriptor file (DCDF). With a DCDF, you can convert individual fields within a record during data transfer between a PC and its host. The DCDF defines the record fields of a particular file for both, the PC and the host environment.

data import. The process of reformatting data that was used under one operating system such that it can subsequently be used under a different operating system.

Data Interfile Transfer, Testing, and Operations (DITTO) utility. An IBM program that provides file-to-file services for card I/O, tape, and disk devices. The latest version is called DITTO/ESA for VSE.

Data Language/I (DL/I). A database access language that is used with CICS.

data link. In SNA, the combination of the link connection and the link stations joining network nodes, for example, a z/Architecture channel and its associated protocols. A link is both logical and physical.

data security. The protection of data against unauthorized disclosure, transfer, modification, or destruction, whether accidental or intentional .

data set header record. In VSE/POWER abbreviated as DSHR, alias NDH or DSH. An NJE control record

either preceding output data or, in the middle of input data, indicating a change in the data format.

data space. A range of up to 2 gigabytes of contiguous virtual storage addresses that a program can directly manipulate through z/Architecture instructions. Unlike an address space, a data space can hold only user data; it does not contain shared areas, or programs. Instructions do not execute in a data space. Contrast with address space.

data terminal equipment (DTE). In SNA, the part of a data station that serves a data source, data sink, or both.

database connector. Is a function introduced with z/VSE 5.1.1, which consists of a client and server part. The client provides an API (CBCLI) to be used by applications on z/VSE, the server on any Java capable platform connects a JDBC driver that is provided by the database. Both client and server communicate via TCP/IP.

Database 2 (DB2). An IBM relational database management system.

DB2-based connector. Is a feature introduced with VSE/ESA 2.5, which includes a customized DB2 version, together with VSAM and DL/I functionality, to provide access to DB2, VSAM, and DL/I data, using DB2 Stored Procedures.

DB2 Runtime only Client edition. The Client Edition for z/VSE comes with some enhanced features and improved performance to integrate z/VSE and Linux on System z.

DB2 Stored Procedure. In the context of z/VSE, a DB2 Stored Procedure is a Language Environment (LE) program that accesses DB2 data. However, from VSE/ESA 2.5 onwards you can also access VSAM and DL/I data using a DB2 Stored Procedure. In this way, it is possible to exchange data between VSAM and DB2.

DBLK. Data block.

DCDF. Data conversion descriptor file.

deblocking. The process of making each record of a block available for processing.

dedicated (disk) device. A device that cannot be shared among users.

device address.

1. The identification of an input/output device by its device number.
2. In data communication, the identification of any device to which data can be sent or from which data can be received.

device driving system (DDS). A software system external to VSE/POWER, such as a CICS spooler or PSF, that writes spooled output to a destination device.

Device Support Facilities (DSF). An IBM supplied system control program for performing operations on disk volumes so that they can be accessed by IBM and user programs. Examples of these operations are initializing a disk volume and assigning an alternative track.

device type code. The four- or five-digit code that is used for defining an I/O device to a computer system. See also **ICKDSF**

dialog. In an interactive system, a series of related inquiries and responses similar to a conversation between two people. For z/VSE, a set of panels that can be used to complete a specific task; for example, defining a file.

dialog manager. The program component of z/VSE that provides for ease of communication between user and system.

digital signature. In computer security, encrypted data, which is appended to or part of a message, that enables a recipient to prove the identity of the sender.

Digital Signature Algorithm (DSA). The Digital Signature Algorithm is the US government-defined standard for digital signatures. The DSA digital signature is a pair of large numbers, computed using a set of rules (that is, the DSA) and a set of parameters such that the identity of the signatory and integrity of the data can be verified. The DSA provides the capability to generate and verify signatures.

directory. In z/VSE the index for the program libraries.

direct access. Accessing data on a storage device using their address and not their sequence. This is the typical access on disk devices as opposed to magnetic tapes. Contrast with *sequential access*.

disk operating system residence volume (DOSRES). The disk volume on which the system sublibrary IJSYSRS.SYSLIB is located including the programs and procedures that are required for system startup.

disk sharing. An option that lets independent computer systems use common data on shared disk devices.

disposition. A means of indicating to VSE/POWER how a job input or output entry is to be handled: according to its local disposition in the RDR/LST/PUN queue or its transmission disposition when residing in the XMT queue. A job might, for example, be deleted or kept after processing.

distribution tape. A magnetic tape that contains, for example, a preconfigured operating system like z/VSE. This tape is shipped to the customer for program installation.

DITTO/ESA for VSE. Data Interfile Transfer, Testing, and Operations utility. An IBM program that provides file-to-file services for disk, tape, and card devices.

DSF. Device Support Facilities.

DSH (R). Data set header record.

dummy device. A device address with no real I/O device behind it. Input and output for that device address are spooled on disk.

duplex. Pertaining to communication in which data can be sent and received at the same time.

DU-AL (dispatchable unit - access list). The access list that is associated with a z/VSE main task or subtask. A program uses the DU-AL associated with its task and the PASN-AL associated with its partition. See also *PASN-AL*.

dynamic class table. Defines the characteristics of dynamic partitions.

dynamic partition. A partition that is created and activated on an 'as needed' basis that does not use fixed static allocations. After processing, the occupied space is released. Dynamic partitions are grouped by class, and jobs are scheduled by class. Contrast with *static partition*.

dynamic space reclamation. A librarian function that provides for space that is freed by the deletion of a library member to become reusable automatically.

