TCP/IP for VSE/ESA

TCP/IP for VSE/ESA – IBM Program Setup and Supplementary Information

SC33-6601-11

TCP/IP for VSE/ESA

TCP/IP for VSE/ESA – IBM Program Setup and Supplementary Information

SC33-6601-11

Note!

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

This edition applies to Version 1 Release 5 of IBM TCP/IP for VSE/ESA, Program Number 5686-A04, and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces SC33-8601-10.

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 3248
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.

Contents

Figures

Tables

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 Web sites 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 Web sites specifically mentioned in this publication or accessed through an IBM Web site 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, ibm.com, Lotus, and Notes are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at `www.ibm.com/legal/copytrade.shtml`

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

Microsoft and Windows 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 is a trademark of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

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.

z/VSE Information

z/VSE information is accessible using screen readers with the BookServer/Library Server versions of z/VSE books in the Internet library at:

```
http://www.ibm.com/systems/z/os/zos/bkserv/vse.html
```

One exception is command syntax that is published in railroad track format. If required, screen-readable copies of z/VSE books with that syntax information are separately available in HTML zipped file form upon request to s390id@de.ibm.com.

About this book

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:

```
►►——KEYWOrd——————————————————————————————————————————————————►◄
```

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

Symbols

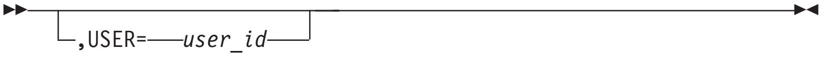
You **must** code these symbols exactly as they appear in the syntax diagram

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

For example:

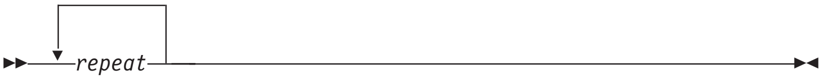
```
* $$ LST
```

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

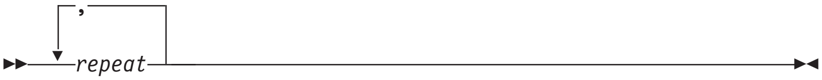


Here you must code USER= as shown and supply an ID for user_id. You may, of course, enter USER in lowercase, but you must not change it otherwise.

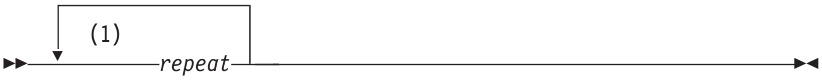
Repetition
An arrow returning to the left means that the item can be repeated.



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



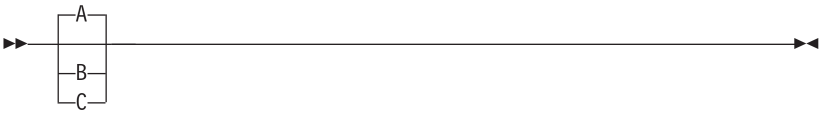
A footnote (1) by the arrow references a limit that tells how many times the item can be repeated.



Notes:

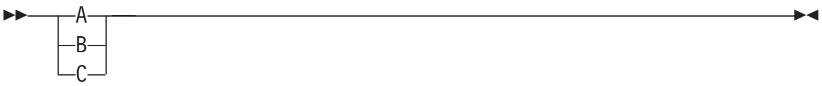
1 Specify *repeat* up to 5 times.

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.

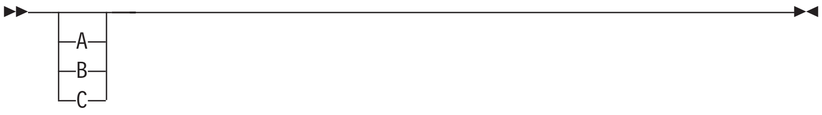
Required Choices
When two or more items are in a stack and one of them is on the line, you **must** specify one item. For example:



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:

```
* $$ EOJ
```

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

Summary of Changes

This manual has been updated to reflect enhancements and changes that are implemented with z/VSE 4.2.2. It also includes terminology, maintenance, and editorial changes.

z/VSE is the successor to VSE/ESA. However, the names of many features and programs related to z/VSE remain unchanged (such as IBM COBOL for VSE/ESA, or TCP/IP for VSE/ESA).

This manual has been updated to reflect the following new functions, especially with support for IPv6:

FREEADDRINFO
The FREEADDRINFO function frees all the address information structures returned by GETADDRINFO in the RES parameter. For details see EZASOKET "FREEADDRINFO" on page 203 and EZASMI "FREEADDRINFO" on page 318.

GETADDRINFO
The GETADDRINFO function translates names into a set of socket addresses and other associated information. For details see EZASOKET "GETADDRINFO" on page 204 and EZASMI "GETADDRINFO" on page 319.

GETNAMEINFO
The GETNAMEINFO function returns the node name and service location of a socket address that is specified in the macro. For details see EZASOKET "GETNAMEINFO" on page 219 and EZASMI "GETNAMEINFO" on page 335.

NTOP The NTOP function converts an IP address from its numeric binary form into a standard text presentation form. For details see EZASOKET "NTOP" on page 250 and EZASMI "NTOP" on page 369.

PTON The PTON function converts an IP address from its standard text presentation form to its numeric binary form. For details see EZASOKET "PTON" on page 252 and EZASMI "PTON" on page 371.

The NAME parameter on the following calls now supports the address of an IPv6 SOCKADDR structure:

ACCEPT
For details see EZASOKET "ACCEPT" on page 190 and EZASMI "ACCEPT" on page 304.

BIND For details see EZASOKET "BIND" on page 193 and EZASMI "BIND" on page 307.

CONNECT
For details see EZASOKET "CONNECT" on page 198 and EZASMI "CONNECT" on page 313.

GETPEERNAME
For details see EZASOKET "GETPEERNAME" on page 223 and EZASMI "GETPEERNAME" on page 339.

GETSOCKNAME
For details see EZASOKET “GETSOCKNAME” on page 225 and EZASMI “GETSOCKNAME” on page 341.

RECVFROM
For details see EZASOKET “RECVFROM” on page 260 and EZASMI “RECVFROM” on page 379.

SENDTO
For details see EZASOKET “SENDTO” on page 273 and EZASMI “SENDTO” on page 391.

The AF parameter of the SOCKET call accepts the new INET6 value to support the IPv6 Internet protocol. For details see EZASOKET “SOCKET” on page 280.

The DOMAIN field of the following functions can now contain AF_INET6 values to support the IPv6 Internet protocol:

GETCLIENTID
For details see EZASOKET “GETCLIENTID” on page 211 and EZASMI “GETCLIENTID” on page 326.

GIVESOCKET
For details see EZASOKET “GIVESOCKET” on page 229 and EZASMI “GIVESOCKET” on page 346.

TAKESOCKET
For details see EZASOKET “TAKESOCKET” on page 282 and EZASMI “TAKESOCKET” on page 402.

If you are coding in COBOL, EZACIC09 interprets the ADDRINFO structure returned by GETADDRINFO. For details see “EZACIC09” on page 296.

The Appendix “TCP/IP for VSE/ESA (5686-A04) History” that contained the APAR history has been removed. For a list of current PTFs and APARs go to the z/VSE Homepage at: http://www.ibm.com/systems/z/os/zvse/support/tcpip.html

Part 1. Using TCP/IP for VSE/ESA

Chapter 1. Important Considerations - Read this First!

This manual primarily describes how to setup and use the TCP/IP for VSE/ESA (5685-A04) program. IBM provides extensions that can be used with this program. They are described in detail and include the LE/VSE C socket interface, the EZASMI macro interface, the EZASOKET call interface, 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 for VSE/ESA program. Check the individual function descriptions for such exceptions.

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 only available on the z/VSE DVD-ROM (SK3T-8348) and from the Internet at http://www.ibm.com/systems/z/os/zvse/.

The TCP/IP for VSE/ESA product documentation consists of 6 books with the original product description from Connectivity Systems Inc., the provider of the TCP/IP for VSE product, plus one manual describing the setup of the TCP/IP for VSE/ESA product IBM is providing – this book.

The 7 books are as follows:

- *TCP/IP for VSE/ESA - IBM Program Setup and Supplementary Information* (this book)
- *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*

All books are available on DVD-ROM (SK3T-8348) and from the Internet in PDF format only. You can use the Adobe Acrobat Reader to view and print these books. If you do not already have Acrobat Reader installed, or if you need information on installing and using Acrobat Reader, see the Adobe Web site at ***http://www.adobe.com***.
The SSL setup used in z/VSE is described in the *z/VSE e-business Connectors User's Guide*.

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

Additional information on the z/VSE system can be found at the z/VSE Home Page at http://www.ibm.com/systems/z/os/zvse/.

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-ROM (SK3T-8348) in PDF format only. This DVD-ROM contains 6 original books 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 below and further in this manual.
- TCP/IP for VSE/ESA from IBM is preinstalled in the PRD1.BASE 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*, SC33-6642.
  - 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 20MB 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, will be 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 PRD1.BASE 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=PRD1.BASE
// 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=(PRD1.BASE)
// 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
/&
```

Notes:

1. In the preceding example, PRD2.CONFIG is the name of the library into which TCP/IP for VSE/ESA's configuration data is being installed.
2. Once you have completed a license agreement for the software, you will replace the string shown in the example with a real product key. The keys that appear here in this example merely are 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 intended 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 decided 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 PRD1.BASE 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 PRD1.BASE. 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 PRD1.BASE library. Do not keep any personalized information, for example the key and customer definition or the TCP/IP startup member in PRD2.CONFIG or any other sublibrary except PRD1.BASE. 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 define the default partition F7 to TCP/IP for VSE/ESA. A default partition startup member TCPSTART.Z can be found in PRD1.BASE. You may adjust it according to your configuration and put it into the VSE/POWER RDR queue using the DTRIINIT utility (see the following example). You may 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 PRD1.BASE
LOAD TCPSTART.Z
/*
/&
* $$ EOJ
```

TCPSTART.Z looks as follows:

```
* $$ JOB JNM=TCPSTART,CLASS=7,DISP=K
* $$ LST CLASS=A,DISP=D
// JOB TCPIP
// LIBDEF *,SEARCH=(PRD2.CONFIG,PRD1.BASE,PRD2.SCEEBASE)
// EXEC PROC=DTRICCF
// SETPFIX LIMIT=(400K,2100K)
// EXEC IPNET,SIZE=IPNET,PARM='ID=00,INIT=IPINIT00',DSPACE=2M
/&
* $$ EOJ
```

Notes

- In the above example PRD1.BASE 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).
- The PRD2.SCEEBASE library contains the LE/VSE C runtime environment and is necessary for IBM product key verification.
- Be sure that the LIBDEF statement specifies "*". If you specify "phase", TCP/IP for VSE/ESA will be unable to locate the initialization member IPINIT00.L.
- 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*,**PRD1.BASE**)

  With z/VSE TCP/IP for VSE/ESA is preinstalled in PRD1.BASE, the same library where CICS resides. Therefore in general no change is required.
- Define the Programs and Transactions to your CICS which should be used with TCP/IP. If you are not using CICS RDO, you have to include all entries from DFHPPTIP.A and DFHPCTIP.A into your existing PPT and PCT, reassemble both tables, and COLD start your CICS/VSE. If you are using CICS RDO, you have two possibilities of implementing:
  - Run the job DEFINE to migrate the tables to the CSD file (see example below)
  - Run the job IPNCSDUP to define the definitions to the CSD file (see page 16).

  The library members DFHPPTIP.A and DFHPCTIP.A are shipped with the TCP/IP for VSE/ESA product in library PRD1.BASE.

Example for CICS/VSE 2.3 only

```
* $$ JOB JNM=DEFINE,CLASS=0,DISP=D
* $$ LST CLASS=A,DISP=D
// JOB DEFINE
// LIBDEF *,SEARCH=(PRD1.BASE)
// EXEC DFHCSDUP
MIGRATE TABLE(DFHPPTIP) TOGROUP(TEMP)
COPY GROUP(TEMP) TO(TCPIP) REPLACE
DELETE ALL GROUP(TEMP)
MIGRATE TABLE(DFHPCTIP) TOGROUP(TEMP)
COPY GROUP(TEMP) TO(TCPIP) REPLACE
DELETE ALL GROUP(TEMP)
ADD GROUP(TCPIP) LIST(VSELIST)
/*
/&
* $$ EOJ
```

If you are using the PC based dialog *TCP/IP for VSE/ESA Configuration* this job is created automatically. This dialog also creates an OS/2, respectively DOS, batch file to upload such output files to the host.

Notes

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

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

Figure 1. DFHPPTIP — CICS Processing Program Table

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

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

Figure 2. DFHPCTIP — CICS Transaction Table (Part 1 of 2)

```
DFHPCT   TYPE=ENTRY,              Entry                       *
      TRANSID=lpr,                Transaction Name            *
      PROGRAM=CLIENT01,           Program Idenitification     *
      RSL=PUBLIC                  Public
DFHPCT   TYPE=ENTRY,              Entry                       *
      TRANSID=PING,               Transaction Name            *
      PROGRAM=CLIENT01,           Program Idenitification     *
      RSL=PUBLIC                  Public
DFHPCT   TYPE=ENTRY,              Entry                       *
      TRANSID=ping,               Transaction Name            *
      PROGRAM=CLIENT01,           Program Idenitification     *
      RSL=PUBLIC                  Public
DFHPCT   TYPE=ENTRY,              Entry                       *
      TRANSID=TCPC,               Transaction Name            *
      PROGRAM=CLIENT01,           Program Idenitification     *
      RSL=PUBLIC                  Public
DFHPCT   TYPE=ENTRY,              Entry                       *
      TRANSID=TCPW,               Transaction Name            *
      PROGRAM=CLIENT01,           Program Idenitification     *
      RSL=PUBLIC                  Public
DFHPCT   TYPE=ENTRY,              Entry                       *
      TRANSID=TCPR,               Transaction Name            *
      PROGRAM=CLIENT01,           Program Idenitification     *
      RSL=PUBLIC                  Public
DFHPCT   TYPE=FINAL
END
```

Figure 2. DFHPCTIP — CICS Transaction Table (Part 2 of 2)

Example for CICS/TS 1.1 and CICS/VSE 2.3

CICS/TS for VSE/ESA 1.1 does not support CICS table definitions any longer. So you always have to use one of the two methods for the definitions to the CSD described above for CICS/VSE 2.3. A member IPNCSD.Z is available which uses command definitions instead of macro definitions. Additionally a member IPNCSDUP.Z is available to use IPNCSD.Z. IPNCSDUP.Z looks as follows:

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

Notes

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

```
*---------------------------------------------------------------------*
*    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(emai)   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(teln)   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                                *
*---------------------------------------------------------------------*
```

Figure 3. IPNCSD.Z shipped with TCP/IP for VSE/ESA

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 PRD1.BASE 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 PRD1.BASE
LOAD HTMLINST.Z
/*
/&
* $$ EOJ
```

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, "Important Considerations - Read this First!" 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 11 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 20.

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 will be 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 may use the IUI-based Configuration Dialog described in "Configuring TCP/IP Using the IUI-based Configuration Dialog" on page 20.

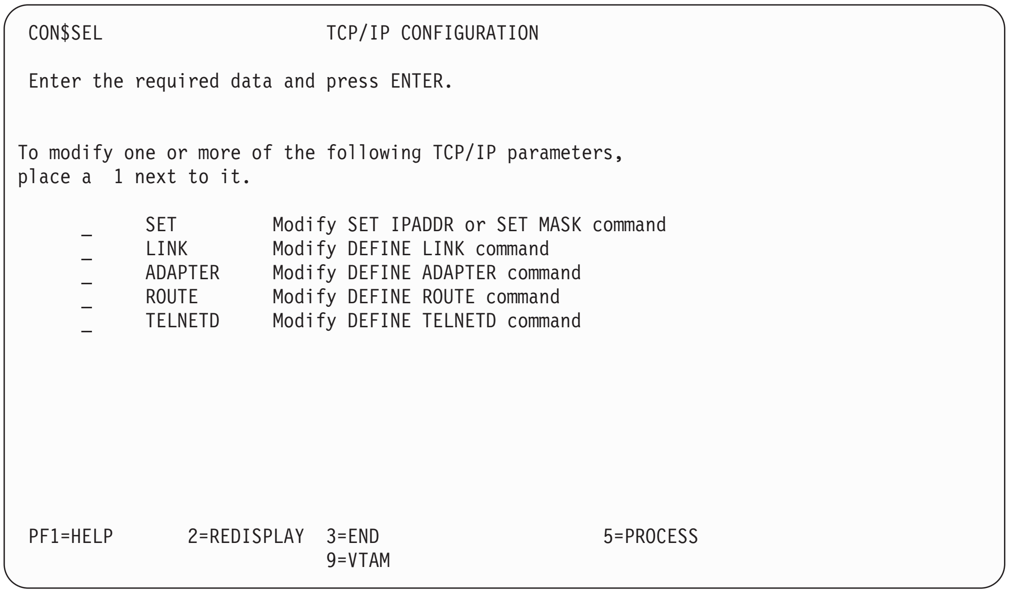
A default member IPINIT00.L is shipped with z/VSE in VSE library PRD1.BASE. It contains many configuration parameters set to their default values. You will find this a good starting point in developing your own configuration.

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

Configuring TCP/IP Using the IUI-based Configuration Dialog

The Interactive Interface has been enhanced with the **TCP/IP Configuration** panel (Fast Path**245**) 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 PRD1.BASE 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 4) below. Press PF5 (PROCESS) in this panel after you have entered the required values.

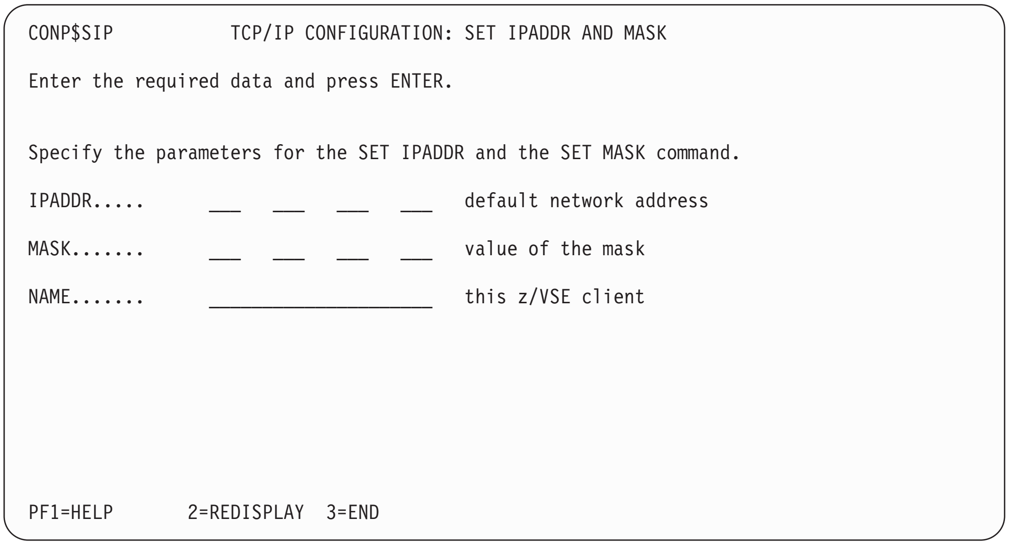
If TELNET daemons are added, we recommend to press PF9=VTAM (see page 25 for more information). In this case, member TCPAPP00.B is updated.



```
 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 4. TCP/IP Configuration Panel CON$SEL

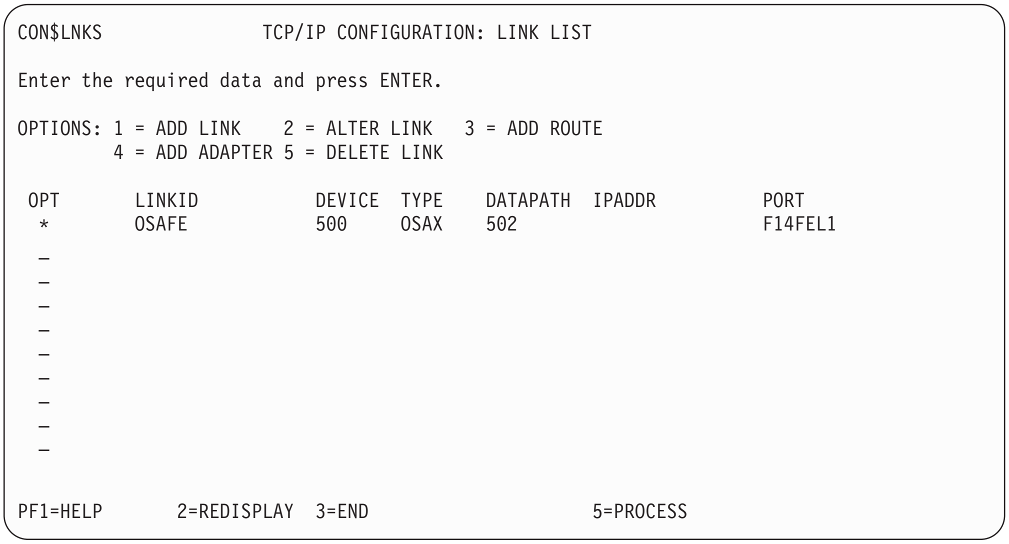
If SET is selected, the ***TCP/IP Configuration: Set IPADDR and MASK*** panel CON$SIP is displayed. The panel shows the already defined values for the SET IPADDR and SET MASK statement and for the DEFINE NAME statement for this z/VSE client. These values can be changed.



```
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 5. TCP/IP Configuration Panel: Set IPADDR and MASK

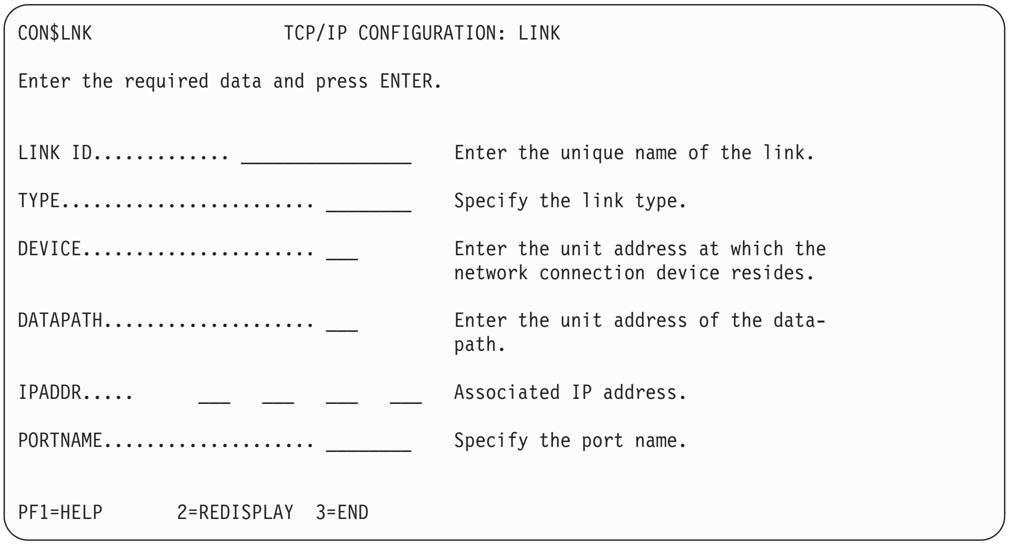
If LINK is selected in the ***TCP/IP Configuration*** panel (Figure 4 on page 20), a list of all defined links is displayed in the ***TCP/IP Configuration: Link List*** panel.



```
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 6. TCP/IP Configuration Panel: Link List

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

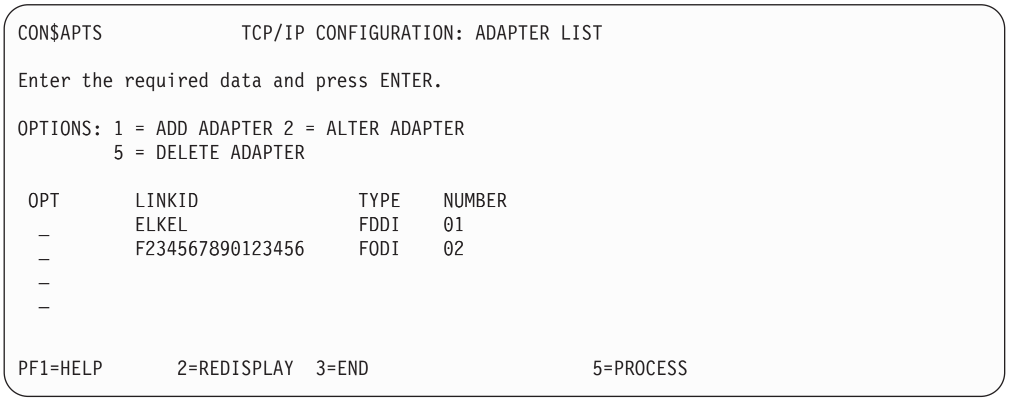


```
CON$LNK                 TCP/IP CONFIGURATION: LINK

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.

PORTNAME................... ________    Specify the port name.


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

Figure 7. TCP/IP Configuration: Link

If the input was correct, the dialog goes back to panel ***TCP/IP Configuration: LINK LIST*** (Figure 6 on page 21).

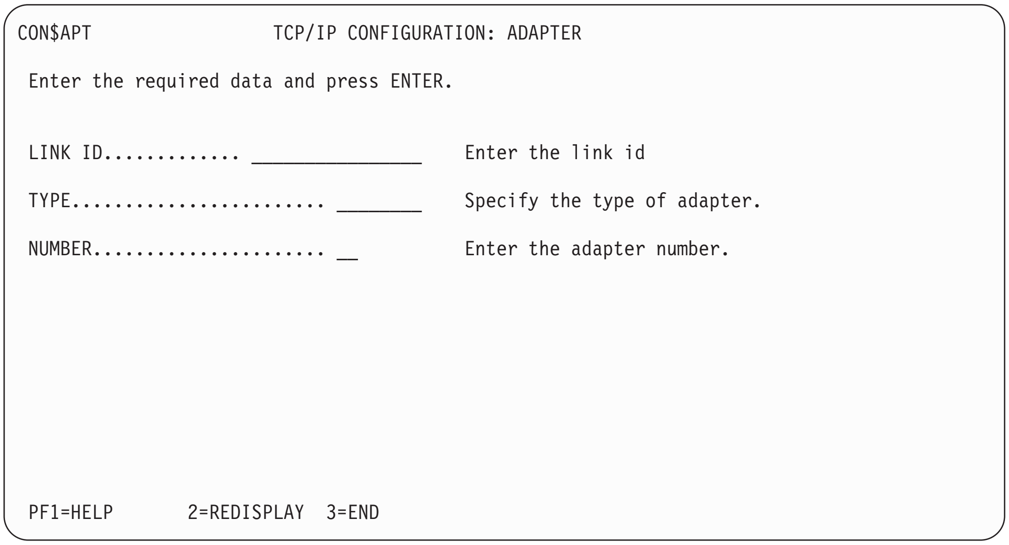
Select ADAPTER on panel ***TCP/IP Configuration*** (Figure 4 on page 20) to get the following panel (note that this panel is only possible for links of type OSA or 3172):



```
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 8. TCP/IP Configuration Panel: Adapter List

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

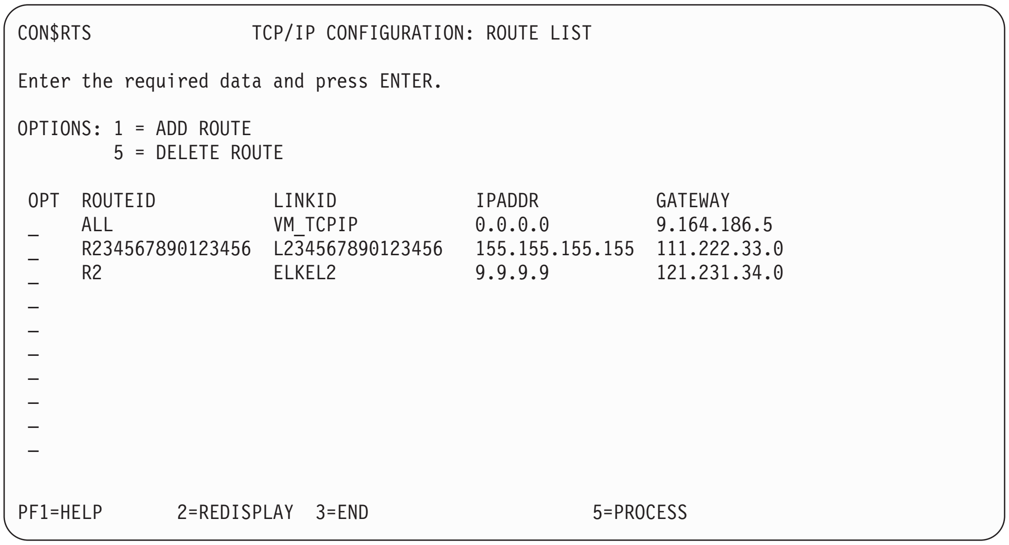


```
CON$APT                 TCP/IP CONFIGURATION: ADAPTER

 Enter the required data and press ENTER.


 LINK ID............ ________________    Enter the link id

 TYPE....................... ________    Specify the type of adapter.

 NUMBER..................... __          Enter the adapter number.










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

Figure 9. TCP/IP Configuration Panel: Adapter

If the input was correct, the dialog goes back to panel ***TCP/IP Configuration: CON$SEL*** (Figure 4 on page 20).

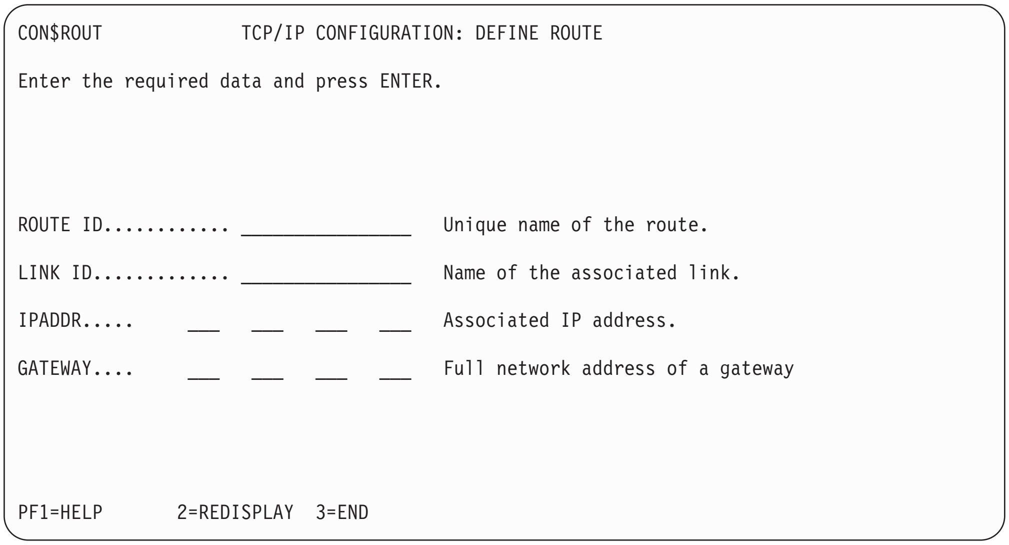
You can now select ROUTE on panel ***TCP/IP Configuration*** (Figure 4 on page 20) and get the following panel:



```
CON$RTS               TCP/IP CONFIGURATION: ROUTE LIST

Enter the required data and press ENTER.

OPTIONS: 1 = ADD ROUTE
         5 = DELETE ROUTE

 OPT  ROUTEID           LINKID            IPADDR           GATEWAY
 _    ALL               VM_TCPIP          0.0.0.0          9.164.186.5
 _    R234567890123456  L234567890123456  155.155.155.155  111.222.33.0
 _    R2                ELKEL2            9.9.9.9          121.231.34.0
 _
 _
 _
 _
 _
 _
 _


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

Figure 10. TCP/IP Configuration Panel: Route List

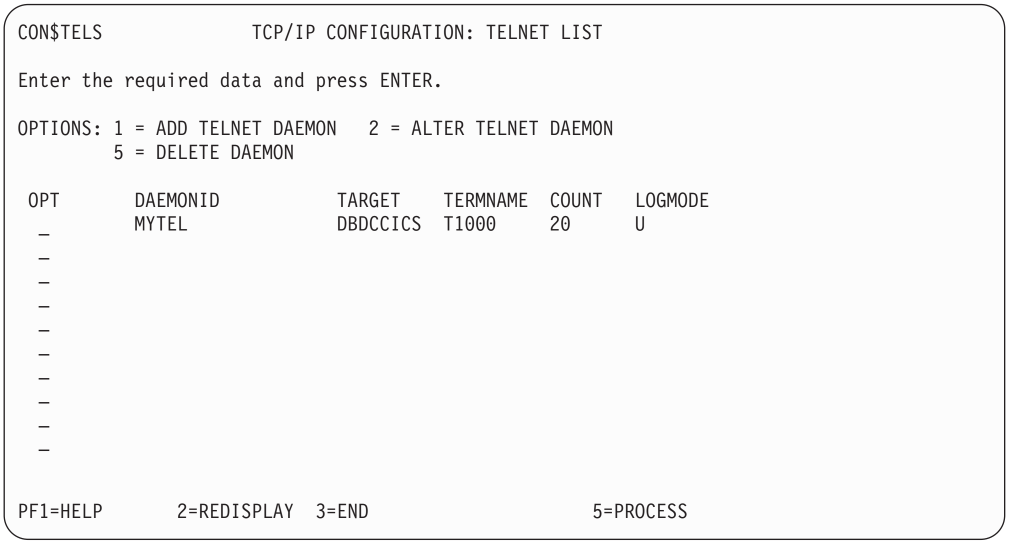
Enter Option 1=ADD ROUTE to get the DEFINE ROUTE panel. You can get the same panel by entering 3=ADD ROUTE in panel ***TCP/IP Configuration: LINK LIST*** (Figure 6 on page 21). In this case the LINKID has already been specified.



```
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 11. TCP/IP Configuration Panel: Define Route

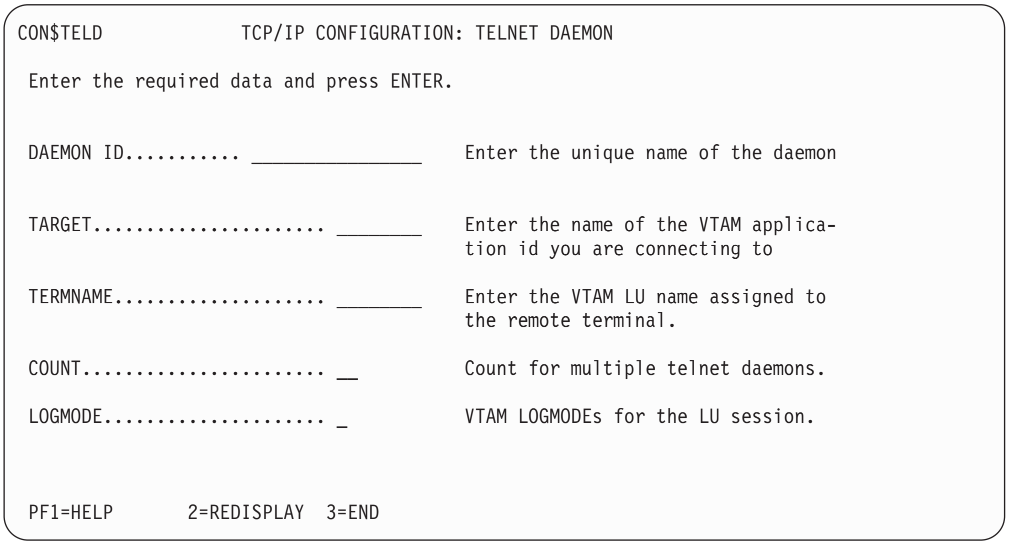
If TELNET DAEMON is selected on the ***TCP/IP Configuration*** panel (Figure 4 on page 20), the TELNET LIST panel is displayed.



```
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 12. 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 13. TCP/IP Configuration Panel: TELNET DAEMON

When a TELNET daemon is added, the TERMname must be defined as a VTAM application. Therefor 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 4 on page 20) to activate the updates. Then a job will be 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*).

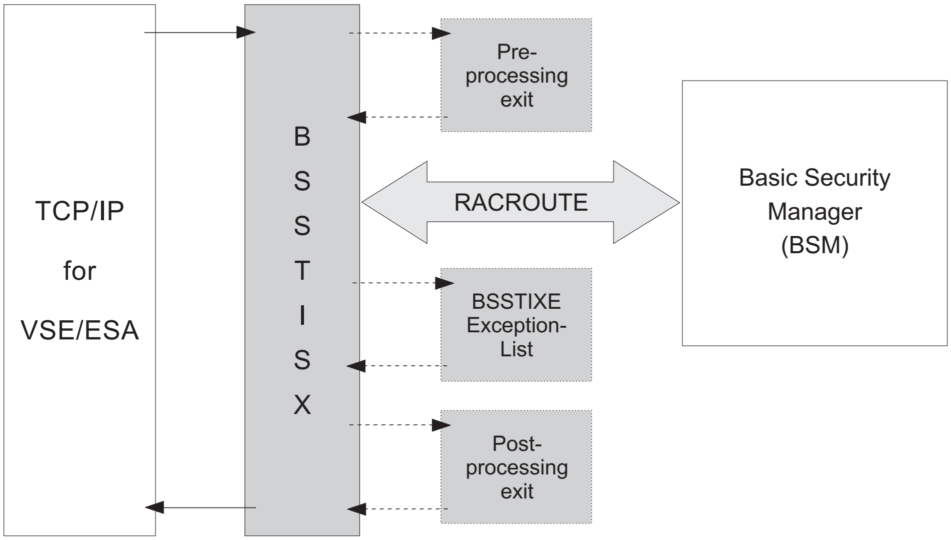
The security concept of TCP/IP for VSE/ESA is described in the *TCP/IP for VSE 1.5 Installation Guide.*

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

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.

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.

Certainly, 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 then 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 can be found 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

R13 Standard save area

R14 Return address

R15 Entry point of preprocessing exit phase

On return:

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

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

On return:

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

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 36.

Overview

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 .`

Notes:

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,PRD1.BASE,PRD2.SCEEBASE)
// 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
/*
/&
* $$ EOJ
```

Figure 14. 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 will be 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` will be 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 14 on page 34, 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 is also transferring 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 -oforms=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 Express2 or PCI Cryptographic Accelerator (PCICA) card 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 Express2 and OSA Express adapters) 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.

Chapter 7. Performance Considerations

Performance and Tuning 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:

- 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

Note:

X means major impact.

x means smaller or secondary impact.

- means no or negligible impact.

Transfer time includes wait for transfer.

DASD time only applicable if DASD involved (for example FTP).

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		X			
TELNETD POOL				X	
SET ALL_BOUND	X				
DISPATCH_TIME	X				
REDISPATCH	X				
ARP_TIME	X				
REUSE_SIZE	x				
FULL_SCAN	X				
GATEWAY	X				
CHECKSUM	x				
Set MAX_SEGMENT			X1		
WINDOW_DEPTH			X1		
CLOSE_DEPTH			X1		
WINDOW_RESTART			X1		
SET RETRANSMIT		X1			
FIXED_RETRANS		x1			
WINDOW		X1			
ADDITIONAL_WINDOW		x1			
SET SLOW_START		x			
SLOW_RESTART		x			
SLOW_INCREMENT		x			
SET TELNETD_BUFFERS				X2	
TRANSFER_BUFFERS					X
MAX_BUFFERS					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 via the z/VSE Internet home page at ***http://www.ibm.com/systems/z/os/zvse/***.

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. Programming Interfaces

Chapter 8. Introducing Socket Programming

TCP/IP for VSE/ESA provides different access methods connecting to/from a VSE host and interchanging data with the system :

- 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 arbitrary Web Browsers (Mosaic, Netscape, MS Internet Explorer, ...) 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 chapter focuses on discussing 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 inter-process 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.
- 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
  - 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).

Nowadays the JAVA language provides a socket API too. There are already JAVA based Client/Server applications showing up exploiting those socket services.

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

- UNIX considerations

  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 considerations

  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 TCP/IP for VSE/ESA

z/VSE provides a series of different socket application programming interfaces (APIs), either provided by TCP/IP for VSE/ESA directly or by the EZA Interface.

- TCP/IP for VSE/ESA 'native' APIs
  - Assembler SOCKET macro interface

    This interface supports to code socket applications, but also to dynamically connect 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 86 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 Reference* for a detailed description of these interfaces.
- TCP/IP APIs using Language Environment for z/VSE
  - LE/VSE 1.4 C socket interface dynamically determines the runtime environment (CICS or Batch). More info regarding this API can be found within this book (see “PL/I” on page 53 and “COBOL” on page 54).
  - The REXX/VSE Socket API support within REXX/VSE. The description of this REXX/VSE Socket API is in the online manual *REXX/VSE Reference*, SC33-6642.
- EZA Interfaces
  - z/VSE provides the EZASMI macro interface for HLASM programmers and the EZASOKET call interface for COBOL, PL/I and HLASM programmers. See Chapter 10, “Using the CALL Instruction Application Programming

Interface (EZASOKET API),” on page 187 and Chapter 11, “Using the Macro Application Programming Interface (EZASMI API),” on page 299 for a description of these interfaces.

Portability Aspects

Assembler

While the TCP/IP for VSE/ESA Assembler Socket Macro Interface is undoubtedly the most efficient way of socket programming, its usage 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.

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 undoubtedly "C". Interfaces for languages other than C may exist in specific environments or may be provided by product specific programming toolkits, which potentially are available for multiple platforms.

If portability to non z/VSE systems isn't essential, the TCP/IP for VSE/ESA preprocessor API, described in the *TCP/IP for VSE 1.5 Programmer’s Reference* manual may be your primary choice. If portability to z/OS or z/VM is essential you should check section “Language Environment” below for further details.

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 de facto 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 WWW by the following URL: ***http://www.opengroup.org/***.

Language Environment

The Language Environment (LE) on the IBM S/390 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

may be 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 a LE conforming Assembler program. This includes support for mixed-language applications.

I.e. while LE based programs, using socket services and written in a programming language other than C are not portable to a non-S/390 system, LE on S/390 provides cross system compatibility as mentioned above.

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

LE Enabled Applications

This manual mentions "LE enabled" applications in several places. An application is considered to be "LE enabled" (or " LE Conforming" or "LE Compliant") when 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. Also C/VSE subroutines including assembler programs using the C prologue/epilogue assembler macros do fulfill this requirement.

Which API to use?

As discussed in the previous chapters already, the selection of the appropriate language and API to use depends on

- Portability

  Ease of cross-platform development (single source code).
- Compatibility

  The S/390 system 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 are showing up differently, depending on the programming language chosen :

Assembler

As mentioned before, the SOCKET macro provided by TCP/IP not only supports to write 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, a 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 may 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.

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

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 - IPNETXP.OBJ

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

The following figure shows an example of the usage of the preprocessor interface. For the complete example please check the *TCP/IP for VSE 1.5 Programmer's Reference* manual.

```
*
*    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.
```

C Language

When using the native TCP/IP BSD-C interface, a TCP/IP socket function call, - for example socket() - causes the stub routine IPNRSTUP.OBJ to be linked to the application code. The control flow of such an application looks like :

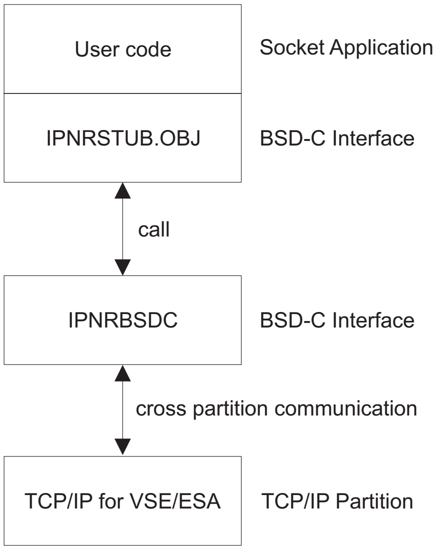


Figure 15. Control Flow when using TCP/IP for VSE/ESA BSD-C Sockets

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.

The figure below shows the logical control flow of a LE/VSE C based socket application. While it is more complex than using TCP/IP's own interfaces directly, it decouples the application from the TCP/IP product.

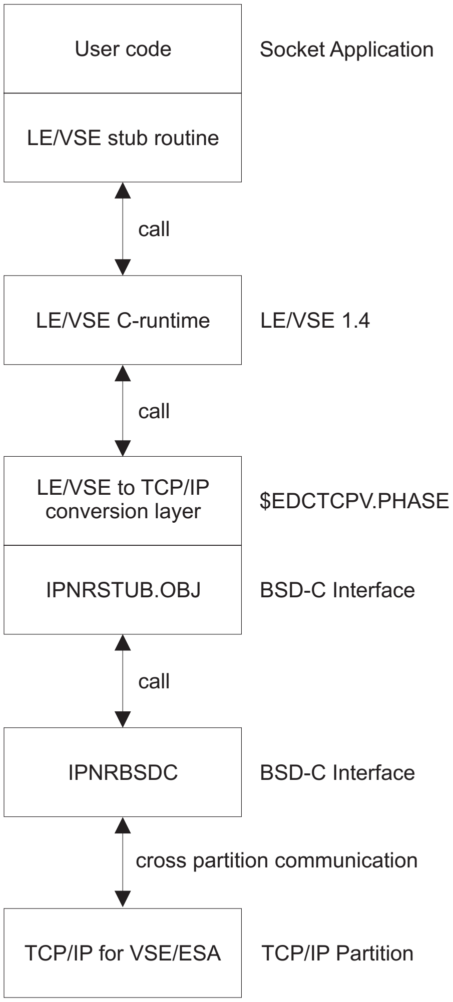


Figure 16. Control Flow when using LE/VSE C Sockets

Notes:

1. When using 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 PRD1.BASE.
2. When using the **C for VSE** compiler we strongly recommend to use the socket API provided by the Language Environment 1.4. The C header files required are provided in VSE library PRD2.SCEEBASE.

Only the IPNRSTUB.OBJ is linked to the user socket application (or the $EDCTCPV phase). Every single BSD/C call - for example socket() - is routed through this IPNRSTUB.OBJ module into the SVA phase IPNRBSDC. This phase performs the main processing for all BSD/C calls, including the cross partition communication with the TCP/IP partition.

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.

C Language

LE/VSE provides socket programming interfaces for C only. While the Language Environment has defined the full range of APIs, as described in Chapter 9, “TCP/IP Support for the LE/VSE C Socket Interface,” on page 89, it bases on TCP/IP's native C interface too. It uses the $EDCTCPV.PHASE as shipped with the TCP/IP for VSE/ESA product to adapt LE calls to the interfaces TCP/IP provides. That is why only the functions available by TCP/IP itself as described in this manual can actually be used. As soon as TCP/IP for VSE/ESA provides additional functionality LE/VSE will automatically pick this up, without requesting to apply LE service.

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 figure shows a simplified sample of the 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 figure shows how the code logic for a server (Daemon) application may look like :

```
  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 may 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 figure uses LE macro CEEENTRY to enable the Language Environment. Then it calls TCP/IP subroutine GETHNAM. Here more complex processing may be coded. At the end of the routine it calls CEETERM to disable the Language Environment as not required any longer.