E

ECI. See *CICS ECI*.

emulation. The use of programming techniques and special machine features that permit a computer system to execute programs that are written for another system or for the use of I/O devices different from those that are available.

emulation program (EP). An IBM control program that allows a channel-attached 3705 or 3725 communication controller to emulate the functions of an IBM 2701 Data Adapter Unit, or an IBM 2703 Transmission Control.

end user.

1. A person who makes use of an application program.
2. In SNA, the ultimate source or destination of user data flowing through an SNA network. Might be an application program or a terminal operator.

Enterprise Java Bean. An EJB is a distributed bean. "Distributed" means, that one part of an EJB runs inside the JVM of a web application server, while the other part runs inside the JVM of a web browser. An EJB either represents one data row in a database (entity bean), or a connection to a remote database (session bean). Normally, both types of an EJB work together. This allows to represent and access data in a standardized way in heterogeneous environments with relational and non-relational data. See also *JavaBean*.

entry-sequenced file. A VSE/VSAM file whose records are loaded without respect to their contents and whose relative byte addresses cannot change. Records are retrieved and stored by addressed access, and new records are added to the end of the file.

Environmental Record Editing and Printing (EREP) program. A z/VSE base program that makes the data that is contained in the system record file available for further analysis.

EPI. See *CICS EPI*.

ESCON Channel (Enterprise Systems Connection Channel). A serial channel, using fiber optic cabling, that provides a high-speed connection between host and control units for I/O devices. It complies with the ESA/390 and System z I/O Interface until z114. The zEC12 processors do not support ESCON channels.

exit routine.

1. Either of two types of routines: installation exit routines or user exit routines. Synonymous with exit program.
2. See *user exit routine*.

extended addressability. The ability of a program to use 31 bit or 64 bit virtual storage in its address space or outside the address space.

extended recovery facility (XRF). In z/VSE, a feature of CICS that provides for enhanced availability of CICS by offering one CICS system as a backup of another.

External Security Manager (ESM). A priced vendor product that can provide extended functionality and flexibility that is compared to that of the Basic Security Manager (BSM), which is part of z/VSE.

F

FASTCOPY. See *VSE/Fast Copy*.

fast copy data set program (VSE/Fast Copy). See *VSE/Fast Copy*.

fast service upgrade (FSU). A service function of z/VSE for the installation of a refresh release without regenerating control information such as library control tables.

FAT-DASD. A subtype of Large DASD, it supports a device with more than 4369 cylinders (64 K tracks) up to 64 K cylinders.

FCOPY. See *VSE/Fast Copy*.

fence. A separation of one or more components or elements from the remainder of a processor complex. The separation is by logical boundaries. It allows simultaneous user operations and maintenance procedures.

fetch.

1. To locate and load a quantity of data from storage.
2. To bring a program phase into virtual storage from a sublibrary and pass control to this phase.
3. The name of the macro instruction (FETCH) used to accomplish 2. See also *loader*.

Fibre Channel Protocol (FCP). A combination of hardware and software conforming to the Fibre Channel standards and allowing system and peripheral connections via FICON and FICON Express feature cards on IBM zSeries processors. In z/VSE, zSeries FCP is employed to access industry-standard SCSI disk devices.

fragmentation (of storage). Inability to allocate unused sections (fragments) of storage in the real or virtual address range of virtual storage.

FSU. Fast service upgrade.

FULIST (Function LIST). A type of selection panel that displays a set of files and/or functions for the choice of the user.

G

generation. See *macro generation*.

generation feature. An IBM licensed program order option that is used to tailor the object code of a program to user requirements.

GETVIS space. Storage space within partition or the shared virtual area, available for dynamic allocation to programs.

guest system. A data processing system that runs under control of another (host) system. On the mainframe z/VSE can run as a guest of z/VM.

H

hard wait. The condition of a processor when all operations are suspended. System recovery from a hard wait is impossible without performing a new system startup.

hash function. A hash function is a transformation that takes a variable-size input and returns a fixed-size

string, which is called the hash value. In cryptography, the hash functions should have some additional properties:

- The hash function should be easy to compute.
- The hash function is one way; that is, it is impossible to calculate the 'inverse' function.
- The hash function is collision-free; that is, it is impossible that different input leads to the same hash value.

hash value. The fixed-sized string resulting after applying a *hash function* to a text.

High-Level Assembler for VSE. A programming language providing enhanced assembler programming support. It is a base program of z/VSE.

home interface. Provides the methods to instantiate a new EJB object, introspect an EJB, and remove an EJB instantiation., as for the remote interface is needed because the deployment tool generates the implementation class. Every Session bean's home interface must supply at least one *create()* method.

host mode. In this operating mode, a PC can access a VSE host. For programmable workstation (PWS) functions, the Move Utilities of VSE can be used.

host system. The controlling or highest level system in a data communication configuration.

host transfer file (HTF). Used by the Workstation File Transfer Support of z/VSE as an intermediate storage area for files that are sent to and from IBM personal computers.