Note: You should enable LE at the very beginning and terminate it at the end of your application, but not call this sequence more than required or there will be high overhead introduced by starting/terminating LE more than necessary.

```
*
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 figure 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 more 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 the following way:

```
#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 the following:

```
...
DCL GETHNAM EXTERNAL ENTRY
    RETURNS(FIXED BIN(31));
DCL HOSTNAME CHAR(20);
DCL HNSIZE FIXED BIN(31);
DCL CRC FIXED BIN(31);
...
HNSIZE = 20;
CRC = GETHNAM(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 more details on how to use ILC calls.

The following 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 'COBGHNAM' USING BY REFERENCE WORKAREA
                                BY CONTENT BUFSIZE
                                BY REFERENCE RVALUE, RDETAIL.

          DISPLAY WORKAREA UPON CONSOLE.

          STOP RUN.
```

The matching C stub routine for calling gethostname() with COBOL linkage could look the following way:

```
#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

Following you will find an example based on LE's ability to write interlanguage communication applications. The example is split into two figures.

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 SOCKTYPE.
           MOVE 6 TO PROTOCOL.

           DISPLAY 'Calling C socket()'.

           CALL 'TCPSOCKT' USING BY CONTENT DOMAIN, SOCKTYPE, PROTOCOL
                                 BY REFERENCE RVALUE, RDETAIL.

           MOVE RVALUE TO LSOCKET.
```

Figure 17. COBOL Program calling LE C socket routines (Part 1 of 3)

```
*
*  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.

*
*  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.
```

Figure 17. COBOL Program calling LE C socket routines (Part 2 of 3)

```
*
*  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.
```

Figure 17. COBOL Program calling LE C socket routines (Part 3 of 3)

The next figure 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, "TCPSOCKT")
#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;
}
```

Figure 18. LE/VSE C socket interface routines for COBOL (Part 1 of 2)

```
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);
       *perr   = errno;
  }

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

Figure 18. LE/VSE C socket interface routines for COBOL (Part 2 of 2)

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 which 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:

```
*       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                                                       *
*                                                                      *
************************************************************************
```

Figure 19. Sample Program Using EZASMI Macro (Synchronously) (Part 1 of 4)

```
*---------------------------------------------------------------------*
* 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                        *
***********************************************************************
         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=SAMPSERV,          to SAMPSERV 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.'
```

Figure 19. Sample Program Using EZASMI Macro (Synchronously) (Part 2 of 4)

```
***********************************************************************
*        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
*---------------------------------------------------------------------*
* 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 Parms*
*--------------------*
         CNOP  0,4
SAMPSERV DC    0CL16' '                SOCKET NAME structure of SERVER
         DC    AL2(2)                  FAMILY (AF-INET)
         DC    H'4000'                 Port of SAMPSERV
         DC    AL1(9),AL1(164),AL1(155),AL1(122) IP-Addr of SAMPSERV
         DC    XL8'00'                 RESERVED
*
*---------------------------------------------------------------------*
* READ MACRO PARMS   *
*--------------------*
READBL   DC    F'40'                   SIZE OF READ BUFFER
READB    DC    CL40' '                 READ BUFFER
*
```

Figure 19. Sample Program Using EZASMI Macro (Synchronously) (Part 3 of 4)

```
*----  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
```

Figure 19. Sample Program Using EZASMI Macro (Synchronously) (Part 4 of 4)

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

```
*        PRINT NOGEN
************************************************************************
*                                                                      *
*   MODULE NAME:  SAMPSERV                                             *
*                                                                      *
*   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                                             *
*----------------------------------------------------------------------*
SAMPSERV START X'78'                   adjust addr behind part savearea
SAMPSERV AMODE 31
SAMPSERV 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 SAMPSERV,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
```

Figure 20. Sample Program Using EZASMI Macro (Asynchronously) (Part 1 of 5)

```
***********************************************************************
*        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
***********************************************************************
*        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.)
```

Figure 20. Sample Program Using EZASMI Macro (Asynchronously) (Part 2 of 5)

```
***********************************************************************
*       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'SAMPSERV 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                                            *
***********************************************************************
         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
```

Figure 20. Sample Program Using EZASMI Macro (Asynchronously) (Part 3 of 5)

```
*----------------------------------------------------------------------*
* 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
```

Figure 20. Sample Program Using EZASMI Macro (Asynchronously) (Part 4 of 5)

```
*---------------------------------------------------------------------*
* READ MACRO PARMS   *
*--------------------*
READBUFL DC    F'40'                  SIZE OF READ BUFFER
READBUF  DC    CL40'none'             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   SAMPSERV
```

Figure 20. Sample Program Using EZASMI Macro (Asynchronously) (Part 5 of 5)

Of course, there is no real need for this simple program to use the asynchronous interface. Asynchronous processing may be helpful when 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 and using the EZASOKET call interface:

```
CBL LIB APOST RMODE(ANY)                                                SAM00010
      IDENTIFICATION DIVISION.                                          SAM00020
                                                                        SAM00030
      PROGRAM-ID.       SAMPSERV                                        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
```

Figure 21. Sample Program Using EZASOKET Call Using COBOL (Part 1 of 4)

```
 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 '  ... SAMPSERV 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
                                                                  SAM01060
```

Figure 21. Sample Program Using EZASOKET Call Using COBOL (Part 2 of 4)

```
*-------------------------------------------------------*         SAM01070
*     CALL EZASOKET - function = LISTEN                 *         SAM01080
*                     input    = backlog=5              *         SAM01090
*-------------------------------------------------------*         SAM01100
                                                                  SAM01110
     MOVE SOKET-LISTEN TO SOKET-FUNCT.                            SAM01120
     MOVE 0 TO RETCODE.                                           SAM01130
     MOVE 0 TO ERRNO.                                             SAM01140
                                                                  SAM01150
     CALL 'EZASOKET' USING SOKET-FUNCT SOKET-ID BACKLOG           SAM01160
                  ERRNO RETCODE.                                  SAM01170
                                                                  SAM01180
                                                                  SAM01190
*-------------------------------------------*                     SAM01200
*     CALL EZASOKET - function = ACCEPT     *                     SAM01210
*                     input    = SOKET-ID   *                     SAM01220
*-------------------------------------------*                     SAM01230
                                                                  SAM01240
     MOVE SOKET-ACCEPT TO SOKET-FUNCT.                            SAM01250
     MOVE 0 TO RETCODE.                                           SAM01260
     MOVE 0 TO ERRNO.                                             SAM01270
                                                                  SAM01280
     CALL 'EZASOKET' USING SOKET-FUNCT  SOKET-ID SOKET-ADDR       SAM01290
                  ERRNO RETCODE.                                  SAM01300
                                                                  SAM01310
     MOVE RETCODE TO SOKET-ID-NEW.                                SAM01320
                                                                  SAM01330
*-------------------------------------------------------*         SAM01340
*     CALL EZASOKET - function = READ                   *         SAM01350
*-------------------------------------------------------*         SAM01360
                                                                  SAM01370
     MOVE SOKET-READ        TO SOKET-FUNCT.                       SAM01380
     MOVE 0 TO RETCODE.                                           SAM01390
     MOVE 0 TO ERRNO.                                             SAM01400
                                                                  SAM01410
     MOVE LOW-VALUES TO INBUFF.                                   SAM01420
     CALL 'EZASOKET' USING SOKET-FUNCT SOKET-ID-NEW INBUFFL       SAM01430
           INBUFF ERRNO RETCODE.                                  SAM01440
                                                                  SAM01450
*-------------------------------------------------------*         SAM01460
*     CALL EZASOKET - function = WRITE                  *         SAM01470
*-------------------------------------------------------*         SAM01480
                                                                  SAM01490
     MOVE SOKET-WRITE       TO SOKET-FUNCT.                       SAM01500
     MOVE 0 TO RETCODE.                                           SAM01510
     MOVE 0 TO ERRNO.                                             SAM01520
                                                                  SAM01530
     CALL 'EZASOKET' USING SOKET-FUNCT SOKET-ID-NEW MSG001L       SAM01540
           MSG001 ERRNO RETCODE.                                  SAM01550
                                                                  SAM01560
*-------------------------------------------------------*         SAM01570
*     CALL EZASOKET - function = CLOSE                  *         SAM01580
*-------------------------------------------------------*         SAM01590
                                                                  SAM01600
     MOVE SOKET-CLOSE       TO SOKET-FUNCT.                       SAM01610
     MOVE 0 TO RETCODE.                                           SAM01620
     MOVE 0 TO ERRNO.                                             SAM01630
                                                                  SAM01640
     CALL 'EZASOKET' USING SOKET-FUNCT SOKET-ID-NEW               SAM01650
                  ERRNO RETCODE.                                  SAM01660
                                                                  SAM01670
```

Figure 21. Sample Program Using EZASOKET Call Using COBOL (Part 3 of 4)

```
*------------------------------------------------------*          SAM01680
*     CALL EZASOKET - function = CLOSE                 *          SAM01690
*------------------------------------------------------*          SAM01700
                                                                  SAM01710
     MOVE SOKET-CLOSE       TO SOKET-FUNCT.                       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 SAMPSERV.                                            SAM01910
```

Figure 21. Sample Program Using EZASOKET Call Using COBOL (Part 4 of 4)

LE/VSE 1.4 C Socket Programming

General C Programming Considerations

While the Language Environment intends to cover the same functionality as do 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 for VSE/ESA BSD-C implementation. Therefore a programmer porting an application from another S/390 operating system environment may eventually find that the VSE socket interfaces require special attention. This is also true for the TCP/IP for VSE/ESA BSD-C socket interfaces, however, a programmer porting an application for example from z/OS may not expect that source code modifications are eventually required.

The following list is aimed to identify the programming areas that may 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 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.

  getsockopt() The TCP/IP support for the LE C socket API supports option SO_LINGER only.

  recv() The recv() routine by TCP/IP for VSE/ESA doesn't support the MSG_PEEK and MSG_OOB options.

  send() The send() routine by TCP/IP for VSE/ESA doesn't support the MSG_OOB and MSG_DONTROUTE options.

  setsockopt() The TCP/IP support for the LE C socket API 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. As TCP/IP always allows for immediate reuse 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.

LE/VSE Sockets versus TCP/IP for VSE/ESA Sockets - Reference List

As mentioned in previous chapters already, 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 the TCP/IP for VSE/ESA product.

The following list covers the LE C Socket routines documented in the manual *C runtime Library Reference*, SC33-6689, and shows whether they are currently available through the TCP/IP for VSE/ESA product. This list bases on the interfaces available with TCP/IP for VSE/ESA. These interfaces are described in Chapter 9, “TCP/IP Support for the LE/VSE C Socket Interface,” on page 89.

Function	Description	C-LE Interface	TCP/IP for VSE/ESA
accept	accept a new socket connection	yes	yes
aio_cancel	cancel an asynchronous I/O request	yes	yes
aio_error	retrieve error status for an asynchronous I/O operation	yes	yes
aio_read	asynchronous read from a socket	yes	yes
aio_return	retrieve status for an asynchronous I/O operation	yes	yes
aio_suspend	wait for an asynchronous I/O request	yes	yes
aio_write	asynchronous write to a socket	yes	yes
bind	bind a name to a socket	yes	yes
close	close a socket	yes	yes
connect	connect a socket	yes	yes
endhostent	close the host information data set	yes	no
endnetent	close network information data set	yes	no
endprotoent	close protocol information data set	yes	no
endservent	close network services information data sets	yes	no
fcntl	control characteristics of socket	yes	yes (1)
getclientid	get the identifier for the calling application	yes	yes
gethostbyaddr	get a host entry by address	yes	yes
gethostbyname	get a host entry by name	yes	yes
gethostent	get the next host entry	yes	no
gethostid	get unique identifier of current host	yes	yes
gethostname	get the name of the host processor	yes	yes
getnetbyaddr	get a network entry by address	yes	no
getnetbyname	get a network entry by name	yes	no
getnetent	get the next network entry	yes	no
getpeername	get the name of the peer connected to a socket	yes	yes
getprotobyname	get a protocol entry by name	yes	no
getprotobynumber	get a protocol entry by number	yes	no
getprotoent	get the next protocol entry	yes	no
getservbyname	get a service entry by name	yes	no
getservbyport	get a service entry by port	yes	no
getservent	get the next service entry	yes	no
getsockname	get the name of a socket	yes	yes
getsockopt	get the options associated with a socket	yes	yes (1)
givesocket	make the specified socket available	yes	yes
gsk_free_memory	free memory allocated by SSL for VSE	yes	yes
gsk_get_cipher_info	request cipher related information for SSL for VSE	yes	yes
gsk_get_dn _by_label	identify the member name containing the private key and certificates	yes	yes

Function	Description	C-LE Interface	TCP/IP for VSE/ESA
gsk_initialize	set the overall SSL for VSE environment in the current partition	yes	yes
gsk_secure_soc_close	end a secure socket connection	yes	yes
gsk_secure_soc_init	initialize a secure socket connection	yes	yes
gsk_secure_soc_read	receive data on a secure socket connection	yes	yes
gsk_secure_soc_reset	refresh the security parameters for a session	yes	yes
gsk_secure_soc_write	send data on a secure socket connection	yes	yes
gsk_uninitialize	remove the overall SSL for VSE environment in the current partition	yes	yes
htonl	translate address host to network long	yes	yes
htons	translate address host to network short	yes	yes
inet_addr	translate an internet address into network byte order	yes	yes
inet_lnaof	translate a local network address into host byte order	yes	yes
inet_makeaddr	create an internet host address	yes	yes
inet_netof	get the network number from the internet host address	yes	yes
inet_network	get the network number from the decimal host address	yes	yes
inet_ntoa	get the decimal internet host address	yes	yes
initapi	connect an application to the TCP/IP interface	yes	no
ioctl	specify the socket operating characteristics	yes	yes (1)
listen	prepare the server for incoming client requests	yes	yes
ntohl	translate a long integer into host byte order	yes	yes
ntohs	translate a short integer into host byte order	yes	yes
poll	monitor activity on socket descriptors	yes	no
read	read data from a socket into a buffers	yes	yes
readv	read data on a socket and store in a set of buffers	yes	no
recv	receive data on a socket	yes	yes (1)
recvfrom	receive messages on a socket	yes	yes
recvmsg	receive messages on a socket and store in an array of messages	yes	no
select	monitor activity on sockets	yes	yes
selectex	monitor activity on sockets	yes	yes
send	send data on a socket	yes	yes (1)
sendmsg	send messages on a socket	yes	no
sendto	send data on a socket	yes	yes
sethostent	open the host information data set	yes	no
setnetent	open the network information data set	yes	no
setprotoent	open the protocol information data set	yes	no

Function	Description	C-LE Interface	TCP/IP for VSE/ESA
setservent	open the services information data set	yes	no
setsockopt	set options associated with a socket	yes	yes (1)
shutdown	shut down all or part of a duplex connection	yes	yes (1)
socket	create a socket	yes	yes (1)
socketpair	create a pair of sockets	yes	no
takesocket	acquire a socket from another program	yes	yes
tcp_cleanup	cleanup LE environment used for TCP/IP	yes	no
termapi	terminate a session created by initapi	yes	no
write	write data from buffer to connected socket	yes	yes
writev	write data to a socket from an array	yes	no

(1) Limited functionality, for example certain parameters are not supported. See Chapter 9, “TCP/IP Support for the LE/VSE C Socket Interface,” on page 89 for details.

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 (PRD1.BASE) 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 (PRD1.BASE) 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.

z/OS EZASMI and EZASOKET Calls Supported by z/VSE

The following table shows, which z/OS EZASMI and EZASOKET Socket Calls are supported , and what the differences are compared to the z/OS interfaces. These Socket Calls apply both to the EZASMI macro interface and to the EZASOKET call interface, except for the GLOBAL and TASK calls, which apply to the EZASMI interface only.

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).

Table 3. Supported z/OS Socket Calls

z/OS Socket Call	z/VSE Support/Difference
ACCEPT	Supported with difference: • Parameter NS ignored
BIND	Supported.
CANCEL	Supported for EZASMI only.
CLOSE	Supported.
CONNECT	Supported.
FCNTL	Supported.
FREEADDRINFO	Supported (not supported by TCP/IP for VSE/ESA program).
GETADDRINFO	Supported (not supported by TCP/IP for VSE/ESA program).
GETCLIENTID	Supported with different CLIENTID structure (total length remains the same): • 4 bytes: Domain-ID • 8 bytes: Address Space Name (partition syslog-id, left adjusted and padded with blanks) • 8 bytes: Subtask Name (as specified or defaulted with the INITAPI function request) • 20 bytes: reserved (binary zeroes)
GETHOSTBYADDR	Supported with different HOSTENT structure: • No ALIASes • Max HostName Length = 64
GETHOSTBYNAME	Supported with different HOSTENT structure: • No ALIASes • Max HostName Length = 64
GETHOSTID	Supported.
GETHOSTNAME	Supported with difference: • Max HostName Length = 64

Table 3. Supported z/OS Socket Calls (continued)

z/OS Socket Call	z/VSE Support/Difference
GETIBMOPT	Not supported.
GETNAMEINFO	Supported (not supported by TCP/IP for VSE/ESA program).
GETPEERNAME	Supported.
GETSOCKNAME	Supported.
GETSOCKOPT	Supported with difference: • Supported option is SO_LINGER
GIVESOCKET	Supported with difference: • see CLIENTID structure with GETCLIENTID function
GLOBAL	Not supported.
INITAPI	Supported with difference: • EZASMI only: Parameter APITYPE=3 is ignored. • EZASMI only: Parameter ASYNC='EXIT' is rejected with ERRNO=EINVAL • EZASMI only: Parameter UUEXIT is ignored • Parameter MAXSOC: maximum number = default number = 8192 sockets • Parameter SUBTASK: Any 8-char name may be specified. If this parameter is not specified (EZASMI macro) or is specified as 8 blanks, a default subtask name is used: **byte 0-3** first 3 chars from JOBNAME (batch environment) or from EIBTRNID (CICS transaction environment) **byte 4** x'F0' (batch environment) or x'F1' (CICS transaction environment) **byte 4-7** VSE TaskId (batch environment) or CICS EIBTASKN (CICS transaction environment) • Parameter IDENT: The ADSNAME parameter can be used to specify the name of the TCP/IP Interface Routine (the default is EZASOH99, the IBM-provided TCP/IP Interface Routine). The specified name of the TCP/IP Interface Routine (or the default) can be overwritten with JCL statement "//SETPARM EZA$PHA=name". Starting with z/VSE 4.2, the TCPNAME parameter can be used to select to local TCP/IP Stack. • Output parameter MAXSNO: The socket descriptor assigned to the application will not be in consecutive order.
IOCTL	Supported with difference: • Only COMMAND=FIONBIO is supported.

Table 3. Supported z/OS Socket Calls (continued)

z/OS Socket Call	**z/VSE Support/Difference**
LISTEN	Supported with difference: • Requires BIND before.
NTOP	Supported (not supported by TCP/IP for VSE/ESA program).
PTON	Supported (not supported by TCP/IP for VSE/ESA program).
READ	Supported with difference: • Parameter ALET ignored.
READV	Supported.
RECV	Supported with difference: • Flags must be ZERO. • Parameter ALET ignored.
RECVFROM	Supported with difference: • Flags must be ZERO. • Parameter ALET ignored. • EZASMI only: no asynchronous support
RECVMSG	Not supported.
SELECT	Supported.
SELECTEX	Supported with difference: • ECB List may hold 254 ECBs.
SEND	Supported with difference: • Flags must be ZERO. • Parameter ALET ignored.
SENDMSG	Not supported.
SENDTO	Supported with difference: • Flags must be ZERO. • Parameter ALET ignored. • EZASMI only: no asynchronous support
SETSOCKOPT	Supported with difference: • Supported options are SO_LINGER, SO_KEEPALIVE, and SO_REUSEADDR
SHUTDOWN	Supported with difference: • Only HOW=SHUT_RDWR is supported. • SHUTDOWN works like CLOSE.
SOCKET	Supported with difference: • RAW sockets are not supported. • Parameter NS is ignored.
TAKESOCKET	Supported with difference: • Parameter NS is ignored • For the different CLIENTID refer to GETCLIENTID.
TASK	Supported for EZASMI only.
TERMAPI	Supported.

Table 3. Supported z/OS Socket Calls (continued)

z/OS Socket Call	z/VSE Support/Difference
WRITE	Supported with difference: • Parameter ALET ignored.
WRITEV	Supported.

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 (when used with TCP/IP for VSE/ESA.

Table 4 shows the ERRNO values sorted by their decimal value. Table 5 on page 85 shows the same table sorted by ERRNO names.