HTTP Session. In the context of z/VSE, identifies the web-browser client that calls a servlet (in other words, identifies the connection between the client and the middle-tier platform).

I

ICCF. See *VSE/ICCF*.

ICKDSF (Device Support Facilities). A z/VSE base program that supports the installation, use, and maintenance of IBM disk devices.

include function. Retrieves a library member for inclusion in program input.

index.

1. A table that is used to locate records in an indexed sequential data set or on indexed file.
2. In, an ordered collection of pairs, each consisting of a key and a pointer, used by to sequence and locate the records of a key-sequenced data set or file; it is organized in levels of index records. See also *alternate index*.

input/output control system (IOCS). A group of IBM supplied routines that handle the transfer of data between main storage and auxiliary storage devices.

integrated communication adapter (ICA). The part of a processor where multiple lines can be connected.

integrated console. In z/VSE, the service processor console available on IBM System z server that operates as the z/VSE system console. The integrated console is typically used during IPL and for recovery purposes when no other console is available.

Interactive Computing and Control Facility (ICCF). An IBM licensed program that serves as interface, on a time-slice basis, to authorized users of terminals that are linked to the system's processor.

interactive partition. An area of virtual storage for the purpose of processing a job that was submitted interactively via VSE/ICCF.

Interactive User Communication Vehicle (IUCV). Programming support available in a VSE supervisor for operation under z/VM. The support allows users to communicate with other users or with CP in the same way they would with a non-preferred guest.

intermediate storage. Any storage device that is used to hold data temporarily before it is processed.

IOCS. Input/output control system.

IPL. Initial program load.

irrecoverable error. An error for which recovery is impossible without the use of recovery techniques external to the computer program or run.

IUCV. Interactive User Communication Vehicle.

J

JAR. Is a platform-independent file format that aggregates many files into one. Multiple applets and their requisite components (.class files, images, and sounds) can be bundled in a JAR file, and then downloaded to a web browser using a single HTTP transaction (much improving the download speed). The JAR format also supports compression, which reduces the files size (and further improves the download speed). The compression algorithm that is used is fully compatible with the ZIP algorithm. The owner of an applet can also digitally sign individual entries in a JAR file to authenticate their origin.

Java application. A Java program that runs inside the JVM of your web browser. The program's code resides on a local hard disk or on the LAN. Java applications might be large programs using graphical interfaces. Java applications have unlimited access to all your local resources.

Java bytecode. Bytecode is created when a file containing Java source language statements is compiled. The compiled Java code or "bytecode" is similar to any program module or file that is ready to be executed (run on a computer so that instructions are performed one at a time). However, the instructions in the bytecode are really instructions to the *Java Virtual Machine*. Instead of being interpreted one instruction at a time, bytecode is instead recompiled for each operating-system platform using a just-in-time (JIT) compiler. Usually, this enables the Java program to run faster. Bytecode is contained in binary files that have the suffix **CLASS**.

Java servlet. See *servlet*.

JHR. Job header record.

job accounting interface. A function that accumulates accounting information for each job step, to be used for charging the users of the system, for planning new applications, and for supervising system operation more efficiently.

job accounting table. An area in the supervisor where accounting information is accumulated for the user.

job catalog. A catalog made available for a job by means of the file name IJSYSUC in the respective DLBL statement.

job entry control language (JECL). A control language that allows the programmer to specify how VSE/POWER should handle a job.

job step. In 1 of a group of related programs complete with the JCL statements necessary for a particular run. Every job step is identified in the job stream by an EXEC statement under one JOB statement for the whole job.

job trailer record (JTR). As VSE/POWER parameter JTR, alias NJT. An NJE control record terminating a job entry in the input or output queue and providing accounting information.

K

key. In VSE/VSAM, one or several characters that are taken from a certain field (key field) in data records for identification and sequence of index entries or of the records themselves.

key sequence. The collating sequence either of records themselves or of their keys in the index or both. The key sequence is alphanumeric.

key-sequenced file. A VSE/VSAM file whose records are loaded in key sequence and controlled by an index. Records are retrieved and stored by keyed access or by addressed access, and new records are inserted in the file in key sequence.

KSDS. Key-sequenced data sets. See *key-sequenced file*.

L

label.

1. An identification record for a tape, disk, or diskette volume or for a file on such a volume.
2. In assembly language programming, a named instruction that is generally used for branching.

label information area. An area on a disk to store label information that is read from job control statements or commands. Synonymous with *label area*.

Language Environment for z/VSE. An IBM software product that is the implementation of Language Environment on the VSE platform.

language translator. A general term for any assembler, compiler, or other routine that accepts statements in one language and produces equivalent statements in another language.

Large DASD. A DASD device that

1. Has a capacity exceeding 64 K tracks and
2. Does not have VSAM space created prior to VSE/ESA 2.6 that is owned by a catalog.

LE/VSE. Short form of Language Environment for z/VSE.

librarian. The set of programs that maintains, services, and organizes the system and private libraries.

library block. A block of data that is stored in a sublibrary.

library directory. The index that enables the system to locate a certain sublibrary of the accessed library.

library member. The smallest unit of a data that can be stored in and retrieved from a sublibrary.

line commands. In VSE/ICCF, special commands to change the declaration of individual lines on your screen. You can copy, move, or delete a line declaration, for example.

linkage editor. A program that is used to create a phase (executable code) from one or more independently translated object modules, from one or more existing phases, or from both. In creating the phase, the linkage editor resolves cross-references among the modules and phases available as input. The program can catalog the newly built phases.

linkage stack. An area of protected storage that the system gives to a program to save status information for a branch and stack or a stacking program call.

link station. In SNA, the combination of hardware and software that allows a node to attach to and provide control for a link.

loader. A routine, commonly a computer program, that reads data or a program into processor storage. See also *relocating loader*.

local shared resources (LSR). A VSE/VSAM option that is activated by three extra macros to share control blocks among files.

lock file. In a shared disk environment under VSE, a system file on disk that is used by the sharing systems to control their access to shared data.

logical partition. In LPAR mode, a subset of the server unit hardware that is defined to support the operation of a system control program.

logical record. A user record, normally pertaining to a single subject and processed by data management as a unit. Contrast with *physical record*, which may be larger or smaller.

logical unit (LU).

1. A name that is used in programming to represent an I/O device address. *physical unit (PU)*, *system services control point (SSCP)*, *primary logical unit (PLU)*, and *secondary logical unit (SLU)*.
2. In SNA, a port through which a user accesses the SNA network,
 - a. To communicate with another user and
 - b. To access the functions of the SSCP. An LU can support at least two sessions. One with an SSCP and one with another LU and might be capable of supporting many sessions with other LUs.

logical unit name. In programming, a name that is used to represent the address of an input/output unit.

logical unit 6.2. A SNA/SDLC protocol for communication between programs in a distributed processing environment. LU 6.2 is characterized by

1. A peer relationship between session partners,
2. Efficient utilization of a session for multiple transactions,
3. Comprehensive end-to-end error processing, and
4. A generic Application Programming Interface (API) consisting of structured verbs that are mapped into a product implementation.

logons interpret interpret routine. In VTAM, an installation exit routine, which is associated with an interpret table entry, that translates logon information. It also verifies the logon.

LPAR mode. Logically partitioned mode. The CP mode that is available on the Configuration (CONFIG) frame when the PR/SM feature is installed. LPAR mode allows the operator to allocate the hardware resources of the processor unit among several logical partitions.

M

macro definition. A set of statements and instructions that defines the name of, format of, and conditions for generating a sequence of assembler statements and machine instructions from a single source statement.

macro expansion. See *macro generation*

macro generation. An assembler operation by which a macro instruction gets replaced in the program by the statements of its definition. It takes place before assembly. Synonymous with *macro expansion*.

macro (instruction).

1. In assembler programming, a user-invented assembler statement that causes the assembler to process a set of statements that are defined previously in the macro definition.
2. A sequence of VSE/ICCF commands that are defined to cause a sequence of certain actions to be performed in response to one request.

maintain system history program (MSHP). A program that is used for automating and controlling various installation, tailoring, and service activities for a VSE system.

main task. The main program within a partition in a multiprogramming environment.

master console. In z/VSE, one or more consoles that receive all system messages, except for those that are directed to one particular console. Contrast this with the *user console*, which receives only those messages that are specifically directed to it, for example messages that are issued from a job that was submitted with the request to echo its messages to that console. The operator of a master console can reply to all outstanding messages and enter all system commands.

maximum (max) CA. A unit of allocation equivalent to the maximum control area size on a count-key-data or fixed-block device. On a CKD device, the max CA is equal to one cylinder.

memory object. Chunk of virtual storage that is allocated above the bar (2 GB) to be created with the IARV64 macro.

message. In VSE, a communication that is sent from a program to the operator or user. It can appear on a console, a display terminal or on a printout.

MSHP. See maintain system history program.

multitasking. Concurrent running of one main task and one or several subtasks in the same partition.

MVS. Multiple Virtual Storage. Implies MVS/390, MVS/XA, MVS/ESA, and the MVS element of the z/OS (OS/390) operating system.

N

NetView. A z/VSE optional program that is used to monitor a network, manage it, and diagnose its problems.

network address. In SNA, an address, consisting of subarea and element fields, that identifies a link, link station, or NAU. Subarea nodes use network addresses; peripheral nodes use local addresses. The boundary function in the subarea node to which a peripheral node is attached transforms local addresses to network addresses and vice versa. See also *network name*.

network addressable unit (NAU). In SNA, a logical unit, a physical unit, or a system services control point. It is the origin or the destination of information that is transmitted by the path control network. Each NAU has a network address that represents it to the path control network. See also *network name*, *network address*.

Network Control Program (NCP). An IBM licensed program that provides communication controller support for single-domain, multiple-domain, and interconnected network capability. Its full name is ACF/NCP.

network definition table (NDT). In VSE/POWER networking, the table where every node in the network is listed.

network name.

1. In SNA, the symbolic identifier by which users refer to a NAU, link, or link station. See also *network address*.
2. In a multiple-domain network, the name of the APPL statement defining a VTAM application program. This is its network name, which must be unique across domains.

node.