Table 4. ERRNO Values Sorted by Value

ERRNO	ERRNO Value from LE/C or EZASMI/ EZASOKET (Note 1)	ERRNO Value from BSD-C (Notes 1/2)	Description	See Note
EBADF	113	2	Bad socket descriptor.	
EFAULT	118	17	Bad address or buffer address not accessible.	
EINVAL	121	3	Invalid parameter.	
EIO	122	9	Socket closed.	
ENFILE	127	6	Too many open sockets.	
ENOENT	129	10	No such socket.	
ENOMEM	132	15	Not enough memory to fulfill the request.	
ENOSYS	134	11	Function not implemented.	
EMVSERR	157	100	An internal error has occurred.	6
EMVSPARM	158	14	Bad parameters.	
EVSE	183	12	Not supported under VSE.	
EWOULDBLOCK	1102	19	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.	
EINPROGRESS	1103	24	Socket connection in progress. O_NONBLOCK is set for the socket descriptor and the connection cannot be immediately established.	

Table 4. ERRNO Values Sorted by Value (continued)

ERRNO	ERRNO Value from LE/C or EZASMI/ EZASOKET (Note 1)	ERRNO Value from BSD-C (Notes 1/2)	Description	See Note
EALREADY	1104	22	Connection request already in progress. A connection request is already in progress for the specified socket.	
EDESTADDRREQ	1106	23	Destination address required. No bind address was specified.	
ENOPROTOPT	1109	7	No Option recognized. The option specified to setsockopt() is not supported.	
EPROTONOSUPPORT	1110	n/a	The protocol is not supported by the address family, or the protocol is not supported by the implementation.	
EOPNOTSUPP	1112	4	Socket call not supported.	
EAFNOSUPPORT	1114	13	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	18	Specified address or port is already in use.	
ENETDOWN	1117	1117	The local interface to use or reach the destination is down.	
ECONNRESET	1121	20	Connection was forcibly closed/reset by the peer.	
ENOBUFS	1122	5	No buffers available. Insufficient buffer resources were available in the system to perform the socket operation.	
EISCONN	1123	21	Specified socket is already connected.	
ENOTCONN	1124	8	Socket is not connected.	

Table 4. ERRNO Values Sorted by Value (continued)

ERRNO	ERRNO Value from LE/C or EZASMI/ EZASOKET (Note 1)	ERRNO Value from BSD-C (Notes 1/2)	Description	See Note
ETIMEDOUT	1127	16	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.	
ECANCELED	1152	1152	The asynchronous I/O request has been canceled.	
EZAINVFU	20000	n/a	Invalid Function used with EZASOKET call	4
EZAINVPA	20001	n/a	Incorrect Parameter with EZASOKET call	4
EZAERL00	20100	n/a	Error loading phase EZASOH00	3
EZAERGTV	20107	n/a	Not enough partition GETVIS	3
EZAERNIN	20108	n/a	First call not INITAPI	3
EZAERREC	20111	n/a	Recursive entry of EZA interface	3
EZAETRNA	20112	n/a	EZATRUE not active (CICS only)	3
EZAERLIF	20113	n/a	LOAD of TCP/IP interface routine failed	3
EZAERBRI	20114	n/a	Bad return code from TCP/IP I/F routine	3

Notes:

1. ERRNO values are shown in decimal.
2. Prior to TCP/IP 1.5 Service Pack E (or if NEW_INTERFACE is not used), these ERRNO values are shown in the BSD-C Trace ("$SOCKDBG trace") provided by TCP/IP for VSE/ESA. See Information APAR II11836 for further details on this trace.
3. Used by EZASMI and EZASOKET interfaces only.
4. Used by EZASOKET interface only.
5. The BSD-C Trace will show the CSI-specific internal error value 100 instead of 157.

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 5. ERRNO Values sorted by Name

ERRNO	Value
EADDRINUSE	1115
EAFNOSUPPORT	1114
EALREADY	1104
EBADF	113
ECANCELED	1152
ECONNRESET	1121
EDESTADDRREQ	1106
EFAULT	118
EINPROGRESSS	1103
EINVAL	121
EIO	122
EISCONN	1123
EMVSERR	157
EMVSPARM	158
ENETDOWN	1117
ENFILE	127
ENOBUFS	1122
ENOENT	129
ENOMEM	132
ENOPROTOOPT	1109
ENOSYS	134
ENOTCONN	1124
EOPNOTSUPP	1112
EPROTONOSUPPORT	1110
ETIMEDOUT	1127
EVSE	183
EWOULDBLOCK	1102
EZAERBRI	20114
EZAERGTV	20107
EZAERL00	20100
EZAERLIF	20113
EZAERNIN	20108
EZAERREC	20111
EZAETRNA	20112
EZAINVFU	20000

Table 5. ERRNO Values sorted by Name (continued)

ERRNO	Value
EZAINVPA	20001

CICS Considerations

The C Socket programming interface supports writing applications for either a CICS or batch execution environment.

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
- 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 9, “TCP/IP Support for the LE/VSE C Socket Interface,” on page 89.

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 6. COMMAREA parameter list

Offset	Length	Description	
0	8	Eyecatcher "EZATRUE"	
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 for VSE/ESA partition that has been assigned ID=00. This assignment is made in the PARM field of the TCP/IP for VSE/ESA EXEC statement. Its default value is also "00". If you wish to connect with a different TCP/IP for VSE/ESA partition for testing or other purposes, you can do so by including an appropriate OPTION statement in your C program's JCL:

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

In the above, xx is the two-digit ID number, coded exactly as in the TCP/IP for z/VSE start-up JCL.

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 the TCP/IP for VSE Application Pak.

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

Overview

This chapter describes the LE/VSE C socket interface as provided by TCP/IP for VSE/ESA.

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 may 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* manual.

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.

Notes:

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 PRD1.BASE. Therefore the socket application is decoupled from the TCP/IP product (see Figure 16 on page 51 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.

   When you receive this message check your application's LIBDEF for correctness and check this chapter 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 chapter for reference and implementation details.

TCP/IP Callable Functions — Function Descriptions

accept() — Accept a New Connection on a Socket

```
#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 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 as defined in the include file **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.

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);
```

Related Information

- “bind() — Bind a Name to a Socket” on page 104
- “connect() — Connect a Socket” on page 108
- “getpeername() — Get the Name of the Peer Connected to a Socket” on page 120
- “listen() — Prepare the Server for Incoming Client Requests” on page 157
- “socket() — Create a Socket” on page 181

aio_cancel() — Cancel an Asynchronous I/O Request

```
#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() will be cancelable if there is no data available at the time that 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.

Related Information

- "aio_read() — Asynchronous Read from a Socket" on page 95
- "aio_write() — Asynchronous Write to a Socket" on page 101
- "aio_return() — Retrieve Status for an Asynchronous I/O Operation" on page 98
- "aio_error() — Retrieve Error Status for an Asynchronous I/O Operation" on page 94

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

```
#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, then the error status will be 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`.

Related Information

- “aio_return() — Retrieve Status for an Asynchronous I/O Operation” on page 98
- “aio_suspend() — Wait for an Asynchronous I/O Request” on page 99
- “aio_read() — Asynchronous Read from a Socket” on page 95

aio_read() — Asynchronous Read from a Socket

```
#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 when the data cannot be delivered immediately).

Asynchronous I/O is currently only supported for sockets. The *aio_offset* field may be set but it will be 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 *aiocb* with `aio_error()` until the result is no longer `EINPROGRESS`.

Be aware that the operation may complete, before control returns from the call to `aio_read()`. Even when the operation does complete this quickly the return value from the call to `aio_read()` will be 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 may 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 aiocbp 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 aiocbp, asynchronous operations using an invalid *aiocbp*, 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 `LIO_READ` by the function `aio_read()`.

`_POSIX-PRIORITIZED_IO` is not supported. The *aio_reqprio* field may be set but it will be 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 may 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 will be 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.

In the case where 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 will be 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()`.

Related Information

- "aio_write() — Asynchronous Write to a Socket" on page 101
- "aio_cancel() — Cancel an Asynchronous I/O Request" on page 92
- "aio_error() — Retrieve Error Status for an Asynchronous I/O Operation" on page 94
- "aio_return() — Retrieve Status for an Asynchronous I/O Operation" on page 98
- "aio_suspend() — Wait for an Asynchronous I/O Request" on page 99

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

```
#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`.

Related Information

- "aio_read() — Asynchronous Read from a Socket" on page 95
- "aio_suspend() — Wait for an Asynchronous I/O Request" on page 99
- "aio_error() — Retrieve Error Status for an Asynchronous I/O Operation" on page 94

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

```
#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 *aoicb* 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.

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 will be 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.

Related Information

- “aio_return() — Retrieve Status for an Asynchronous I/O Operation” on page 98
- “aio_read() — Asynchronous Read from a Socket” on page 95
- “aio_error() — Retrieve Error Status for an Asynchronous I/O Operation” on page 94

aio_write() — Asynchronous Write to a Socket

```
#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_reqprio	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 when the data cannot be delivered immediately).

Asynchronous I/O is currently only supported for sockets. The *aio_offset* field may be set but it will be 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 *aiocb* with `aio_error()` until the result is no longer `EINPROGRESS`.

Be aware that the operation may complete before control returns from the call to `aio_read()`. Even when the operation does complete this quickly, the return value from the call to `aio_read()` will be 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 may 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 *aiocbp* 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 *aiocbp*, attempting asynchronous operations using an invalid *aiocbp*, 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 it will be 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_ filedes* 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 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 will be set to the corresponding value.

Error Status / errno
Description

EBADF
The *aio_ filedes* argument is not a valid socket descriptor open for writing.

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

EINVAL
The *aio_nbytes* is not a valid value or *aio_sigevent* contains an invalid value.

In the case where 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 `aio_cancel()`.

Related Information

- “aio_read() — Asynchronous Read from a Socket” on page 95
- “aio_cancel() — Cancel an Asynchronous I/O Request” on page 92

- “aio_error() — Retrieve Error Status for an Asynchronous I/O Operation” on page 94
- “aio_return() — Retrieve Status for an Asynchronous I/O Operation” on page 98
- “aio_suspend() — Wait for an Asynchronous I/O Request” on page 99

bind() — Bind a Name to a Socket

```
#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
socket	The socket descriptor returned by a previous `socket()` call.
name	The pointer to a **sockaddr** structure containing the name that is to be bound to *socket*.
namelen	The size of *name* 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*.

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 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.

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 AF_INET.
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.

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));
```

Related Information

- “connect() — Connect a Socket” on page 108
- “getsockname() — Get the Name of a Socket” on page 121
- “inet_addr() — Translate an Internet Address into Network Byte Order” on page 149
- “listen() — Prepare the Server for Incoming Client Requests” on page 157
- “socket() — Create a Socket” on page 181

close() — Close a Socket

```
#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.

For AF_INET 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, and it can be found 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.

Related Information

- "accept() — Accept a New Connection on a Socket" on page 90

connect() — Connect a Socket

```
#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 will be 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()`, `writev()`, `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*.

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.

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 immediately established. The connection will be 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.

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 104.

```
#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));
```

Related Information

- "accept() — Accept a New Connection on a Socket" on page 90
- "bind() — Bind a Name to a Socket" on page 104
- "inet_addr() — Translate an Internet Address into Network Byte Order" on page 149
- "listen() — Prepare the Server for Incoming Client Requests" on page 157
- "select() — Monitor Activity on Sockets" on page 167
- "selectex() — Monitor Activity on Sockets" on page 171
- "socket() — Create a Socket" on page 181

fcntl() — Control Open Socket Descriptors

```
#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 flagsmarks *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:

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.

Example

Sockets 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 */
```

Related Information

- “close() — Close a Socket” on page 107
- “getsockopt() — Get the Options Associated with a Socket” on page 123
- “ioctl() — Control Socket” on page 156
- “setsockopt() — Set Options Associated with a Socket” on page 177
- “socket() — Create a Socket” on page 181

getclientid() — Get the Identifier for the Calling Application

```
#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.

gethostbyaddr() — Get a Host Entry by Address

```
#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.)
address_len	The size of *address* in bytes.
domain	The address domain supported (AF_INET).

The `gethostbyaddr()` call returns a pointer to a **hostent** structure for the host address specified on the call.

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);
```

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.

Related Information

- "gethostbyname() — Get a Host Entry by Name" on page 116

gethostbyname() — Get a Host Entry by Name

```
#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 an 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.

Related Information

- "gethostbyaddr() — Get a Host Entry by Address" on page 114
- "gethostname() — Get the Name of the Host Processor" on page 119

gethostid() — Get the Unique Identifier of the Current Host

```
#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.

Related Information

- “gethostname() — Get the Name of the Host Processor” on page 119

gethostname() — Get the Name of the Host Processor

```
#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.

Related Information

- “gethostbyname() — Get a Host Entry by Name” on page 116
- “gethostid() — Get the Unique Identifier of the Current Host” on page 118

getpeername() — Get the Name of the Peer Connected to a Socket

```
#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.

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.

Related Information

- “accept() — Accept a New Connection on a Socket” on page 90
- “connect() — Connect a Socket” on page 108
- “getsockname() — Get the Name of a Socket” on page 121
- “socket() — Create a Socket” on page 181

getsockname() — Get the Name of a Socket

```
#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*.

The `getsockname()` call is often used to discover the port assigned to a socket after the socket has been implicitly bound to a port. For example, an application can call `connect()` without previously calling `bind()`. In this case, the `connect()` call completes the binding necessary by assigning a port to the socket. This assignment can be 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.

Related Information

- “accept() — Accept a New Connection on a Socket” on page 90
- “bind() — Bind a Name to a Socket” on page 104
- “connect() — Connect a Socket” on page 108

- “getpeername() — Get the Name of the Peer Connected to a Socket” on page 120
- “socket() — Create a Socket” on page 181

getsockopt() — Get the Options Associated with a Socket

```
#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. Only SOL_SOCKET is supported.
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.

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**

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 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_LINGER Lingers on close if data is present. When this option is enabled and there is unsent data present when close() is called, the calling application is blocked during the close() call until the data is transmitted or the connection has timed out. If this option is disabled, the 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.

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_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.

Example

The following are examples of the getsockopt() call. See “setsockopt() — Set Options Associated with a Socket” on page 177 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  */
     }
}
```

Related Information

- “bind() — Bind a Name to a Socket” on page 104
- “close() — Close a Socket” on page 107
- “setsockopt() — Set Options Associated with a Socket” on page 177
- “socket() — Create a Socket” on page 181

givesocket() — Make the Specified Socket Available

Format

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

int givesocket(int socket,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.
- 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, *subtaskname* parameter will be set to blanks.

c_reserved.type When set to `SO_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 `SO_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.

Related Information

- "accept() — Accept a New Connection on a Socket" on page 90
- "close() — Close a Socket" on page 107
- "getclientid() — Get the Identifier for the Calling Application" on page 113
- "listen() — Prepare the Server for Incoming Client Requests" on page 157
- "select() — Monitor Activity on Sockets" on page 167
- "takesocket() — Acquire a Socket from Another Program" on page 184

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: This function is currently not used under VSE.

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.

Related Information

- "gsk_get_dn_by_label() — Get Distinguished Name Based on the Label" on page 132
- For more details refer to *TCP/IP for VSE 1.5 Optional Features*.

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.

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 sucessful 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` 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.

Related Information

- "gsk_secure_soc_init() — Initialize Data Areas for a Secure Socket Connection" on page 136
- For more details refer to *TCP/IP for VSE 1.5 Optional Features*.

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. 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: `gsk_get_dn_by_label()` cannot be called prior to calling `gsk_initialize()`.

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.

Related Information

- "gsk_free_memory() — Free memory allocated for SSL" on page 129
- "gsk_initialize() — Initialize the SSL Environment" on page 133
- "gsk_secure_soc_init() — Initialize Data Areas for a Secure Socket Connection" on page 136
- For more details refer to *TCP/IP for VSE 1.5 Optional Features*.

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:

```
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.

Related Information

- “gsk_secure_soc_close() — Close a Secure Socket Connection” on page 135
- “gsk_secure_soc_init() — Initialize Data Areas for a Secure Socket Connection” on page 136
- “gsk_secure_soc_read() — Receive Data on a Secure Socket Connection” on page 140
- “gsk_secure_soc_write() — Send Data on a Secure Socket Connection” on page 143
- “gsk_uninitialize() — Remove Current Settings for the SSL Environment” on page 145
- For more details refer to *TCP/IP for VSE 1.5 Optional Features*.

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: If you do not call gsk_secure_soc_close(), the storage referenced by the *user_socket* parameter is not be freed.

Note: 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.

Related Information

- "gsk_initialize() — Initialize the SSL Environment" on page 133
- "gsk_secure_soc_init() — Initialize Data Areas for a Secure Socket Connection" on page 136
- "gsk_secure_soc_read() — Receive Data on a Secure Socket Connection" on page 140
- "gsk_secure_soc_write() — Send Data on a Secure Socket Connection" on page 143
- For more details refer to *TCP/IP for VSE 1.5 Optional Features*.

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 imples 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 will be 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 130 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 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 a failor occurres 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.

Related Information

- "gsk_get_cipher_info() — Query Cipher Related Information" on page 130
- "gsk_get_dn_by_label() — Get Distinguished Name Based on the Label" on page 132
- "gsk_initialize() — Initialize the SSL Environment" on page 133
- "gsk_secure_soc_close() — Close a Secure Socket Connection" on page 135
- "gsk_secure_soc_read() — Receive Data on a Secure Socket Connection" on page 140
- "gsk_secure_soc_reset() — Refresh the Security Parameters" on page 142
- "gsk_secure_soc_write() — Send Data on a Secure Socket Connection" on page 143
- For more details refer to *TCP/IP for VSE 1.5 Optional Features*.

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.

Related Information

- "gsk_initialize() — Initialize the SSL Environment" on page 133
- "gsk_secure_soc_close() — Close a Secure Socket Connection" on page 135
- "gsk_secure_soc_init() — Initialize Data Areas for a Secure Socket Connection" on page 136
- "gsk_secure_soc_write() — Send Data on a Secure Socket Connection" on page 143
- For more details refer to *TCP/IP for VSE 1.5 Optional Features*.

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.

Related Information

- "gsk_secure_soc_init() — Initialize Data Areas for a Secure Socket Connection" on page 136
- 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.

Related Information

- "gsk_initialize() — Initialize the SSL Environment" on page 133
- "gsk_secure_soc_close() — Close a Secure Socket Connection" on page 135

- "gsk_secure_soc_init() — Initialize Data Areas for a Secure Socket Connection" on page 136
- "gsk_secure_soc_read() — Receive Data on a Secure Socket Connection" on page 140
- For more details refer to *TCP/IP for VSE 1.5 Optional Features*.

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.

Related Information

- “gsk_initialize() — Initialize the SSL Environment” on page 133
- “gsk_secure_soc_init() — Initialize Data Areas for a Secure Socket Connection” on page 136
- For more details refer to *TCP/IP for VSE 1.5 Optional Features*.

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.

Related Information

- "gsk_initialize() — Initialize the SSL Environment" on page 133
- "gsk_secure_soc_init() — Initialize Data Areas for a Secure Socket Connection" on page 136
- For more details refer to *TCP/IP for VSE 1.5 Optional Features*.

htonl() — Translate Address Host to Network Long

```
#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 S/390, 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.

Related Information

- “htons() — Translate an Unsigned Short Integer into Network Byte Order” on page 148
- “ntohl() — Translate a Long Integer into Host Byte Order” on page 159
- “ntohs() — Translate an Unsigned Short Integer into Host Byte Order” on page 160

htons() — Translate an Unsigned Short Integer into Network Byte Order

```
#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 S/390, 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.

Related Information

- "htonl() — Translate Address Host to Network Long" on page 147
- "ntohl() — Translate a Long Integer into Host Byte Order" on page 159
- "ntohs() — Translate an Unsigned Short Integer into Host Byte Order" on page 160

inet_addr() — Translate an Internet Address into Network Byte Order

```
#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).

Related Information

- “inet_makeaddr() — Create an Internet Host Address” on page 152
- “inet_netof() — Get the Network Number from the Internet Host Address” on page 153

- "inet_network() — Get the Network Number from the Decimal Host Address" on page 154
- "inet_ntoa() — Get the Decimal Internet Host Address" on page 155

inet_lnaof() — Translate a Local Network Address into Host Byte Order

```
#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
in	The host Internet address.

Returned Value

The local network address is returned in host byte order.

Related Information

- "inet_makeaddr() — Create an Internet Host Address" on page 152
- "inet_netof() — Get the Network Number from the Internet Host Address" on page 153
- "inet_network() — Get the Network Number from the Decimal Host Address" on page 154
- "inet_ntoa() — Get the Decimal Internet Host Address" on page 155

inet_makeaddr() — Create an Internet Host Address

```
#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
net	The network number.
lna	The local network address.

Returned Value

The Internet address is returned in network byte order.

Related Information

- "inet_lnaof() — Translate a Local Network Address into Host Byte Order" on page 151
- "inet_netof() — Get the Network Number from the Internet Host Address" on page 153
- "inet_network() — Get the Network Number from the Decimal Host Address" on page 154
- "inet_ntoa() — Get the Decimal Internet Host Address" on page 155

inet_netof() — Get the Network Number from the Internet Host Address

```
#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.

Related Information

- “inet_lnaof() — Translate a Local Network Address into Host Byte Order” on page 151
- “inet_makeaddr() — Create an Internet Host Address” on page 152
- “inet_ntoa() — Get the Decimal Internet Host Address” on page 155

inet_network() — Get the Network Number from the Decimal Host Address

```
#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.

Related Information

- "inet_lnaof() — Translate a Local Network Address into Host Byte Order" on page 151
- "inet_makeaddr() — Create an Internet Host Address" on page 152
- "inet_ntoa() — Get the Decimal Internet Host Address" on page 155

inet_ntoa() — Get the Decimal Internet Host Address

```
#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.

Related Information

- “inet_addr() — Translate an Internet Address into Network Byte Order” on page 149
- “inet_lnaof() — Translate a Local Network Address into Host Byte Order” on page 151
- “inet_makeaddr() — Create an Internet Host Address” on page 152
- “inet_netof() — Get the Network Number from the Internet Host Address” on page 153
- “inet_network() — Get the Network Number from the Decimal Host Address” on page 154

ioctl() — Control Socket

```
#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.

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.
EINVAL	The request is invalid or not supported.
EMVSPARM	Incorrect parameters were passed to the service.

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);
⋮
```

Related Information

- “close() — Close a Socket” on page 107
- “fcntl() — Control Open Socket Descriptors” on page 111
- “read() — Read From a Socket” on page 161
- “write() — Write Data on a Socket” on page 185

listen() — Prepare the Server for Incoming Client Requests

```
#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.