1. In SNA, an end point of a link or junction common to several links in a network. Nodes can be distributed to host processors, communication controllers, cluster controllers, or terminals. Nodes can vary in routing and other functional capabilities.
2. In VTAM, a point in a network that is defined by a symbolic name. Synonymous with *network node*. See *major node and minor node*.

node type. In SNA, a designation of a node according to the protocols it supports and the network addressable units (NAUs) it can contain.

O

object module (program). A program unit that is the output of an assembler or compiler and is input to a linkage editor.

online application program. An interactive program that is used at display stations. When active, it waits for data. Once input arrives, it processes it and send a response to the display station or to another device.

operator command. A statement to a control program, issued via a console or terminal. It causes the control program to provide requested information, alter normal operations, initiate new operations, or end existing operations.

optional licensed program. An IBM licensed program that a user can install on VSE by way of available installation-assist support.

output parameter text block (OPTB). in VSE/POWER's spool-access support, information that is contained in an output queue record if a * \$\$ LST or * \$\$ PUN statement includes any user-defined keywords that have been defined for autostart.

P

page data set (PDS). One or more extents of disk storage in which pages are stored when they are not needed in processor storage.

page fixing. Marking a page so that it is held in processor storage until explicitly released. Until then, it cannot be paged out.

page I/O. Page-in and page-out operations.

page pool. The set of page frames available for paging virtual-mode programs.

panel. The complete set of information that is shown in a single display on terminal screen. Scrolling back and forth through panels like turning manual pages. See also *selection panel*.

partition balancing. A z/VSE facility that allows the user to specify that two or more or all partitions of the system should receive about the same amount of time on the processor.

PASN-AL (primary address space number - access list). The access list that is associated with a partition. A program uses the PASN-AL associated with its partition and the DU-AL associated with its task (work unit). See also *DU-AL*.

Each partition has its own unique PASN-AL. All programs running in this partition can access data spaces through the PASN-AL. Thus a program can create a data space, add an entry for it in the PASN-AL, and obtain the ALET that indexes the entry. By passing the ALET to other programs in the partition, the program can share the data space with other programs running in the same partition.

PDS. Page data sets.

phase. The smallest complete unit of executable code that can be loaded into virtual storage.

physical record. The amount of data that is transferred to or from auxiliary storage. Synonymous with *block*.

PNET. Programming support available with VSE/POWER; it provides for the transmission of selected jobs, operator commands, messages, and program output between the nodes of a network.

POWER. See *VSE/POWER*.

pregenerated operating system. An operating system such as z/VSE that is shipped by IBM mainly in object code. IBM defines such key characteristics as the size of the main control program, the organization, and size of libraries, and required system areas on disk. The customer does not have to generate an operating system.

preventive service. The installation of one or more PTFs on a VSE system to avoid the occurrence of anticipated problems.

primary address space. In z/VSE, the address space where a partition is executed. A program in primary mode fetches data from the primary address space.

primary library. A VSE library owned and directly accessible by a certain terminal user.

printer/keyboard mode. Refers to 1050 or 3215 console mode (device dependent).

Print Services Facility (PSF)/VSE. An access method that provides support for the advanced function printers.

private area. The virtual space between the shared area (24 bit) and shared area (31 bit), where (private) partitions are allocated. Its maximum size can be defined during IPL. See also *shared area*.

private memory object. Memory object (chunk of virtual storage) that is allocated above the 2 GB line (bar) only accessible by the partition that created it.

private partition. Any of the system's partitions that are not defined as shared. See also *shared partition*.

production library.

1. In a pre-generated operating system (or product), the program library that contains the object code for this system (or product).
2. A library that contains data that is needed for normal processing. Contrast with *test library*.

programmer logical unit. A logical unit available primarily for user-written programs. See also *logical unit name*.

program temporary fix (PTF). A solution or by-pass of one or more problems that are documented in APARs. PTFs are distributed to IBM customers for preventive service to a current release of a program.

PSF/VSE. Print Services Facility/VSE.

PTF. See *Program temporary fix*.

Q

Queue Control Area (QCA). In VSE/POWER, an area of the data file, which might contain:

- Extended checkpoint information
- Control information for a shared environment.

queue file. A direct-access file that is maintained by VSE/POWER that holds control information for the spooling of job input and job output.

R

random processing. The treatment of data without respect to its location on disk storage, and in an arbitrary sequence that is governed by the input against which it is to be processed.

real address area. In z/VSE, processor storage to be accessed with dynamic address translation (DAT) off

real address space. The address space whose addresses map one-to-one to the addresses in processor storage.

real mode. In VSE, a processing mode in which a program might not be paged. Contrast with *virtual mode*.

recovery management support (RMS). System routines that gather information about hardware failures and that initiate a retry of an operation that failed because of processor, I/O device, or channel errors.

refresh release. An upgraded VSE system with the latest level of maintenance for a release.

relative-record file. A VSE/VSAM file whose records are loaded into fixed-length slots and accessed by the relative-record numbers of these slots.

release upgrade. Use of the FSU functions to install a new release of z/VSE.

relocatable module. A library member of the type object. It consists of one or more control sections cataloged as one member.

relocating loader. A function that modifies addresses of a phase, if necessary, and loads the phase for running into the partition that is selected by the user.

remote interface. In the context of z/VSE, the remote interface allows a client to make method calls to an EJB although the EJB is on a remote z/VSE host. The container uses the remote interface to create client-side stubs and server-side proxy objects to handle incoming method calls from a client to an EJB.

remote procedure call (RPC).