Related Information

- "accept() — Accept a New Connection on a Socket" on page 90
- "bind() — Bind a Name to a Socket" on page 104

- “connect() — Connect a Socket” on page 108
- “socket() — Create a Socket” on page 181

ntohl() — Translate a Long Integer into Host Byte Order

```
#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 S/390, 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.

Related Information

- “htonl() — Translate Address Host to Network Long” on page 147
- “htons() — Translate an Unsigned Short Integer into Network Byte Order” on page 148
- “ntohs() — Translate an Unsigned Short Integer into Host Byte Order” on page 160

ntohs() — Translate an Unsigned Short Integer into Host Byte Order

```
#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 S/390, 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.

Related Information

- "htonl() — Translate Address Host to Network Long" on page 147
- "htons() — Translate an Unsigned Short Integer into Network Byte Order" on page 148
- "ntohl() — Translate a Long Integer into Host Byte Order" on page 159

read() — Read From a Socket

```
#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**
fs	The socket descriptor.
buf	The pointer to the buffer that receives the data.
N	The length in bytes of the buffer pointed to by the *buf* parameter.

Behavior for Sockets

The read() call reads data on a socket with descriptor *fs* and stores it in a buffer. The read() all 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 111 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.

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 runtime always returns the value -1 and errno is set to ENOTCONN.

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;
  }
}
```

Related Information

- “close() — Close a Socket” on page 107
- “connect() — Connect a Socket” on page 108
- “fcntl() — Control Open Socket Descriptors” on page 111
- “getsockopt() — Get the Options Associated with a Socket” on page 123
- “ioctl() — Control Socket” on page 156
- “recv() — Receive Data on a Socket” on page 163
- “recvfrom() — Receive Messages on a Socket” on page 165
- “select() — Monitor Activity on Sockets” on page 167
- “selectex() — Monitor Activity on Sockets” on page 171
- “send() — Send Data on a Socket” on page 173
- “sendto() — Send Data on a Socket” on page 175
- “setsockopt() — Set Options Associated with a Socket” on page 177
- “socket() — Create a Socket” on page 181
- “write() — Write Data on a Socket” on page 185

recv() — Receive Data on a Socket

```
#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.

Parameter	Description
socket	The socket descriptor.
buf	The pointer to the buffer that receives the data.
len	The length in bytes of the buffer pointed to by the *buf* parameter.
flags	reserved zero

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 111 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.

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.

Related Information

- “connect() — Connect a Socket” on page 108
- “fcntl() — Control Open Socket Descriptors” on page 111
- “getsockopt() — Get the Options Associated with a Socket” on page 123
- “ioctl() — Control Socket” on page 156
- “read() — Read From a Socket” on page 161
- “recvfrom() — Receive Messages on a Socket” on page 165
- “select() — Monitor Activity on Sockets” on page 167
- “selectex() — Monitor Activity on Sockets” on page 171
- “send() — Send Data on a Socket” on page 173
- “sendto() — Send Data on a Socket” on page 175
- “setsockopt() — Set Options Associated with a Socket” on page 177
- “socket() — Create a Socket” on page 181
- “write() — Write Data on a Socket” on page 185

recvfrom() — Receive Messages on a Socket

```
#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
socket	The socket descriptor.
buffer	The pointer to the buffer that receives the data.
length	The length in bytes of the buffer pointed to by the *buffer* parameter.
flags	reserved zero
name	A pointer to a socket address structure from which data is received. If *name* is a nonzero value, the source address is returned.
namelen	The size of *name* 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 111 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.

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.

Related Information

- “fcntl() — Control Open Socket Descriptors” on page 111
- “getsockopt() — Get the Options Associated with a Socket” on page 123
- “ioctl() — Control Socket” on page 156
- “read() — Read From a Socket” on page 161
- “recv() — Receive Data on a Socket” on page 163
- “select() — Monitor Activity on Sockets” on page 167
- “selectex() — Monitor Activity on Sockets” on page 171
- “send() — Send Data on a Socket” on page 173
- “sendto() — Send Data on a Socket” on page 175
- “setsockopt() — Set Options Associated with a Socket” on page 177
- “socket() — Create a Socket” on page 181
- “write() — Write Data on a Socket” on page 185

select() — Monitor Activity on Sockets

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <types.h>
#include <time.h>

int select(int            num,
          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: 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 will be used.

Note: 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 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 *writelist* 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*, *writelist*, 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. Should the return value for socket descriptors be greater than 65,535, only 65,535 will be 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.

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);
}
```

Related Information

- “selectex() — Monitor Activity on Sockets” on page 171

selectex() — Monitor Activity on Sockets

```
#define _XOPEN_SOURCE_EXTENDED 1
#define _ALL_SOURCE
#include <types.h>
#include <time.h>

int selectex( int            num,
              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. Should the return value for socket descriptors be greater than 65,535, only 65,535 will be 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.

Related Information

- "accept() — Accept a New Connection on a Socket" on page 90
- "connect() — Connect a Socket" on page 108
- "recv() — Receive Data on a Socket" on page 163
- "selectex() — Monitor Activity on Sockets" on page 171
- "send() — Send Data on a Socket" on page 173

send() — Send Data on a Socket

```
#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
socket	The socket descriptor.
msg	The pointer to the buffer containing the message to transmit.
length	The length of the message pointed to by the *msg* parameter. Unless the PTF for APAR PQ55591 is installed, the maximum number of bytes to be specified is 64K.
flags	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 111 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	*socket* 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.

Related Information

- “connect() — Connect a Socket” on page 108
- “fcntl() — Control Open Socket Descriptors” on page 111
- “getsockopt() — Get the Options Associated with a Socket” on page 123
- “ioctl() — Control Socket” on page 156
- “read() — Read From a Socket” on page 161
- “recv() — Receive Data on a Socket” on page 163
- “recvfrom() — Receive Messages on a Socket” on page 165
- “select() — Monitor Activity on Sockets” on page 167
- “selectex() — Monitor Activity on Sockets” on page 171
- “sendto() — Send Data on a Socket” on page 175
- “socket() — Create a Socket” on page 181
- “write() — Write Data on a Socket” on page 185

sendto() — Send Data on a Socket

```
#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
socket	The socket descriptor.
msg	The pointer to the buffer containing the message to transmit.
length	The length of the message in the buffer pointed to by the *msg* parameter. Unless the PTF for APAR PQ55591 is installed, the maximum number of bytes to be specified is 64K.
flags	reserved zero
address	The address of the target.
address_length	The size of the address pointed to by *address*.

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 111 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

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).
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.

Related Information

- “read() — Read From a Socket” on page 161
- “recv() — Receive Data on a Socket” on page 163
- “recvfrom() — Receive Messages on a Socket” on page 165
- “select() — Monitor Activity on Sockets” on page 167
- “send() — Send Data on a Socket” on page 173
- “socket() — Create a Socket” on page 181
- “write() — Write Data on a Socket” on page 185

setsockopt() — Set Options Associated with a Socket

```
#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);
```

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. Only SOL_SOCKET is supported.

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**

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 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 begin 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. When 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.

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.

Example

The following are examples of the `setsockopt()` call. See “getsockopt() — Get the Options Associated with a Socket” on page 123 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));
```

Related Information

- "fcntl() — Control Open Socket Descriptors" on page 111
- "getsockopt() — Get the Options Associated with a Socket" on page 123
- "ioctl() — Control Socket" on page 156
- "socket() — Create a Socket" on page 181

shutdown() — Shut Down a Connection

```
#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
socket	The socket descriptor.
how	The condition of the shutdown. *how* can have a value of **SHUT_RDWR**, which ends communication both to and from socket *socket*.

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	*socket* is not a valid socket descriptor.
EINVAL	The *how* 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.

Related Information

- "accept() — Accept a New Connection on a Socket" on page 90
- "close() — Close a Socket" on page 107
- "connect() — Connect a Socket" on page 108
- "socket() — Create a Socket" on page 181

socket() — Create a Socket

```
#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, AF_INET.
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 family supported is AF_INET, 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 domain.
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 domain.

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 domain, 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. This is done by setting the *Internet address* field in the **address** structure to the constant INADDR_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 `writev()`). 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 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()`, `writev()`, `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 108.

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).
EINVAL	The request is invalid or not supported.
ENOBUFS	Insufficient system resources are available to complete the call.

Example

The following are examples of the socket() call.

```
int s;
char *name;
⋮
/* Get stream socket in Internet domain with default protocol */
s = socket (AF_INET, SOCK_STREAM, 0);
⋮
```

Related Information

- "accept() — Accept a New Connection on a Socket" on page 90
- "bind() — Bind a Name to a Socket" on page 104
- "close() — Close a Socket" on page 107
- "connect() — Connect a Socket" on page 108
- "fcntl() — Control Open Socket Descriptors" on page 111
- "getsockname() — Get the Name of a Socket" on page 121
- "getsockopt() — Get the Options Associated with a Socket" on page 123
- "ioctl() — Control Socket" on page 156
- "read() — Read From a Socket" on page 161
- "recv() — Receive Data on a Socket" on page 163
- "recvfrom() — Receive Messages on a Socket" on page 165
- "select() — Monitor Activity on Sockets" on page 167
- "selectex() — Monitor Activity on Sockets" on page 171
- "send() — Send Data on a Socket" on page 173
- "shutdown() — Shut Down a Connection" on page 180
- "write() — Write Data on a Socket" on page 185

takesocket() — Acquire a Socket from Another Program

```
#define _OPEN_SYS_SOCK_EXT
#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.
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 126 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.

Related Information

- "getclientid() — Get the Identifier for the Calling Application" on page 113
- "givesocket() — Make the Specified Socket Available" on page 126

write() — Write Data on a Socket

```
#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
socket	The socket descriptor.
buf	The pointer to the buffer holding the data to be written.
N	The length in bytes of the buffer pointed to by the *buf* 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 111 or “ioctl() — Control Socket” on page 156 for a description of how to set the nonblocking mode.

When 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.
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.

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;

 number ofBytesWritten=
   write(aSocket, aString, strlen(aString));
 if (numberOfBytesWritten < 0)
 { perror("write"); return -1; }
 else
 { printf("number of bytes written is %d.\n", numberOfBytesWritten);
   return numberOfBytesWritten:
 }
}
```

Related Information

- “connect() — Connect a Socket” on page 108
- “fcntl() — Control Open Socket Descriptors” on page 111
- “getsockopt() — Get the Options Associated with a Socket” on page 123
- “ioctl() — Control Socket” on page 156
- “read() — Read From a Socket” on page 161
- “recv() — Receive Data on a Socket” on page 163
- “recvfrom() — Receive Messages on a Socket” on page 165
- “select() — Monitor Activity on Sockets” on page 167
- “selectex() — Monitor Activity on Sockets” on page 171
- “send() — Send Data on a Socket” on page 173
- “sendto() — Send Data on a Socket” on page 175
- “setsockopt() — Set Options Associated with a Socket” on page 177
- “socket() — Create a Socket” on page 181

Chapter 10. 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:

- CICS TS is required (if running under CICS)
- 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 86.

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.

Notes:

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──────────────►◄
```

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 22 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 22. Storage Definition Statement Examples

Error Messages and Return Codes

For information about error messages, see *z/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, see “ERRNO Values” on page 82.

Debugging

See Appendix B, “Debugging Facility for EZASMI and EZASOKET Interfaces (EZAAPI Trace),” on page 491.

Code CALL Instructions

This section contains the description, syntax, parameters, and other related information for each call instruction included in this API.

ACCEPT

A server issues the ACCEPT call to accept a connection request from a client. The call points to a socket that was previously created with a SOCKET call and marked by a LISTEN call.

The ACCEPT call is a blocking call. When issued, the ACCEPT call:

1. Accepts the first connection on a queue of pending connections
2. Creates a new socket with the same properties as s, and returns its descriptor in RETCODE. The original sockets remain available to the calling program to accept more connection requests.
3. The address of the client is returned in NAME for use by subsequent server calls.

Notes:

1. The blocking or nonblocking mode of a socket affects the operation of certain commands. The default is blocking; nonblocking mode can be established by use of the FCNTL and IOCTL calls. When a socket is in blocking mode, an I/O call waits for the completion of certain events. For example, a READ call will block until the buffer contains input data. When an I/O call is issued: if the socket is blocking, program processing is suspended until the event completes; if the socket is nonblocking, program processing continues.
2. If the queue has no pending connection requests, ACCEPT blocks the socket unless the socket is in nonblocking mode. The socket can be set to nonblocking by calling FCNTL or IOCTL.
3. When multiple socket calls are issued, a SELECT call can be issued prior to the ACCEPT to ensure that a connection request is pending. Using this technique ensures that subsequent ACCEPT calls will not block.
4. TCP/IP does not provide a function for screening clients. As a result, it is up to the application program to control which connection requests it accepts, but it can close a connection immediately after discovering the identity of the client.

Important Note

IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply when this program is used.

Figure 23 on page 191 shows an example of ACCEPT call instructions.

```
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.
```

Figure 23. ACCEPT Call Instructions Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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 when processed as input. The field is set to 0 when processed as output.

FAMILY
A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary 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 82 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 Note

IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply when this program is used.

Figure 24 shows an example of BIND call instructions.

```
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.
```

Figure 24. BIND Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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 when processed as input. The field is set to 0 when processed as output.

FAMILY
A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary 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 82, 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.

  Notes:

  1. If a stream socket is closed while input or output data is queued, the TCP connection is reset and data transmission may be incomplete. The SETSOCKOPT call can be used to set a *linger* condition, in which TCP/IP will continue to attempt to complete data transmission for a specified period of time after the CLOSE call is issued. See SO-LINGER in the description of “SETSOCKOPT” on page 276.
  2. A concurrent server differs from an iterative server. An iterative server provides services for one client at a time; a concurrent server receives connection requests from multiple clients and creates child servers that actually serve the clients. When a child server is created, the concurrent server obtains a new socket, passes the new socket to the child server, and 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.

Figure 25 shows an example of CLOSE call instructions.

```
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.
```

Figure 25. CLOSE Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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 82 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.
  - 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 263.

Important Note

IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply when this program is used.

Figure 26 on page 199 shows an example of CONNECT call instructions.

```
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.
```

Figure 26. CONNECT Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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 when processed as input. The field is set to 0 when processed as output.

FAMILY
A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary 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 82 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 247 for another way to control a socket’s blocking mode.

Figure 27 shows an example of FCNTL call instructions.

```
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.
```

Figure 27. FCNTL Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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 82 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 Note

This function call is not available with TCP/IP for VSE/ESA.

Figure 28 shows an example of FREEADDRINFO call instructions.

```
WORKING-STORAGE SECTION.
    01 SOC-FUNCTION PIC X(16) VALUE IS 'FREEADDRINFO'.
    01 ADDRINFO PIC 9(8) BINARY.
    01 ERRNO PIC 9(8) BINARY.
    01 RETCODE PIC S9(8) BINARY.

PROCEDURE DIVISION.
    CALL 'EZASOKET' USING SOC-FUNCTION ADDRINFO ERRNO RETCODE.
```

Figure 28. FREEADDRINFO Call Instruction Example

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 82 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.

Important Note

This function call is not available with TCP/IP for VSE/ESA.

Figure 29 shows an example of GETADDRINFO call instructions.

```
WORKING-STORAGE SECTION.
    01 SOC-FUNCTION PIC X(16) VALUE IS 'GETADDRINFO'.
    01 NODE PIC X(255).
    01 NODELEN PIC 9(8) BINARY.
    01 SERVIC PIC X(32).
    01 SERVLEN 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.
```

Figure 29. GETADDRINFO Call Instruction Example (Part 1 of 2)

```
PROCEDURE DIVISION.
    MOVE 'www.hostname.com' TO NODE.
    MOVE 16 TO NODELEN.
    MOVE 'TELNET' TO SERVIC.
    MOVE 6 TO SERVLEN.
    SET HINTS TO ADDRESS OF HINTS-ADDRINFO.
    CALL 'EZASOKET' USING SOC-FUNCTION NODE NODELEN SERVIC SERVLEN HINTS RES CANNLEN
    ERRNO RETCODE.
```

Figure 29. GETADDRINFO Call Instruction Example (Part 2 of 2)

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 SERVLEN. The SERVIC name being queried consists of up to SERVLEN or up to the first binary zero.

SERVLEN

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 of the bitwise, OR of one or more of the following:

AI-PASSIVE (X'00000001') or a decimal value of 1.
Specifies how to fill in the NAME pointed to by the returned RES.

If this flag is specified, 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 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. When the AI-ALL flag is not also specified, if no IPv6 addresses are found, a query is made for IPv4 addresses. If IPv4 addresses are found, they are returned as IPv4-mapped IPv6 addresses.

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

AI-ALL (X'00000020') or a decimal value of 32.
When the AF field has a value of AF_INET6 and AI-ALL is set, the AI-V4MAPPED flag must also be set to indicate that the caller will accept all addresses (IPv6 and IPv4-mapped IPv6 addresses). When the AF field has a value of AF_UNSPEC when the system supports IPv6 and

AI-ALL is set, the caller accepts IPv6 addreses 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-ADDRCONFG 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 when 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.

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 SOCTYPE 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 SOCTYPE 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-ADDRCONFG 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 CANONNAME 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 82 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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

The ADDRINFO structure uses indirect addressing to return a variable number of NAMES. If you are coding in PL/I or assembler language, this structure can be processed in a relatively straight-forward manner. If you are coding in COBOL, this structure might be difficult to interpret. You can use the subroutine "EZACIC09" on page 296 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 and TAKESOCKET calls. See "GIVESOCKET" on page 229 for a discussion of the use of GIVESOCKET and TAKESOCKET calls.

If GETCLIENTID is called by a server, the identifier of the *caller* (not necessarily the *client*) is returned.

> **Important Note**
>
> IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply when this program is used.

Figure 30 shows an example of GETCLIENTID call instructions.

```
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.
```

Figure 30. GETCLIENTID Call Instruction Example

For equivalent PL/I and assembler language declarations, see "Converting Parameter Descriptions" on page 189.

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 82 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.

Figure 31 shows an example of GETHOSTBYADDR call instructions.

```
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.
```

Figure 31. GETHOSTBYADDR Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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 82 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 returns the HOSTENT structure shown in Figure 32 on page 214.

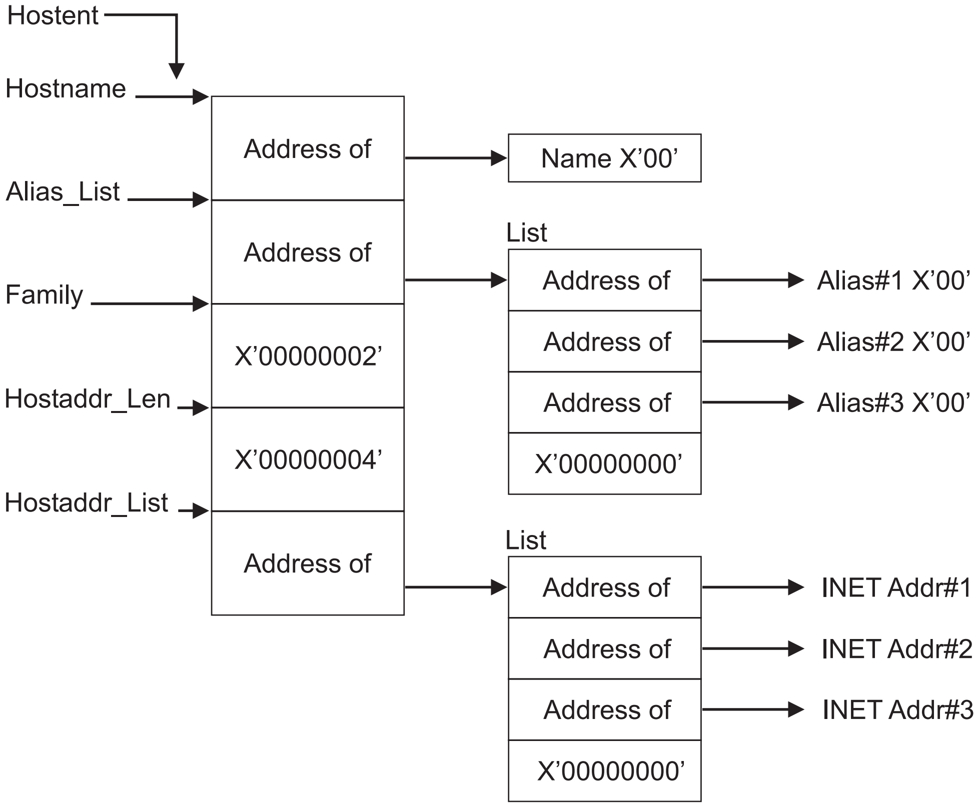


Figure 32. HOSTENT Structure Returned by the GETHOSTBYADDR Call

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'.

  Note: 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 293.

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. 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 an 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.

Figure 33 shows an example of GETHOSTBYNAME call instructions.

```
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.
```

Figure 33. GETHOSTBYNAME Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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

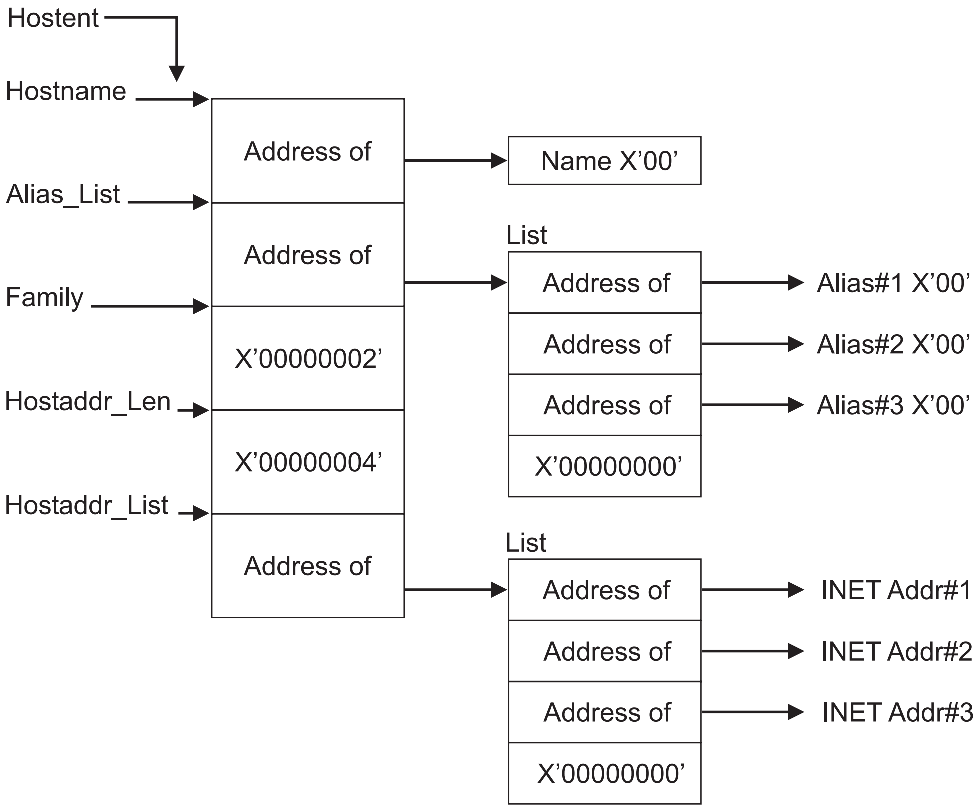


Figure 34. HOSTENT Structure Returned by the GETHOSTYBYNAME Call

GETHOSTBYNAME returns the HOSTENT structure shown in Figure 34. 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'.

  Note: 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 293.

GETHOSTID

The GETHOSTID call returns the 32-bit internet address for the current host.

Figure 35 shows an example of GETHOSTID call instructions.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'GETHOSTID'.
    01  RETCODE         PIC S9(8) BINARY.

PROCEDURE
     CALL 'EZASOKET' USING SOC-FUNCTION RETCODE.
```

Figure 35. GETHOSTID Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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.