1. A facility that a client uses to request the execution of a procedure call from a server. This facility includes a library of procedures and an external data representation.
2. A client request to service provider in another node.

residency mode (RMODE). A program attribute that refers to the location where a program is expected to reside in virtual storage. RMODE 24 indicates that the program must reside in the 24-bit addressable area (below 16 megabytes), RMODE ANY indicates that the program can reside anywhere in 31-bit addressable storage (above or below 16 megabytes).

REXX/VSE. A general-purpose programming language, which is particularly suitable for command procedures, rapid batch program development, prototyping, and personal utilities.

RMS. Recovery management support.

RPG II. A commercially oriented programming language that is specifically designed for writing application programs that are intended for business data processing.

S

SAM ESDS file. A SAM file that is managed in VSE/VSAM space, so it can be accessed by both SAM and VSE/VSAM macros.

SCP. System control programming.

SDL. System directory list.

search chain. The order in which chained sublibraries are searched for the retrieval of a certain library member of a specified type.

second-level directory. A table in the SVA containing the highest phase names that are found on the directory tracks of the system sublibrary.

Secure Sockets Layer (SSL). A security protocol that allows the client to authenticate the server and all data and requests to be encrypted. SSL was developed by Netscape Communications Corp. and RSA Data Security, Inc..

segmentation. In VSE/POWER, a facility that breaks list or punch output of a program into segments so that printing or punching can start before this program has finished generating such output.

selection panel. A displayed list of items from which a user can make a selection. Synonymous with *menu*.

sense. Determine, on request or automatically, the status or the characteristics of a certain I/O or communication device.

sequential access method (SAM). A data access method that writes to and reads from an I/O device record after record (or block after block). On request, the support performs device control operations such as line spacing or page ejects on a printer or skip some tape marks on a tape drive.

service node. Within the VSE unattended node support, a processor that is used to install and test a master VSE system, which is copied for distribution to the unattended nodes. Also, program fixes are first applied at the service node and then sent to the unattended nodes.

service program. A computer program that performs function in support of the system. See with *utility program*.

service refresh. A form of service containing the current version of all software. Also referred to as a *system refresh*.

service unit. One or more PTFs on disk or tape (cartridge).

shared area. In z/VSE, shared areas (24 bit) contain the Supervisor areas and SVA (24 bit) and shared areas (31 bit) the SVA (31 bit). Shared areas (24 bit) are at the beginning of the address space (below 16 MB), shared area (31 bit) at the end (below 2 GB).

shared disk option. An option that lets independent computer systems use common data on shared disk devices.

shared memory objects. Chunks of virtual storage allocated above the 2 GB line (bar), that can be shared among partitions.

shared partition. In z/VSE, a partition that is allocated for a program (VSE/POWER, for example) that provides services and communicates with programs in other partitions of the system's virtual address spaces. In most cases shared partitions are no longer required.

shared spooling. A function that permits the VSE/POWER account file, data file, and queue file to be shared among several computer systems with VSE/POWER.

shared virtual area (SVA). In z/VSE, a high address area that contains a list system directory list (SDL) of frequently used phases, resident programs that are shared between partitions, and an area for system support.

SIT (System Initialization Table). A table in CICS that contains data used the system initialization process. In particular, the SIT can identify (by suffix characters) the version of CICS system control programs and CICS tables that you have specified and that are to be loaded.

skeleton. A set of control statements, instructions, or both, that requires user-specific information to be inserted before it can be submitted for processing.

socksified. See *socks-enabled*.

Socks-enabled. Pertaining to TCP/IP software, or to a specific TCP/IP application, that understands the *socks protocol*. "Socksified" is a slang term for socks-enabled.

socks protocol. A protocol that enables an application in a secure network to communicate through a firewall via a *socks server*.

socks server. A circuit-level gateway that provides a secure one-way connection through a firewall to server applications in a nonsecure network.

source member. A library member containing source statements in any of the programming languages that are supported by VSE.

split. To double a specific unit of storage space (CI or CA) dynamically when the specified minimum of free space gets used up by new records.

spooling. The use of disk storage as buffer storage to reduce processing delays when transferring data between peripheral equipment and the processor of a computer. In z/VSE, this is done under the control of VSE/POWER.

Spool Access Protection. An optional feature of VSE/POWER that restricts individual spool file entry access to user IDs that have been authenticated by having performed a security logon.

spool file.