Figure 36 shows an example of GETHOSTNAME call instructions.

```
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.
```

Figure 36. GETHOSTNAME Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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.

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See “ERRNO Values” on page 82 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

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 Note

This function call is not available with TCP/IP for VSE/ESA.

Figure 37 shows an example of GETNAMEINFO call instructions.

```
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 SERVLEN PIC 9(8) BINARY.
    01 FLAGS PIC 9(8) BINARY VALUE 0.
    01 NI-NOFQDN PIC 9(8) BINARY VALUE 1.
    01 NI-NUMERICHOST PIC 9(8) BINARY VALUE 2.
    01 NI-NAMEREQD PIC 9(8) BINARY VALUE 4.
    01 NI-NUMERICSERVER PIC 9(8) BINARY VALUE 8.
    01 NI-DGRAM PIC 9(8) BINARY VALUE 16.
    01 NI-NUMERICSCOPE PIC 9(8) BINARY VALUE 32.

* IPv4 socket structure.
    01 NAME.
       03 FAMILY PIC 9(4) BINARY.
       03 PORT PIC 9(4) BINARY.
       03 IP-ADDRESS PIC 9(8) BINARY.
       03 RESERVED PIC X(8).

* IPv6 socket structure.
    01 NAME.
       03 FAMILY PIC 9(4) BINARY.
       03 PORT PIC 9(4) BINARY.
       03 FLOWINFO PIC 9(8) BINARY.
       03 IP-ADDRESS.
          10 FILLER PIC 9(16) BINARY.
          10 FILLER PIC 9(16) BINARY.
       03 SCOPE-ID PIC 9(8) BINARY.
    01 ERRNO PIC 9(8) BINARY.
    01 RETCODE PIC S9(8) BINARY.

PROCEDURE DIVISION.
    MOVE 28 TO NAMELEN.
    MOVE 255 TO HOSTLEN.
    MOVE 32 TO SERVLEN.
    MOVE NI-NAMEREQD TO FLAGS.
    CALL 'EZASOKET' USING SOC-FUNCTION NAME NAMELEN HOST
         HOSTLEN SERVIC SERVLEN FLAGS ERRNO RETCODE.
```

Figure 37. GETNAMEINFO Call Instruction Example

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 when processed as input. The field is set to 0 when processed as output.

FAMILY
A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary number that specifies the port number.

FLOW-INFO
This field is ignored by the TYPE=GETNAMEINFO macro.

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

SCOPE-ID
A fullword binary field that specifies the scope for an IPv6 address as an interface index. The resolver ignores the SCOPE_ID field, unless the address in IPv6-ADDRESS is a link-local address and the HOST parameter 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 SERVLEN parameters

Otherwise, an error occurs. The HOST name being queried consists of up to HOSTLEN or up to the first binary 0.

If the IPv6-ADDRESS value is a link-local address, and the SCOPE_ID interface index is nonzero, scope information is appended to the resolved host name using the format *host%scope information*. The scope information can be the numeric form of the SCOPE_ID interface index or the interface name that is associated with the SCOPE_ID interface index. Use the NI_NUMERICSCOPE option to select which form is returned. The combined host name and scope information is 255 bytes or less. For more information about scope information and TYPE=GETNAMEINFO processing.

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 SERVLEN 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 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 82 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 Note**
>
> IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply when this program is used.

Figure 38 shows an example of GETPEERNAME call instructions.

```
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.
```

Figure 38. GETPEERNAME Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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.

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 when processed as input. The field is set to 0 when processed as output.

FAMILY
A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary 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 82 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 Note**
>
> IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply when this program is used.

Since a stream socket is not assigned a name until after a successful call to either BIND, CONNECT, or ACCEPT, the GETSOCKNAME call can be used after an implicit bind to discover which port was assigned to the socket.

Figure 39 shows an example of GETSOCKNAME call instructions.

```
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.
```

Figure 39. GETSOCKNAME Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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 address structure. Any value specified by the use of this field is ignored when processed as input. The field is set to 0 when processed as output.

FAMILY
A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary 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 82 for information about ERRNO return codes.

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

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

GETSOCKOPT

The GETSOCKOPT call queries the options that are set by the SETSOCKOPT call.

Several options are associated with each socket. These options are described below. You must specify the option to be queried when you issue the GETSOCKOPT call.

Figure 40 shows an example of GETSOCKOPT call instructions.

```
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.
```

Figure 40. GETSOCKOPT Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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:

SO-LINGER Requests the status of LINGER.

- When 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 276.

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 82 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 Note

IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply when this program is used.

Figure 41 shows an example of GIVESOCKET call instructions.

```
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.
```

Figure 41. GIVESOCKET Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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.

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 82, 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 a previous call to an SSL function.

For example, the distinguished name returned in the null-terminated string by the GSKGETDNBYLAB call must be freed using GSKFREEMEM.

Figure 42 shows an example of the GSKFREEMEM call instruction:

```
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.
```

Figure 42. GSKFREEMEM Call Instruction Example

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.

Figure 43 shows an example of the GSKGETCIPHINF call instruction:

```
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.
```

Figure 43. GSKGETCIPHINF Call Instruction Example

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).

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.

Figure 44 shows an example of the GSKGETDNBYLAB call instruction:

```
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.
```

Figure 44. GSKGETDNBYLAB Call Instruction Example

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.

Figure 45 shows an example of the GSKINIT call instruction:

```
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.
```

Figure 45. GSKINIT Call Instruction Example

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 verifcation.

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 when 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.

Figure 46 shows an example of the GSKSSOCCLOSE call instruction:

```
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.
```

Figure 46. GSKSSOCCLOSE Call Instruction Example

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.

Figure 47 shows an example of the GSKSSOCINIT call instruction:

```
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  V3CIPHSEL     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 V3CIPHSEL CERTINFO REASCODE
                 ERRNO RETCODE.
```

Figure 47. GSKSSOCINIT Call Instruction Example

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 EZASOKET 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 when 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

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
When 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.

Figure 48 shows an example of the GSKSSOCREAD call instruction:

```
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.
```

Figure 48. GSKSSOCREAD Call Instruction Example

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.

Figure 49 shows an example of the GSKSSOCRESET call instruction:

```
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.
```

Figure 49. GSKSSOCRESET Call Instruction Example

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.

Figure 50 shows an example of the GSKSSOCWRITE call instruction:

```
WORKING-STORAGE SECTION.
    01  SOC-FUNCTION  PIC X(16) VALUE 'GSKSSOCWRITE    '.
    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.
```

Figure 50. GSKSSOCWRITE Call Instruction Example

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.

Figure 51 shows an example of the GSKUNINIT call instruction:

```
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.
```

Figure 51. GSKUNINIT Call Instruction Example

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

Figure 52 shows an example of INITAPI call instructions.

```
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.
```

Figure 52. INITAPI Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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
Optional 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. Please note that this specification can be overwritten with the following JCL statement: // SETPARM 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 jobname 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. 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 82 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.

Figure 53 shows an example of IOCTL call instructions.

```
   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.
   01  RETCODE                PIC 9(8) BINARY.


PROCEDURE
     CALL 'EZASOKET' USING SOC-FUNCTION S COMMAND REQARG
           RETARG ERRNO RETCODE.
```

Figure 53. IOCTL Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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 7 on page 365.

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 7 on page 365.

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 82 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.

Note: The LISTEN call is not supported for datagram sockets or raw sockets.

The LISTEN call is typically used by a server to receive connection requests from clients. When a connection request is received, a new socket is created by a subsequent ACCEPT call, and the original socket continues to listen for additional connection requests. The LISTEN call converts an active socket to a passive socket and conditions it to accept connection requests from clients. Once a socket becomes passive it cannot initiate connection requests.

Figure 54 shows an example of LISTEN call instructions.

```
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.
```

Figure 54. LISTEN Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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 82 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 Note

This function call is not available with TCP/IP for VSE/ESA.

Figure 55 shows an example of NTOP call instructions.

```
WORKING-STORAGE SECTION.
    01 SOC-ACCEPT-FUNCTION PIC X(16) VALUE IS 'ACCEPT'.
    01 SOC-NTOP-FUNCTION PIC X(16) VALUE IS 'NTOP'.
    01 S PIC 9(4) BINARY.

* IPv4 socket structure.
    01 NAME.
       03 FAMILY PIC 9(4) BINARY.
       03 PORT PIC 9(4) BINARY.
       03 IP-ADDRESS PIC 9(8) BINARY.
       03 RESERVED PIC X(8).

* IPv6 socket structure.
    01 NAME.
       03 FAMILY PIC 9(4) BINARY.
       03 PORT PIC 9(4) BINARY.
       03 FLOWINFO PIC 9(8) BINARY.
       03 IP-ADDRESS.
          10 FILLER PIC 9(16) BINARY.
          10 FILLER PIC 9(16) BINARY.
       03 SCOPE-ID PIC 9(8) BINARY.
    01 NTOP-FAMILY PIC 9(8) BINARY.
    01 ERRNO PIC 9(8) BINARY.
    01 RETCODE PIC S9(8) BINARY.
    01 PRESENTABLE-ADDRESS PIC X(45).
    01 PRESENTABLE-ADDRESS-LEN PIC 9(4) BINARY.

PROCEDURE DIVISION.
    CALL 'EZASOKET' USING SOC-ACCEPT-FUNCTION S NAME ERRNO RETCODE.
    CALL 'EZASOKET' USING SOC-NTOP-FUNCTION NTOP-FAMILY IP-ADDRESS
          PRESENTABLE-ADDRESS PRESENTABLE-ADDRESS-LEN ERRNO RETCODE.
```

Figure 55. NTOP Call Instruction Example

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 82 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 Note

This function call is not available with TCP/IP for VSE/ESA.

Figure 56 shows an example of PTON call instructions.

```
WORKING-STORAGE SECTION.
    01 SOC-BIND-FUNCTION PIC X(16) VALUE IS 'BIND'.
    01 SOC-PTON-FUNCTION PIC X(16) VALUE IS 'PTON'.
    01 S PIC 9(4) BINARY.

* IPv4 socket structure.
    01 NAME.
       03 FAMILY PIC 9(4) BINARY.
       03 PORT PIC 9(4) BINARY.
       03 IP-ADDRESS PIC 9(8) BINARY.
       03 RESERVED PIC X(8).

* IPv6 socket structure.
    01 NAME.
       03 FAMILY PIC 9(4) BINARY.
       03 PORT PIC 9(4) BINARY.
       03 FLOWINFO PIC 9(8) BINARY.
       03 IP-ADDRESS.
          10 FILLER PIC 9(16) BINARY.
          10 FILLER PIC 9(16) BINARY.
       03 SCOPE-ID PIC 9(8) BINARY.
    01 AF-INET PIC 9(8) BINARY VALUE 2.
    01 AF-INET6 PIC 9(8) BINARY VALUE 19.

* IPv4 address.
    01 PRESENTABLE-ADDRESS PIC X(45).
    01 PRESENTABLE-ADDRESS-IPV4 REDEFINES PRESENTABLE-ADDRESS.
       05 PRESENTABLE-IPV4-ADDRESS PIC X(15) VALUE '192.26.5.19'.
       05 FILLER PIC X(30).
    01 PRESENTABLE-ADDRESS-LEN PIC 9(4) BINARY VALUE 11.

* IPv6 address.
    01 PRESENTABLE-ADDRESS PIC X(45) VALUE '12f9:0:0:c30:123:457:9cb:1112'.
    01 PRESENTABLE-ADDRESS-LEN PIC 9(4) BINARY VALUE 29.

* IPv4-mapped IPv6 address.
    01 PRESENTABLE-ADDRESS PIC X(45) VALUE '12f9:0:0:c30:123:457:192.26.5.19'.
    01 PRESENTABLE-ADDRESS-LEN PIC 9(4) BINARY VALUE 32.
    01 ERRNO PIC 9(8) BINARY.
    01 RETCODE PIC S9(8) BINARY.
```

Figure 56. PTON Call Instruction Example (Part 1 of 2)

```
PROCEDURE DIVISION.
* IPv4 address.
    CALL 'EZASOKET' USING SOC-PTON-FUNCTION AF-INET PRESENTABLE-ADDRESS
    PRESENTABLE-ADDRESS-LEN IP-ADDRESS ERRNO RETURN-CODE.

* IPv6 address.
    CALL 'EZASOKET' USING SOC-PTON-FUNCTION AF-INET6 PRESENTABLE-ADDRESS
                    PRESENTABLE-ADDRESS-LEN IP-ADDRESS ERRNO RETURN-CODE.
    CALL 'EZASOKET' USING SOC-BIND-FUNCTION S NAME ERRNO RETURN-CODE.
```

Figure 56. PTON Call Instruction Example (Part 2 of 2)

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 82 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 290 for a subroutine that will translate ASCII input data to EBCDIC.

Figure 57 shows an example of READ call instructions.

```
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.
```

Figure 57. READ Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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 82 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.

Figure 58 shows an example of READV call instructions.

```
WORKING-STORAGE SECTION.
        01 SOC-FUNCTION       PIC X(16) VALUE 'READV           '.
        01 S                  PIC 9(4) BINARY.
        01 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 'CEEGTST' USING HEAPID, IOV-LENGTH(1), IOV-POINTER(1),
                            FC.
       CALL 'CEEGTST' USING HEAPID, IOV-LENGTH(2), IOV-POINTER(2),
                            FC.
       CALL 'CEEGTST' USING HEAPID, IOV-LENGTH(3), IOV-POINTER(3),
                            FC.
       CALL 'CEEGTST' USING HEAPID, IOV-LENGTH(4), IOV-POINTER(4),
                            FC.
       CALL 'CEEGTST' USING HEAPID, IOV-LENGTH(5), IOV-POINTER(5),
                            FC.

       CALL 'EZASOKET' USING SOC-FUNCTION S IOV IOVCNT
                     ERRNO RETCODE.
```

Figure 58. READV Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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 82 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 occured. 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 201 or “IOCTL” on page 247 for a description of how to set nonblocking mode.

Note: See “EZACIC05” on page 290 for a subroutine that will translate ASCII input data to EBCDIC.

Figure 59 shows an example of RECV call instructions.

```
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.
```

Figure 59. RECV Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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 zet 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 82 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 201 or “IOCTL” on page 247 for a description of how to set nonblocking mode.

Important Note

IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply when this program is used.

Note: See “EZACIC05” on page 290 for a subroutine that will translate ASCII input data to EBCDIC.

Figure 60 on page 261 shows an example of RECVFROM call instructions.

```
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.
```

Figure 60. RECVFROM Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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 when processed as input. The field is set to 0 when processed as output.

FAMILY
A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary 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 82 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 when one of the tested sockets is ready; thereby ensuring that the I/O call will not block.

To use the SELECT call as a timer in your program, do one of the following:

- Set the read, write, and except arrays to zeros.
- Specify MAXSOC <= 0.

Defining Which Sockets to Test

The SELECT call monitors for read operations, write operations, and exception operations:

- When a socket is ready to read, one of the following has occurred:
  - A buffer for the specified sockets contains input data. If input data is available for a given socket, a read operation on that socket will not block.
  - A connection has been requested on that socket.
- When a socket is ready to write, TCP/IP can accommodate additional output data. If TCP/IP can accept additional output for a given socket, a write operation on that socket will not block.
- When an exception condition has occurred on a specified socket it is an indication that a TAKESOCKET has occurred for that socket.

Each socket descriptor is represented by a bit in a bit string. The bit strings are contained in 32-bit fullwords, numbered from right to left. The rightmost bit represents socket descriptor 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 291.

Read Operations

Read operations include ACCEPT, READ, RECV, or RECVFROM calls. A socket is ready to be read when data has been received for it, or when a connection request has occurred.

To test whether any of several sockets is ready for reading, set the appropriate bits in RSNDMSK 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) when:

- TCP/IP can accept additional outgoing data.

- The socket is marked nonblocking and a previous CONNECT did not complete immediately. In this case, CONNECT returned an ERRNO with a value EINPROGRESS. This socket will be selected for write when the CONNECT completes.

A call to WRITE, SEND, or SENDTO blocks when the amount of data to be sent exceeds the amount of data TCP/IP can accept. To avoid this, you can precede the write operation with a SELECT call to ensure that the socket is ready for writing.

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. When this condition is selected, the calling program (concurrent server) should issue CLOSE to dissociate itself from the socket.
- A socket has received out-of-band data. On this condition, a READ will return the out-of-band data ahead of program data.

To test whether any of several sockets have an exception condition, set the ESNDMSK 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.

Figure 61 on page 265 shows an example of SELECT call instructions.

```
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  RSNDMSK        PIC X(*).
    01  WSNDMSK        PIC X(*).
    01  ESNDMSK        PIC X(*).
    01  RRETMSK        PIC X(*).
    01  WRETMSK        PIC X(*).
    01  ERETMSK        PIC X(*).
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

PROCEDURE
     CALL 'EZASOKET' USING SOC-FUNCTION MAXSOC TIMEOUT
                    RSNDMSK WSNDMSK ESNDMSK
                    RRETMSK WRETMSK ERETMSK
                    ERRNO RETCODE.
```

* The bit mask lengths can be determined from the expression:

```
((maximum socket number +32)/32 (drop the remainder))*4
```

Figure 61. SELECT Call Instruction Example

Bit masks are 32-bit fullwords with one bit for each socket. Up to 32 sockets fit into one 32-bit mask [PIC X(4)]. If you have 33 sockets, you must allocate two 32-bit masks [PIC X(8)].

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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.

RSNDMSK
A bit string sent to request read event status.

- For each socket to be checked for pending read events, the corresponding bit in the string should be set to 1.
- For sockets to be ignored, the value of the corresponding bit should be set to 0.

If this parameter is set to all zeros, the SELECT will not check for read events.

WSNDMSK
A bit string sent to request write event status.

- For each socket to be checked for pending write events, the corresponding bit in the string should be set to 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 82 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 when either one of the sockets has activity, the time value expires, or one of the ECBs is posted.

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

- Set the read, write, and except arrays to zeros
- Specify MAXSOC <= 0

For a detailed description on testing sockets, refer to the description of “SELECT” on page 263.

Figure 62 shows an example of SELECTEX call instructions.

```
 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  RSNDMSK        PIC X(*).
     01  WSNDMSK        PIC X(*).
     01  ESNDMSK        PIC X(*).
     01  RRETMSK        PIC X(*).
     01  WRETMSK        PIC X(*).
     01  ERETMSK        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
                    RSNDMSK WSNDMSK ESNDMSK
                    RRETMSK WRETMSK ERETMSK
                    SELECB ERRNO RETCODE.
```

* The bit mask lengths can be determined from the expression:

`((maximum socket number +32)/32 (drop the remainder))*4`

Figure 62. SELECTEX Call Instruction Example

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:

- 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 82 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 289 for a subroutine that will translate EBCDIC input data to ASCII.

Figure 63 shows an example of SEND call instructions.

```
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.
```

Figure 63. SEND Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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.

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 82 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 289 for a subroutine that will translate EBCDIC input data to ASCII.

Important Note

IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply when this program is used.

Figure 64 shows an example of SENDTO call instructions.

```
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.
```

Figure 64. SENDTO Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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 when processed as input. The field is set to 0 when processed as output.

FAMILY
A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary 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 82 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 domain.

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.

Figure 65 shows an example of SETSOCKOPT call instructions.

```
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.
```

Figure 65. SETSOCKOPT Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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.

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.

- When LINGER is enabled and CLOSE is called, the calling program is blocked until the data is successfully transmitted or the connection has timed out.

- When 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.
```

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 82 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

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.

When the CLOSE call is used, the SETSOCKOPT OPTVAL LINGER parameter determines the amount of time the system will wait before releasing the connection. For example, with a LINGER value of 30 seconds, system resources 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, rather than waiting for the 30 second delay.

Figure 66 shows an example of SHUTDOWN call instructions.

```
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.
```

Figure 66. SHUTDOWN Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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 82 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 Note

IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply when this program is used.

Figure 67 shows an example of SOCKET call instructions.

```
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.
```

Figure 67. SOCKET Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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**

'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.

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 82, 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 229 for a discussion of the use of GIVESOCKET and TAKESOCKET calls.

Note: When TAKESOCKET is issued, a new socket descriptor is returned in RETCODE. You should use this new socket descriptor in subsequent calls such as GETSOCKOPT, which require the S (socket descriptor) parameter.

> **Important Note**
>
> IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply when this program is used.

Figure 68 shows an example of TAKESOCKET call instructions.

```
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.
```

Figure 68. TAKESOCKET Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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 82 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

TERMAPI

This call terminates the session created by INITAPI.

Figure 69 shows an example of TERMAPI call instructions.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'TERMAPI'.

PROCEDURE
     CALL 'EZASOKET' USING SOC-FUNCTION.
```

Figure 69. TERMAPI Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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 289 for a subroutine that will translate EBCDIC output data to ASCII.

Figure 70 shows an example of WRITE call instructions.

```
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.
```

Figure 70. WRITE Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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 82 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.

Figure 71 shows an example of WRITEV call instructions.

```
WORKING-STORAGE SECTION.
       01 SOC-FUNCTION       PIC X(16) VALUE 'WRITEV          '.
       01 S                  PIC 9(4) BINARY.
       01 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 'CEEGTST' USING HEAPID, IOV-LENGTH(1), IOV-POINTER(1),
                           FC.
      CALL 'CEEGTST' USING HEAPID, IOV-LENGTH(2), IOV-POINTER(2),
                           FC.
      CALL 'CEEGTST' USING HEAPID, IOV-LENGTH(3), IOV-POINTER(3),
                           FC.
      CALL 'CEEGTST' USING HEAPID, IOV-LENGTH(4), IOV-POINTER(4),
                           FC.
      CALL 'CEEGTST' 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.
```

Figure 71. WRITEV Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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 82 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:

- 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.

Figure 72 shows an example of EZACIC04 call instructions.

```
WORKING STORAGE
    01  OUT-BUFFER   PIC X(length of output).
    01  LENGTH       PIC 9(8) BINARY.

PROCEDURE
     CALL 'EZACIC04' USING OUT-BUFFER LENGTH.
```

Figure 72. EZACIC04 Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

OUT-BUFFER
A buffer that contains the following:
- When called – EBCDIC data
- Upon return – ASCII data

LENGTH
Specifies the length of the data to be translated.

EZACIC05

The EZACIC05 program is used to translate ASCII data to EBCDIC data. EBCDIC data is required by COBOL, PL/I, and assembler language programs.

Figure 73 shows an example of EZACIC05 call instructions.

```
WORKING STORAGE
    01  IN-BUFFER    PIC X(length of output)
    01  LENGTH       PIC 9(8) BINARY VALUE

PROCEDURE
     CALL 'EZACIC05' USING IN-BUFFER LENGTH.
```

Figure 73. EZACIC05 Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

IN-BUFFER
A buffer that contains the following:

- When called – ASCII data
- Upon return – EBCDIC data.

LENGTH
Specifies the length of the data to be translated.

EZACIC06

The SELECT call uses bit strings to specify the sockets to test and to return the results of the test. Because bit strings are difficult to manage in COBOL, you might want to use the assembler language program EZACIC06 to translate them to character strings to be used with the SELECT call.

Figure 74 shows an example of EZACIC06 call instructions.

```
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.
```

Figure 74. EZACIC06 Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

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.

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.

Figure 75 on page 294 shows an example of EZACIC08 call instructions.

```
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
```

Figure 75. EZAZIC08 Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 189.

Parameter Values set by the Application

HOSTENT-ADDR

This fullword binary field must contain the address of the HOSTENT structure (as returned by the GETHOSTBY*xxxx* 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.

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.

Figure 76 shows an example of EZACIC09 call instructions.

```
WORKING-STORAGE SECTION.
*
* Variables used for the GETADDRINFO call
*
 01 getaddrinfo-parms.
    02 node-name pic x(255).
    02 node-name-len pic 9(8) binary.
    02 service-name pic x(32).
    02 service-name-len pic 9(8) binary.
    02 canonical-name-len pic 9(8) binary.
    02 ai-passive pic 9(8) binary value 1.
    02 ai-canonnameok pic 9(8) binary value 2.
    02 ai-numerichost pic 9(8) binary value 4.
    02 ai-numericserv 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.
*
```

Figure 76. EZACIC09 Call Instruction Example (Part 1 of 3)

```
* 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.
```

Figure 76. EZACIC09 Call Instruction Example (Part 2 of 3)

```
PROCEDURE DIVISION USING L1.
*
* Get and address from the resolver.
*
     move 'yournodename' to node-name.
     move 12 to node-name-len.
     move spaces to service-name.
     move 0 to service-name-len.
     move af-inet6 to hints-ai-family.
     move 49 to hints-ai-flags move 0 to hints-ai-socktype.
     move 0 to hints-ai-protocol.
     set address of results-addrinfo to res-addrinfo-ptr.
     set hints-addrinfo-ptr to address of hints-addrinfo.
     call 'EZASOKET' using soket-getaddrinfo node-name node-name-len
                     service-name service-name-len hints-addrinfo-ptr
                     res-addrinfo-ptr canonical-name-len errno retcode.
*
* Use EZACIC09 to extract the IP address
*
     set address of results-addrinfo to res-addrinfo-ptr.
     set res to address of results-addrinfo.
     move zeros to res-name-len.
     move spaces to res-canonical-name.
     set res-name to nulls.
     set res-next-addrinfo to nulls.
     call 'EZACIC09' using res res-name-len res-canonical-name
                     res-name res-next-addrinfo retcode.
     set address of output-ip-name to res-name.
     move output-ipv6-ipaddr to server-ipaddr.
```

Figure 76. EZACIC09 Call Instruction Example (Part 3 of 3)

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 11. 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:
- CICS/TS (if running under CICS)
- 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, please refer to “CICS Considerations for the EZA Interfaces” on page 86.

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.

Notes:

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.

Note: 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 when asynchronous macro calls are used.

The ECB input parameter for asynchronous calls must point to a 160- byte storage area:

ECB (4 bytes)	Storage Area (156 bytes)

Figure 77. 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, see *z/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, see “ERRNO Values” on page 82.

Debugging

See Appendix B, “Debugging Facility for EZASMI and EZASOKET Interfaces (EZAAPI Trace),” on page 491.

Macros for Assembler Programs

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

ACCEPT

The ACCEPT macro is issued when the server receives a connection request from a client. ACCEPT points to a socket that was created with a SOCKET macro and marked by a LISTEN macro. If a process waits for the completion of connection requests from several peer processes, a later ACCEPT macro can block until one of the CONNECT macros completes. To avoid this, issue a SELECT macro between the CONNECT and the ACCEPT macros. Concurrent server programs use the ACCEPT macro to pass connection requests to subtasks.

> **Important Note**
>
> IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply when 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.

```
►►──EZASMI──TYPE=ACCEPT──,S──=──┬─number───┬──,NAME──=──┬─address──┬──────────────────►
                                ├─address──┤            ├─*indaddr─┤
                                ├─*indaddr─┤            └─(reg)────┘
                                └─(reg)────┘

►─,ERRNO──=──┬─address──┬──,RETCODE──=──┬─address──┬──┬──────────────────────────┬────►
             ├─*indaddr─┤               ├─*indaddr─┤  └─,NS──=──┬─address──┬─────┘
             └─(reg)────┘               └─(reg)────┘            ├─*indaddr─┤
                                                                └─(reg)────┘

►──┬───────────────────────┬──┬───────────────────────────┬─────────────────────────►
   └─,ECB=──┬─address──┬───┘  └─,ERROR──=──┬─address──┬───┘
            ├─*indaddr─┤                   ├─*indaddr─┤
            └─(reg)────┘                   └─(reg)────┘

►──┬────────────────────────────┬───────────────────────────────────────────────────►◄
   └─,TASK──=──┬─address──┬─────┘
               ├─*indaddr─┤
               └─(reg)────┘
```

Keyword	Description
S	Input parameter. A value or the address of a halfword binary number specifying the descriptor of the socket from which the connection is accepted.
NAME	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.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:

Field **Description**

NAMELEN
A 1-byte binary field that specifies the length of the IPv6 socket address structure. Any value specified by the use of this field is ignored when processed as input. The field is set to 0 when processed as output.

FAMILY
A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary 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 82 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.

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 when the application programming interface (API) processing module cannot be loaded.

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

BIND

In a server program, the BIND macro normally follows a SOCKET macro to complete the new socket creation process.

The BIND macro can specify the port or let the system choose the port. A listener program should always bind to the same well-known port so that clients know the socket address to use when issuing a CONNECT 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 Note**
>
> IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply when this program is used.

```
►►──EZASMI──TYPE=BIND──,S──=──┬─number───┬──,NAME──=──┬─address──┬──────────────────►
                              ├─address──┤            ├─*indaddr─┤
                              ├─*indaddr─┤            └─(reg)────┘
                              └─(reg)────┘

►─,ERRNO──=──┬─address──┬──,RETCODE──=──┬─address──┬──┬────────────────────────┬──────►
             ├─*indaddr─┤               ├─*indaddr─┤  └─,ECB=──┬─address──┬──┘
             └─(reg)────┘               └─(reg)────┘           ├─*indaddr─┤
                                                               └─(reg)────┘

►─┬───────────────────────────┬──┬──────────────────────────┬──────────────────────►◄
  └─,ERROR──=──┬─address──┬──┘  └─,TASK──=──┬─address──┬──┘
               ├─*indaddr─┤                 ├─*indaddr─┤
               └─(reg)────┘                 └─(reg)────┘
```

Keyword **Description**

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:

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 when processed as input. The field is set to 0 when processed as output.

FAMILY
A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary 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.

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 82, 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 when the application programming interface (API) processing module cannot be loaded.

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

CANCEL

The CANCEL function terminates a call in progress. The call being cancelled must have specified ECB .

```
►►──EZASMI──TYPE=CANCEL──,CALAREA──=──┬─address──┬──┬──────────────────────┬──────────►
                                      ├─*indaddr─┤  └─,ECB=─┬─address──┬─┘
                                      └─(reg)────┘          ├─*indaddr─┤
                                                            └─(reg)────┘

►─,ERRNO──=──┬─address──┬──,RETCODE──=──┬─address──┬──────────────────────────────────►
             ├─*indaddr─┤               ├─*indaddr─┤
             └─(reg)────┘               └─(reg)────┘

►──┬───────────────────────────┬──┬──────────────────────────┬────────────────────────►◄
   └─,ERROR──=──┬─address──┬──┘  └─,TASK──=──┬─address──┬──┘
                ├─*indaddr─┤                 ├─*indaddr─┤
                └─(reg)────┘                 └─(reg)────┘
```

Keyword **Description**

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 82, 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 when the application programming interface (API) processing module cannot be loaded.

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

CLOSE

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

CLOSE can also be issued by a concurrent server after it gives a socket to a subtask program. After issuing GIVESOCKET and receiving notification that the client child has successfully issued TAKESOCKET, the concurrent server issues the CLOSE macro to complete the transfer of ownership.

Note: If a stream socket is closed while input or output data is queued, the stream connection is reset and data transmission can be incomplete. 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 394.

```
►►──EZASMI──TYPE=CLOSE──,S──=──┬─number───┬──,ERRNO──=──┬─address──┬──────────────►
                               ├─address──┤             ├─*indaddr─┤
                               ├─*indaddr─┤             └─(reg)────┘
                               └─(reg)────┘

►─,RETCODE──=──┬─address──┬──┬─────────────────────────┬──────────────────────────►
               ├─*indaddr─┤  └─,ECB=──┬─address──┬─────┘
               └─(reg)────┘           ├─*indaddr─┤
                                      └─(reg)────┘

►──┬─────────────────────────────┬──┬─────────────────────────────┬───────────────►◄
   └─,ERROR──=──┬─address──┬─────┘  └─,TASK──=──┬─address──┬─────┘
                ├─*indaddr─┤                    ├─*indaddr─┤
                └─(reg)────┘                    └─(reg)────┘
```

Keyword	Description
S	Input parameter. A value or the address of a halfword binary number specifying the socket to be closed.
ERRNO	Output parameter. A fullword binary field. If RETCODE is negative, ERRNO contains a valid error number. Otherwise, ignore ERRNO. See “ERRNO Values” on page 82, for information about ERRNO return codes.
RETCODE	A fullword binary field that returns one of the following:
ECB	Input parameter. It points to a 160-byte field containing:

RETCODE values:

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

ECB field contents:

- 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 when the application programming interface (API) processing module cannot be loaded.

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

CONNECT

The CONNECT macro is used by a client to establish a connection between a local socket and a remote socket.

> **Important Note**
> IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply when 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.

```
►►──EZASMI──TYPE=CONNECT──,S──=──┬─number───┬──,NAME──=──┬─address──┬────────────────►
                                 ├─address──┤            ├─*indaddr─┤
                                 ├─*indaddr─┤            └─(reg)────┘
                                 └─(reg)────┘

►─,ERRNO──=──┬─address──┬──,RETCODE──=──┬─address──┬──┬──────────────────────┬──────►
             ├─*indaddr─┤               ├─*indaddr─┤  └─,ECB=──┬─address──┬──┘
             └─(reg)────┘               └─(reg)────┘           ├─*indaddr─┤
                                                               └─(reg)────┘
```

```
►─┬──────────────────────────┬──┬─────────────────────────┬──────────────────────────►◄
  └─,ERROR──=──┬─address──┬─┘  └─,TASK──=──┬─address──┬─┘
               ├─*indaddr─┤                ├─*indaddr─┤
               └─(reg)────┘                └─(reg)────┘
```

Keyword **Description**

S Input parameter. A value or the address of a halfword binary number specifying the socket descriptor.

NAME Input parameter. The NAME parameter for CONNECT specifies the IPv4 or IPv6 socket address of the 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 when processed as input. The field is set to 0 when processed as output.

FAMILY
A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary 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 82 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 when the application programming interface (API) processing module cannot be loaded.

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

FCNTL

The blocking mode for a socket can be queried or set to nonblocking using the FNDELAY flag. You can query or set the FNDELAY flag even though it is not defined in your program.

See “IOCTL” on page 365 for another way to control socket blocking.

```
►►──EZASMI──TYPE=FCNTL──,S──=──┬─number───┬──,COMMAND──=──┬─'F_GETFL'─┬──────────►
                               ├─address──┤                ├─'F_SETFL'─┤
                               ├─*indaddr─┤                ├─address───┤
                               └─(reg)────┘                ├─*indaddr──┤
                                                           └─(reg)─────┘

►─,REQARG──=──┬─address──┬──,ERRNO──=──┬─address──┬──,RETCODE──=──┬─address──┬────►
              ├─*indaddr─┤             ├─*indaddr─┤               ├─*indaddr─┤
              └─(reg)────┘             └─(reg)────┘               └─(reg)────┘

►──┬────────────────────────┬──┬──────────────────────────┬─────────────────────►
   └─,ECB=──┬─address──┬────┘  └─,ERROR──=──┬─address──┬──┘
            ├─*indaddr─┤                    ├─*indaddr─┤
            └─(reg)────┘                    └─(reg)────┘

►──┬───────────────────────────┬─────────────────────────────────────────────────►◄
   └─,TASK──=──┬─address──┬────┘
               ├─*indaddr─┤
               └─(reg)────┘
```

Keyword **Description**

S
Input parameter. A value or the address of a halfword binary number specifying the socket descriptor for the socket that you want to unblock or query.

COMMAND
Input parameter. A fullword binary field or a literal that sets the FNDELAY flag to one of the following values:

Value **Description**

3 or 'F_GETFL'
Query the blocking mode for the socket.

4 or 'F_SETFL'
Set the mode to nonblocking for the socket. REQARG is set by TCP/IP.

The FNDELAY flag sets the nonblocking mode for the socket. If data is not present on calls that can block (READ, READV, and RECV), the call returns a -1, and ERRNO is set to 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 82 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 when the application programming interface (API) processing module cannot be loaded.

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

FREEADDRINFO

The FREEADDRINFO macro frees all the address information structures returned by GETADDRINFO in the RES parameter.

Important Note

This function call is not available with TCP/IP for VSE/ESA.

```
►►──EZASMI──TYPE=FREEADDRINFO──,ADDRINFO──=──┬─address──┬────────────────────────►
                                              ├─*indaddr─┤
                                              └─(reg)────┘

►─,ERRNO──=──┬─address───┬──,RETCODE──=──┬─address──┬────────────────────────────►
             ├─*indaddr──┤               ├─*indaddr─┤
             └─(reg)─────┘               └─(reg)────┘

►─┬──────────────────────────┬───────────────────────────────────────────────────►◄
  └─,ERROR──=──┬─address───┬─┘
               ├─*indaddr──┤
               └─(reg)─────┘
```

Keyword　**Description**

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 82 for information about ERRNO return codes.

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

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

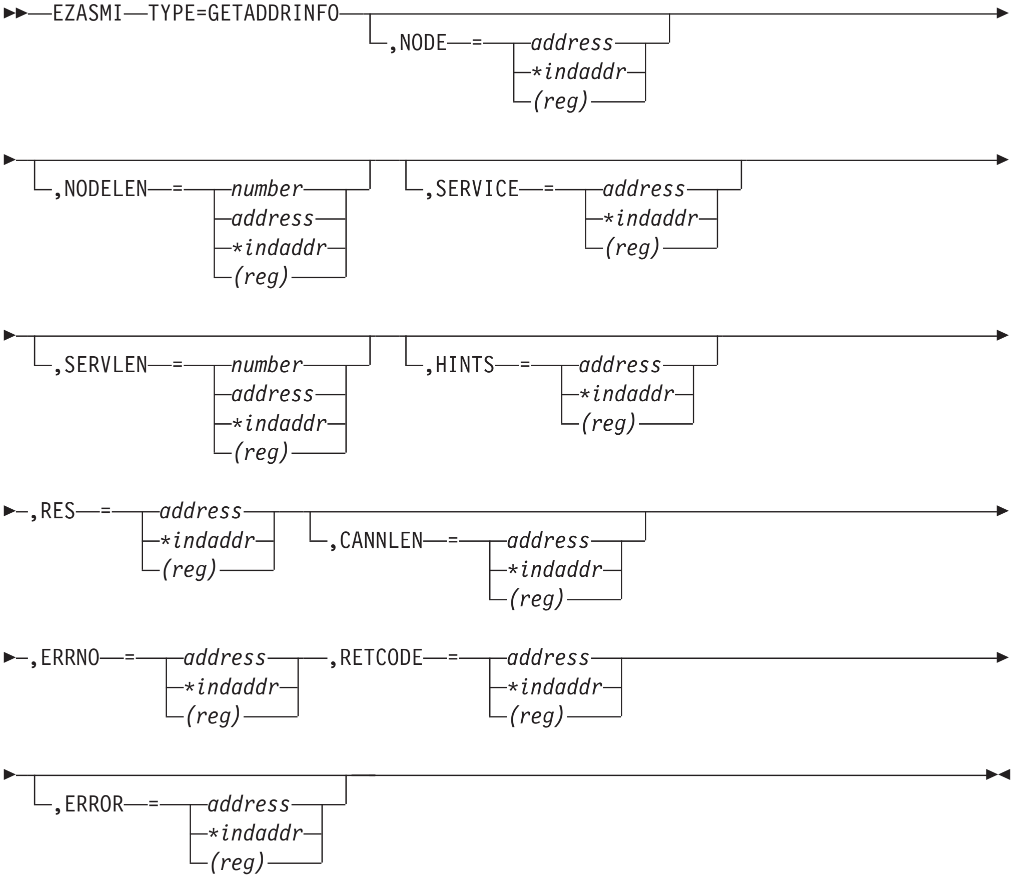
ERROR
Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

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 Note

This function call is not available with TCP/IP for VSE/ESA.



Keyword **Description**

NODE
An input parameter. Storage up to 255 bytes long that contains the host name being queried. If the AI_NUMERICHOST flag is specified in the storage pointed to by the HINTS operand, 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:

Value Description

FLAGS

A fullword binary field. Must have the value of 0 of the bitwise, OR of one or more of the following:

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

Specifies how to fill in the NAME pointed to by the returned RES.

If this flag is specified, 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 when IPv6 is supported, the caller will accept IPv4-mapped IPv6 addresses. When the AI-ALL flag is not also specified, if no IPv6 addresses are found, a query is made for IPv4 addresses. If IPv4 addresses are found, they are returned as IPv4-mapped IPv6 addresses.

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

AI-ALL (X'00000020') or a decimal value of 32.
When the AF field has a value of AF_INET6 and AI-ALL is set, the AI-V4MAPPED flag must also be set to indicate that the caller will accept all addresses (IPv6 and IPv4-mapped IPv6 addresses). When the AF field has a value of AF_UNSPEC when the system supports IPv6 and AI-ALL is set, the caller accepts IPv6 addreses 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-ADDRCONFG 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 when 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

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:

Value **Description**

FLAGS
A fullword binary field that is not used as output.

AF A fullword binary field. The value returned in this field can be used as the AF= argument on the TYPE=SOCKET macro to create a socket suitable for use with the returned address NAME.

SOCTYPE
A fullword binary field. The value returned in this field can be used as the SOCTYPE= argument on the TYPE=SOCKET macro to create a socket suitable for use with the returned address NAME.

PROTO
A fullword binary field. The value returned in this field can be used as the PROTO= argument on the TYPE=SOCKET macro to create a socket suitable for use with the returned address NAME.

NAMELEN
A fullword binary field. 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 CANONNAME field in the first returned address information structure contains the address of storage containing the canonical name corresponding to the input NODE argument. If the canonical name is not available, the CANONNAME field refers to the NODE argument or a string with the same contents. The CANNLEN field contains the length of the returned canonical name.

NAME
A fullword binary field. 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 CANONNAME field. This is an optional field.

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

See "ERRNO Values" on page 82 for information about ERRNO return codes.

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

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

ERROR Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

GETCLIENTID

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

> **Important Note**
>
> IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply when this program is used.

When GETCLIENTID is called by a server or client, the identifier of the calling application is returned.

```
►►──EZASMI──TYPE=GETCLIENTID──,CLIENT──=──┬─address──┬─────────────────────────────────────►
                                          ├─*indaddr─┤
                                          └─(reg)────┘

►─,ERRNO──=──┬─address──┬──,RETCODE──=──┬─address──┬──┬───────────────────────┬──────────►
             ├─*indaddr─┤               ├─*indaddr─┤  └─,ECB=─┬─address──┬────┘
             └─(reg)────┘               └─(reg)────┘          ├─*indaddr─┤
                                                              └─(reg)────┘

►─┬─────────────────────────┬─┬────────────────────────┬──────────────────────────────►◄
  └─,ERROR──=──┬─address──┬─┘ └─,TASK──=──┬─address──┬─┘
               ├─*indaddr─┤               ├─*indaddr─┤
               └─(reg)────┘               └─(reg)────┘
```

Keyword **Description**

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 82 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 when the application programming interface (API) processing module cannot be loaded.

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

GETHOSTBYADDR

The GETHOSTBYADDR macro returns domain and alias names of the host whose internet address is specified by the macro. A TCP/IP host can have multiple alias names and host internet addresses.

```
►►──EZASMI──TYPE=GETHOSTBYADDR──,HOSTADR──=──┬─number───┬──────────────────────►
                                             ├─address──┤
                                             ├─*indaddr─┤
                                             └─(reg)────┘

►─,HOSTENT──=──┬─address────┬──,RETCODE──=──┬─address──┬───────────────────────►
               ├─*indaddr─┤                 ├─*indaddr─┤
               └─(reg)────┘                 └─(reg)────┘

►──┬──────────────────────────┬──┬──────────────────────────┬──────────────────►◄
   └─,ERROR──=──┬─address──┬──┘  └─,TASK──=──┬─address──┬───┘
                ├─*indaddr─┤                 ├─*indaddr─┤
                └─(reg)────┘                 └─(reg)────┘
```

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.

Keyword **Description**

HOSTADR Input parameter. A fullword unsigned binary field set to the internet address of the host whose name you want to find.

HOSTENT Input parameter. A fullword word containing the address of the HOSTENT structure returned by the macro. For information about the HOSTENT structure, see Figure 78 on page 329.

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 when the application programming interface (API) processing module cannot be loaded.