1. A file that contains output data that is saved for later processing.
2. One of three VSE/POWER files on disk: queue file, data file, and account file.

stacked tape. An IBM supplied product-shipment tape containing the code of several licensed programs.

standard label. A fixed-format record that identifies a volume of data such as a tape reel or a file that is part of a volume of data.

stand-alone program. A program that runs independently of (not controlled by) the VSE system.

startup. The process of performing IPL of the operating system and of getting all subsystems and applications programs ready for operation.

start option. In VTAM, a user-specified or IBM specified option that determines conditions for the time a VTAM system is operating. Start options can be predefined or specified when VTAM is started.

static partition. A partition, which is defined at IPL time and occupying a defined amount of virtual storage that remains constant. See also *dynamic partition*.

storage director. An independent component of a storage control unit; it performs all of the functions of a storage control unit and thus provides one access path to the disk devices that are attached to it. A storage control unit has two storage directors.

storage fragmentation. Inability to allocate unused sections (fragments) of storage in the real or virtual address range of virtual storage.

suballocated file. A VSE/VSAM file that occupies a portion of an already defined data space. The data space might contain other files. See also *unique file*.

sublibrary. In VSE, a subdivision of a library. Members can only be accessed in a sublibrary.

sublibrary directory. An index for the system to locate a member in the accessed sublibrary.

submit. A VSE/POWER function that passes a job to the system for processing.

SVA. See shared virtual area.

Synchronous DataLink Control (SDLC). A discipline for managing synchronous, code-transparent, serial-by-bit information transfer over a link connection. Transmission exchanges might be duplex or half-duplex over switched or non-switched links. The configuration of the link connection might be point-to-point, multipoint, or loop.

SYSRES. See system residence volume.

system control programming (SCP). IBM supplied, non-licensed program fundamental to the operation of a system or to its service or both.

system directory list (SDL). A list containing directory entries of frequently used phases and of all phases resident in the SVA. The list resides in the SVA.

system file. In z/VSE, a file that is used by the operating system, for example, the hardcopy file, the recorder file, the page data set.

System Initialization Table (SIT). A table in CICS that contains data that is used by the system initialization process. In particular, the SIT can identify (by suffix characters) the version of CICS system control programs and CICS tables that you have specified and that are to be loaded.

system recorder file. The file that is used to record hardware reliability data. Synonymous with *recorder file*.

system refresh. See *service refresh*.

system refresh release. See *refresh release*.

system residence file (SYSRES). The z/VSE system sublibrary IJSYSRS.SYSLIB that contains the operating system. It is stored on the system residence volume DORSES.

system residence volume (SYSRES). The disk volume on which the system sublibrary is stored and from which the hardware retrieves the initial program load routine for system startup.

system sublibrary. The sublibrary that contains the operating system. It is stored on the system residence volume (SYSRES).

T

task management. The functions of a control program that control the use, by tasks, of the processor and other resources (except for input/output devices).

time event scheduling support. In VSE/POWER, the time event scheduling support offers the possibility to schedule jobs for processing in a partition at a predefined time once repetitively. The time event scheduling operands of the *\$\$ JOB statement are used to specify the wanted scheduling time.

track group. In VSE/POWER, the basic organizational unit of a file for CKD devices.

track hold. A function that protects a track that is being updated by one program from being accessed by another program.

transaction.

1. In a batch or remote batch entry, a job or job step. 2. In CICS TS, one or more application programs that can be used by a display station operator. A given transaction can be used concurrently from one or more display stations. The execution of a transaction for a certain operator is also referred to as a task.
2. A given task can relate only to one operator.

transient area. An area within the control program that is used to provide high-priority system services on demand.

Turbo Dispatcher. A facility of z/VSE that allows to use multiprocessor systems (also called CEC: Central Electronic Complexes). Each CPU within such a CEC has accesses to be shared virtual areas of z/VSE: supervisor, shared areas (24 bit), and shared areas (31 bit). The CPUs have equal rights, which means that any

CPU might receive interrupts and work units are not dedicated to any specific CPU.

U

UCB. Universal character set buffer.

universal character set buffer (UCB). A buffer to hold UCS information.

UCS. Universal character set.

user console. In z/VSE, a console that receives only those system messages that are specifically directed to it. These are, for example, messages that are issued from a job that was submitted with the request to echo its messages to that console. Contrast with *master console*.

user exit. A programming service that is provided by an IBM software product that can be requested during the execution of an application program for the service of transferring control back to the application program upon the later occurrence of a user-specified event.

V

variable-length relative-record data set (VRDS). A relative-record data set with variable-length records. See also *relative-record data set*.

variable-length relative-record file. A VSE/VSAM relative-record file with variable-length records. See also *relative-record file*.

VIO. See virtual I/O area.

virtual address. An address that refers to a location in virtual storage. It is translated by the system to a processor storage address when the information stored at the virtual address is to be used.

virtual addressability extension (VAE). A storage management support that allows to use multiple virtual address spaces.

virtual address space. A subdivision of the virtual address area (virtual storage) available to the user for the allocation of private, nonshared partitions.

virtual disk. A range of up to 2 gigabytes of contiguous virtual storage addresses that a program can use as workspace. Although the virtual disk exists in storage, it appears as a real FBA disk device to the user program. All I/O operations that are directed to a virtual disk are intercepted and the data to be written to, or read from, the disk is moved to or from a data space.

Like a data space, a virtual disk can hold only user data; it does not contain shared areas, system data, or programs. Unlike an address space or a data space,

data is not directly addressable on a virtual disk. To manipulate data on a virtual disk, the program must perform I/O operations.

Starting with z/VSE 5.2, a virtual disk may be defined in a shared memory object.

virtual I/O area (VIO). An extension of the page data set; used by the system as intermediate storage, primarily for control data.

virtual mode. The operating mode of a program, where the virtual storage of the program can be paged, if not enough processor (real) storage is available to back the virtual storage.

virtual partition. In VSE, a division of the dynamic area of virtual storage.

virtual storage. Addressable space image for the user from which instructions and data are mapped into processor storage locations.

virtual tape. In z/VSE, a virtual tape is a file (or data set) containing a tape image. You can read from or write to a virtual tape in the same way as if it were a physical tape. A virtual tape can be:

- A VSE/VSAM ESDS file on the z/VSE local system.
- A remote file on the server side; for example, a Linux, UNIX, or Windows file. To access such a remote virtual tape, a TCP/IP connection is required between z/VSE and the remote system.

volume ID. The volume serial number, which is a number in a volume label that is assigned when a volume is prepared for use by the system.

VRDS. Variable-length relative-record data sets. See *variable-length relative record file*.

VSAM. See *VSE/VSAM*.

VSE (Virtual Storage Extended). A system that consists of a basic operating system and any IBM supplied and user-written programs that are required to meet the data processing needs of a user. VSE and hardware it controls form a complete computing system. Its current version is called z/VSE.

VSE/Advanced Functions. As part of VSE Central Functions, a base program of z/VSE. A program that provides basic system control and includes the supervisor and system programs such as the Librarian and the Linkage Editor.

VSE Connector Server. Is the host part of the VSE JavaBeans, and is started using the job STARTVCS, which is placed in the reader queue during installation of z/VSE. Runs by default in dynamic class R.

VSE/DITTO (VSE/Data Interfile Transfer, Testing, and Operations Utility). An IBM licensed program that provides file-to-file services for disk, tape, and card devices.

VSE/ESA (Virtual Storage Extended/Enterprise Systems Architecture). The predecessor system of z/VSE.

VSE/Fast Copy. A utility program for fast copy data operations from disk to disk and dump/restore operations via an intermediate dump file on magnetic tape or disk.

VSE/FCOPY (VSE/Fast Copy Data Set program). An IBM licensed program for fast copy data operations from disk to disk and dump/restore operations via an intermediate dump file on magnetic tape or disk. There is also a stand-alone version: the FASTCOPY utility.

VSE/ICCF (VSE/Interactive Computing and Control Facility). An IBM licensed program that serves as interface, on a time-slice basis, to authorized users of terminals that are linked to the system's processor.

VSE/ICCF library. A file that is composed of smaller files (libraries) including system and user data, which can be accessed under the control of VSE/ICCF.

VSE JavaBeans. Are JavaBeans that allow access to all VSE-based file systems (VSE/VSAM, Librarian, and VSE/ICCF), submit jobs, and access the z/VSE operator console. The class library is contained in the *VSEConnector.jar* archive. See also *JavaBeans*.

VSE library. A collection of programs in various forms and storage dumps stored on disk. The form of a program is indicated by its member type such as source code, object module, phase, or procedure. A VSE library consists of at least one sublibrary, which can contain any type of member.

VSE/POWER. An IBM licensed program that is primarily used to spool input and output. The program's networking functions enable a VSE system to exchange files with or run jobs on another remote processor.

VSE/VSAM (VSE/Virtual Storage Access Method). An IBM access method for direct or sequential processing of fixed and variable length records on disk devices.

VSE/VSAM catalog. A file containing extensive file and volume information that VSE/VSAM requires to locate files, to allocate and deallocate storage space, to verify the authorization of a program or an operator to gain access to a file, and to accumulate use statistics for files.

VSE/VSAM managed space. A user-defined space on disk that is placed under the control of VSE/VSAM.

W

wait for run subqueue. In VSE/POWER, a subqueue of the reader queue with dispatchable jobs ordered in execution start time sequence.

wait state. The condition of a processor when all operations are suspended. System recovery from a hard wait is impossible without performing a new system startup. See *hard wait*.

Workstation File Transfer Support. Enables the exchange of data between IBM Personal Computers (PCs) linked to a z/VSE host system where the data is kept in intermediate storage. PC users can retrieve that data and work with it independently of z/VSE.

work file. A file that is used for temporary storage of data being processed.

Numerics

24-bit addressing. Provides addressability for address spaces up to 16 megabytes.

31-bit addressing. Provides addressability for address spaces up to 2 gigabytes.

64-bit addressing. Provides addressability for address spaces up to 2 gigabytes and above.

Readers' Comments — We'd Like to Hear from You

IBM z/VSE
z/VSE TCP/IP Support
Version 5

Publication No. SC34-2640-03

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your 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. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Submit your comments using one of these channels:

- Send your comments to the address on the reverse side of this form.
- Send a fax to the following number: +49-7031-163456
- Send your comments via email to: s390id@de.ibm.com
- Send a note from the web page: <http://www.ibm.com/systems/z/os/zvse/>

If you would like a response from IBM, please fill in the following information:

Name

Address

Company or Organization

Phone No.

Email address



Fold and Tape

Please do not staple

Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Deutschland Research & Development GmbH
Department 3282
Schoenaicher Strasse 220
71032 Boeblingen
Germany

Fold and Tape

Please do not staple

Fold and Tape



Product Number: 5609-ZV5

SC34-2640-03