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

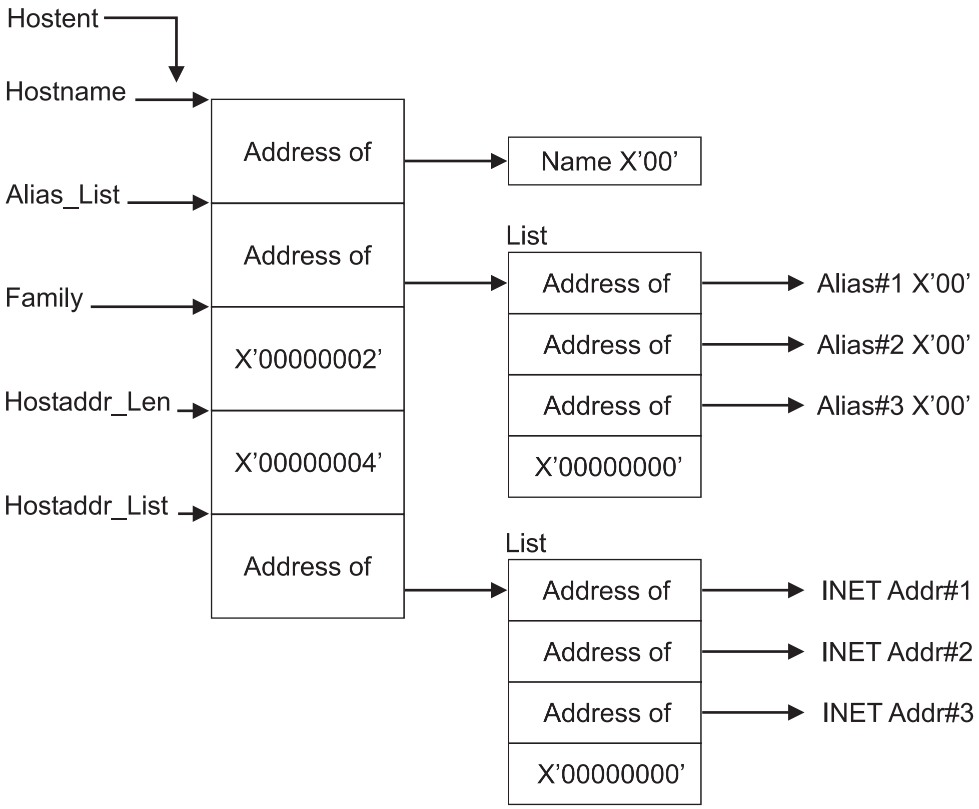


Figure 78. HOSTENT Structure Returned by the GETHOSTBYADDR Macro

GETHOSTBYADDR returns the HOSTENT structure shown in Figure 78. 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.

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 DNS (set with SET DNSx). TCP/IP for VSE/ESA will try each DNS, beginning with DNS1, until an 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.

```
►►──EZASMI──TYPE=GETHOSTBYNAME──,NAMELEN──=──┬─number───┬──────────────────────────►
                                              ├─address──┤
                                              ├─*indaddr─┤
                                              └─(reg)────┘

►─,NAME──=──┬─address───┬──,HOSTENT──=──┬─address───┬──,RETCODE──=──┬─address───┬────►
            ├─*indaddr──┤               ├─*indaddr──┤                ├─*indaddr──┤
            └─(reg)─────┘               └─(reg)─────┘                └─(reg)─────┘

►──┬───────────────────────────┬──┬──────────────────────────┬──────────────────►◄
   └─,ERROR──=──┬─address───┬──┘  └─,TASK──=──┬─address───┬──┘
                ├─*indaddr──┤                 ├─*indaddr──┤
                └─(reg)─────┘                 └─(reg)─────┘
```

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.

Keyword **Description**

NAMELEN
Input parameter. A value or the address of a fullword binary field specifying the length of the name and alias fields. This length has a maximum value of 255 bytes.

NAME
A character string, up to 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 79 on page 331.

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

Value	Description
0	Successful call
–1	An error occurred

ERROR
Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

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

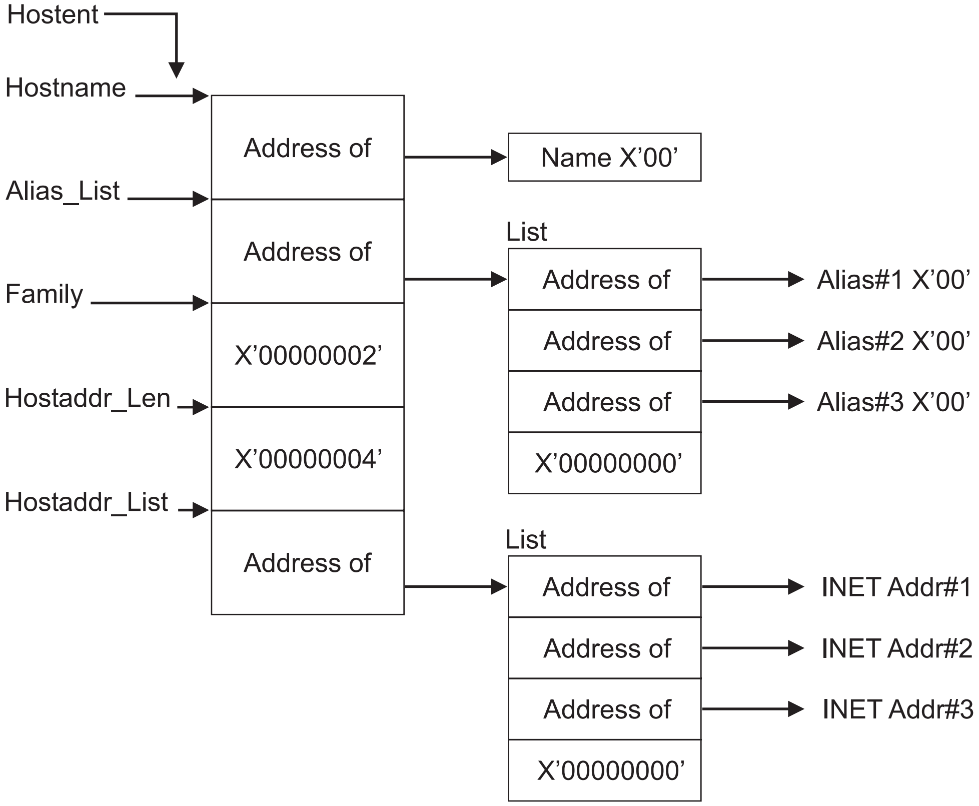


Figure 79. HOSTENT Structure Returned by the GETHOSTBYNAME Macro

GETHOSTBYNAME returns the HOSTENT structure shown in Figure 79. 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.

```
►►──EZASMI──TYPE=GETHOSTID──,RETCODE──=──┬─address──┬──┬───────────────────────┬──────────►
                                         ├─*indaddr─┤  └─,ECB=─┬─address──┬───┘
                                         └─(reg)────┘          ├─*indaddr─┤
                                                               └─(reg)────┘

►──┬──────────────────────────┬──┬─────────────────────────┬─────────────────────────────►◄
   └─,ERROR──=──┬─address──┬──┘  └─,TASK──=──┬─address──┬──┘
                ├─*indaddr─┤                 ├─*indaddr─┤
                └─(reg)────┘                 └─(reg)────┘
```

Keyword
Description

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 when the application programming interface (API) processing module cannot be loaded.

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

GETHOSTNAME

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

```
►►──EZASMI──TYPE=GETHOSTNAME──,NAMELEN──=──┬─address──┬──────────────────────────────►
                                           ├─*indaddr─┤
                                           └─(reg)────┘

►─,NAME──=──┬─address──┬─,ERRNO──=──┬─address──┬─,RETCODE──=──┬─address──┬──────────►
            ├─*indaddr─┤            ├─*indaddr─┤              ├─*indaddr─┤
            └─(reg)────┘            └─(reg)────┘              └─(reg)────┘

►──┬──────────────────────┬──┬──────────────────────────┬────────────────────────────►
   └─,ECB=──┬─address──┬──┘  └─,ERROR──=──┬─address──┬──┘
            ├─*indaddr─┤                  ├─*indaddr─┤
            └─(reg)────┘                  └─(reg)────┘

►──┬─────────────────────────┬───────────────────────────────────────────────────────►◄
   └─,TASK──=──┬─address──┬──┘
               ├─*indaddr─┤
               └─(reg)────┘
```

Keyword **Description**

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 82 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 when the application programming interface (API) processing module cannot be loaded.

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

GETNAMEINFO

The GETNAMEINFO macro returns the node name and service location of a socket address that is specified in the macro. On successful completion, GETNAMEINFO returns the node and service named, if requested, in the buffers provided.

Important Note
This function call is not available with TCP/IP for VSE/ESA.

```
►►──EZASMI──TYPE=GETNAMEINFO──,NAME──=──┬─address──┬─────────────────────────────────►
                                        ├─*indaddr─┤
                                        └─(reg)────┘

►──,NAMELEN──=──┬─number───┬──┬──────────────────────────┬───────────────────────────►
                ├─address──┤  └─,HOST──=──┬─address──┬───┘
                ├─*indaddr─┤              ├─*indaddr─┤
                └─(reg)────┘              └─(reg)────┘

►──┬──────────────────────────────┬──┬────────────────────────────────┬──────────────►
   └─,HOSTLEN──=──┬─number───┬────┘  └─,SERVICE──=──┬─address──┬──────┘
                  ├─address──┤                      ├─*indaddr─┤
                  ├─*indaddr─┤                      └─(reg)────┘
                  └─(reg)────┘

►──┬──────────────────────────────┬──┬──────────────────────────────────────┬────────►
   └─,SERVLEN──=──┬─number───┬────┘  └─,FLAGS──=──┬─'NI_DGRAM'──────────┬──┘
                  ├─address──┤                    ├─'NI_NAMEREQD'───────┤
                  ├─*indaddr─┤                    ├─'NI_NOFQDN'─────────┤
                  └─(reg)────┘                    ├─'NI_NUMERICHOST'────┤
                                                  ├─'NI_NUMERICSCOPE'───┤
                                                  ├─'NI_NUMERICSERV'────┤
                                                  ├─number──────────────┤
                                                  ├─address─────────────┤
                                                  ├─*indaddr────────────┤
                                                  └─(reg)───────────────┘

►──,ERRNO──=──┬─address──┬──,RETCODE──=──┬─address──┬────────────────────────────────►
              ├─*indaddr─┤               ├─*indaddr─┤
              └─(reg)────┘               └─(reg)────┘

►──┬────────────────────────────┬────────────────────────────────────────────────────►◄
   └─,ERROR──=──┬─address──┬────┘
                ├─*indaddr─┤
                └─(reg)────┘
```

Keyword **Description**

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 when processed as input. The field is set to 0 when processed as output.

FAMILY
A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary number that specifies the port number.

FLOW-INFO
This field is ignored by the TYPE=GETNAMEINFO macro.

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

SCOPE-ID
A fullword binary field that specifies the scope for an IPv6 address as an interface index. The resolver ignores the SCOPE_ID field, unless the address in IPv6-ADDRESS is a link-local address and the HOST parameter 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 SERVLEN parameters

Otherwise, an error occurs. The HOST name being queried consists of up to HOSTLEN or up to the first binary 0.

If the IPv6-ADDRESS value is a link-local address, and the SCOPE_ID interface index is nonzero, scope information is appended to the resolved host name using the format *host%scope information*. The scope information can be the numeric form of the SCOPE_ID interface index or the interface name that is associated with the SCOPE_ID interface index. Use the NI_NUMERICSCOPE option to select which form is returned. The combined host name and scope information is 255 bytes or less. For more information about scope information and TYPE=GETNAMEINFO processing.

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 SERVLEN 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 SERVLEN parameter. The SERVICE 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 SERVICE and SERVLEN parameters

Otherwise, an error occurs.

SERVLEN Initially an input parameter. A fullword binary field that contains the length of the SERVICE storage used to contain the returned resolved service name. If SERVLEN is 0 on input, 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 SERVICE parameter. One or both of the following groups of parameters are required:

- The HOST and HOSTLEN parameters
- The SERVICE 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.

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 82 for information about ERRNO return codes.

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

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

ERROR Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

GETPEERNAME

The GETPEERNAME macro returns the name of the remote socket to which the local socket is connected.

Important Note

IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply when this program is used.

```
►►──EZASMI──TYPE=GETPEERNAME──,S──=──┬─number───┬──,NAME──=──┬─address──┬──────────►
                                     ├─address──┤             ├─*indaddr─┤
                                     ├─*indaddr─┤             └─(reg)────┘
                                     └─(reg)────┘

►─,ERRNO──=──┬─address──┬──,RETCODE──=──┬─address──┬──┬─────────────────────┬──────►
             ├─*indaddr─┤               ├─*indaddr─┤  └─,ECB=──┬─address──┬─┘
             └─(reg)────┘               └─(reg)────┘           ├─*indaddr─┤
                                                               └─(reg)────┘

►──┬─────────────────────────┬──┬────────────────────────┬─────────────────────►◄
   └─,ERROR──=──┬─address──┬─┘  └─,TASK──=──┬─address──┬─┘
                ├─*indaddr─┤                ├─*indaddr─┤
                └─(reg)────┘                └─(reg)────┘
```

Keyword **Description**

S
A value, or the address of a halfword binary number specifying the local socket connected to the remote peer whose address is required.

NAME
Initially points to the peer name structure. It is filled when the call completes with the IPv4 or IPv6 address structure for the remote socket connected to the local socket, specified by S. Include the 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 when processed as input. The field is set to 0 when processed as output.

FAMILY
A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary 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 82 for information about ERRNO return codes.

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 when the application programming interface (API) processing module cannot be loaded.

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

GETSOCKNAME

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

Stream sockets are not assigned a name until after a successful call to BIND, CONNECT, or ACCEPT.

Use the GETSOCKNAME macro to determine the port assigned to a socket after that socket has been implicitly bound to a port. If an application calls CONNECT without previously calling BIND, the CONNECT macro completes the binding necessary by assigning a port to the socket. You can determine the port assigned to the socket by issuing GETSOCKNAME.

Important Note

IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply when this program is used.

```
►►──EZASMI──TYPE=GETSOCKNAME──,S──=──┬─number───┬──,NAME──=──┬─address───┬──────────►
                                     ├─address──┤            ├─*indaddr─┤
                                     ├─*indaddr─┤            └─(reg)────┘
                                     └─(reg)────┘

►─,ERRNO──=──┬─address──┬──,RETCODE──=──┬─address──┬──┬─────────────────────┬────►
             ├─*indaddr─┤               ├─*indaddr─┤  └─,ECB=──┬─address──┬─┘
             └─(reg)────┘               └─(reg)────┘           ├─*indaddr─┤
                                                               └─(reg)────┘

►─┬───────────────────────────┬──┬──────────────────────────┬────────────────────►◄
  └─,ERROR──=──┬─address──┬───┘  └─,TASK──=──┬─address──┬───┘
               ├─*indaddr─┤                  ├─*indaddr─┤
               └─(reg)────┘                  └─(reg)────┘
```

Keyword **Description**

S
: Input parameter. A value, or the address of a halfword binary number specifying the socket descriptor.

NAME
: Initially, the application provides a pointer to the IPv4 or IPv6 socket address structure, which is filled in on completion of the call with the socket name. Include the 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 address structure. Any value specified by the use of this field is ignored when processed as input. The field is set to 0 when processed as output.

FAMILY
A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary 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 Output parameter. A fullword binary field. If RETCODE is negative, this field contains an error number. See “ERRNO Values” on page 82 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 when the application programming interface (API) processing module cannot be loaded.

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

GETSOCKOPT

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

The options for each socket are described by the following parameters. You must specify the option that you want when you issue the GETSOCKOPT macro.

```
►►──EZASMI──TYPE=GETSOCKOPT──,S──=──┬─number───┬──,OPTNAME──=──┬─'SO_LINGER'─┬──────►
                                    ├─address──┤               ├─address─────┤
                                    ├─*indaddr─┤               ├─*indaddr────┤
                                    └─(reg)────┘               └─(reg)───────┘

►─,OPTVAL──=──┬─address──┬──,OPTLEN──=──┬─address──┬──,ERRNO──=──┬─address──┬──────►
              ├─*indaddr─┤              ├─*indaddr─┤             ├─*indaddr─┤
              └─(reg)────┘              └─(reg)────┘             └─(reg)────┘

►─,RETCODE──=──┬─address──┬──┬────────────────────────┬────────────────────────────►
               ├─*indaddr─┤  └─,ECB=──┬─address──┬────┘
               └─(reg)────┘           ├─*indaddr─┤
                                      └─(reg)────┘

►──┬───────────────────────────┬──┬──────────────────────────┬──────────────────────►◄
   └─,ERROR──=──┬─address──┬───┘  └─,TASK──=──┬─address──┬───┘
                ├─*indaddr─┤                  ├─*indaddr─┤
                └─(reg)────┘                  └─(reg)────┘
```

Keyword **Description**

S
Input parameter. A value, or the address of a halfword binary number specifying the socket descriptor of the socket requiring options.

OPTNAME
Input parameter. Set OPTNAME to one of the following options before you issue GETSOCKOPT.

SO_LINGER
Requests the status of SO_LINGER.

- When 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 276.

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 82 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 when 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 Note**
>
> IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply when this program is used.

```
►►──EZASMI──TYPE=GIVESOCKET──,S──=──┬─number───┬──,CLIENT──=──┬─address──┬──────────►
                                    ├─address──┤              ├─*indaddr─┤
                                    ├─*indaddr─┤              └─(reg)────┘
                                    └─(reg)────┘

►─,ERRNO──=──┬─address──┬──,RETCODE──=──┬─address──┬──┬──────────────────────┬──────►
             ├─*indaddr─┤               ├─*indaddr─┤  └─,ECB=──┬─address──┬──┘
             └─(reg)────┘               └─(reg)────┘           ├─*indaddr─┤
                                                               └─(reg)────┘

►──┬───────────────────────────┬──┬──────────────────────────┬─────────────────────►◄
   └─,ERROR──=──┬─address──┬───┘  └─,TASK──=──┬─address──┬───┘
                ├─*indaddr─┤                  ├─*indaddr─┤
                └─(reg)────┘                  └─(reg)────┘
```

Keyword	Description
S	Input parameter. A value, or the address of a halfword binary number specifying the descriptor of the socket to be given.
CLIENT	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 82 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 when 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.

EZASMI macro

```
►►──EZASMI──TYPE=GSKFREEMEM──,──AREA=──┬─address──┬─,RETCODE=──┬─address──┬─,──────►
                                       ├─*indaddr─┤             ├─*indaddr─┤
                                       └─(reg)────┘             └─(reg)────┘

►─ERRNO=──┬─address──┬──────────────────────────────────────────────────────────►◄
          ├─*indaddr─┤
          └─(reg)────┘
```

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 call 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.

```
►►──EZASMI──TYPE=GSKGETCIPHINF──,──CIPHLEVEL=──┬─number───┬─,──────────────────────►
                                               ├─address──┤
                                               ├─*indaddr─┤
                                               └─(reg)────┘

►─SECLEVEL=──┬─address──┬─,RETCODE=──┬─address──┬─,──ERRNO=──┬─address──┬──────────►◄
             ├─*indaddr─┤            ├─*indaddr─┤            ├─*indaddr─┤
             └─(reg)────┘            └─(reg)────┘            └─(reg)────┘
```

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)

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.

```
►►──EZASMI──TYPE=GSKGETDNBYLAB──,─────────────────────────────────────►

►─KEYLABEL=──┬─address──┬─,RETCODE=──┬─address──┬─,──ERRNO=──┬─address──┬──────────►◄
             ├─*indaddr─┤            ├─*indaddr─┤            ├─*indaddr─┤
             └─(reg)────┘            └─(reg)────┘            └─(reg)────┘
```

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 call.

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.

```
►►──EZASMI──TYPE=GSKINIT──,─────────────────────────────────────────────────►

►─SECTYPE=──┬─address──┬─,─┬──────────────────────────────┬─────────────────►
            ├─*indaddr─┤   └─KEYRING=──┬─address──┬─,─┘
            └─(reg)────┘               ├─*indaddr─┤
                                       └─(reg)────┘

►─┬────────────────────────────────┬────────────────────────────────────────►
  └─V3TIMEOUT=──┬─number───┬─,─┘
                ├─address──┤
                ├─*indaddr─┤
                └─(reg)────┘

►─┬─────────────────────────────────────────────────────┬─RETCODE=──┬─address──┬─,──►
  └─CAROOTS=──┬─number───┬─,AUTHTYPE=──┬─number───┬─,─┘            ├─*indaddr─┤
              ├─address──┤             ├─address──┤                 └─(reg)────┘
              ├─*indaddr─┤             ├─*indaddr─┤
              └─(reg)────┘             └─(reg)────┘

►─ERRNO=──┬─address──┬──────────────────────────────────────────────────────►◄
          ├─*indaddr─┤
          └─(reg)────┘
```

Keyword **Descriptions**

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.

CAROOTS
(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 when the CAROOTS field is set to 1. It is ignored when CAROOTS 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.

```
►►──EZASMI──TYPE=GSKSSOCCLOSE──,──────────────────────────────────────────────►

►─SSOCDATA=─┬─address──┬─,RETCODE=─┬─address──┬─,──ERRNO=─┬─address──┬───────────►◄
            ├─*indaddr─┤            ├─*indaddr─┤             ├─*indaddr─┤
            └─(reg)────┘            └─(reg)────┘             └─(reg)────┘
```

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.

```
►►──EZASMI──TYPE=GSKSSOCINIT──,──S=──┬─number───┬─,HANDSHAKE=──┬─number───┬─,──────────►
                                     ├─address──┤               ├─address──┤
                                     ├─*indaddr─┤               ├─*indaddr─┤
                                     └─(reg)────┘               └─(reg)────┘

►─┬──────────────────────────┬─SECTYPE=──┬─address──┬─,──────────────────────────────►
  └─DNAME=──┬─address──┬─,──┘            ├─*indaddr─┤
            ├─*indaddr─┤                 └─(reg)────┘
            └─(reg)────┘

►─┬─────────────────────────────┬─V3CIPHSEL=──┬─address──┬─,─────────────────────────►
  └─V3CIPHER=──┬─address──┬─,──┘              ├─*indaddr─┤
               ├─*indaddr─┤                   └─(reg)────┘
               └─(reg)────┘

►─┬─────────────────────────────┬─REASCODE=──┬─address──┬─,──────────────────────────►
  └─CERTINFO=──┬─address──┬─,──┘             ├─*indaddr─┤
               ├─*indaddr─┤                  └─(reg)────┘
               └─(reg)────┘

►─┬───────────────────────────┬─┬───────────────────────────┬────────────────────────►
  └─SKREAD=──┬─address──┬─,──┘ └─SKWRITE=──┬─address──┬─,──┘
             ├─*indaddr─┤                  ├─*indaddr─┤
             └─(reg)────┘                  └─(reg)────┘

►─RETCODE=──┬─address──┬─,ERRNO=──┬─address──┬───────────────────────────────────────►◄
            ├─*indaddr─┤          ├─*indaddr─┤
            └─(reg)────┘          └─(reg)────┘
```

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 when 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 TIECLEN 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(TIECLEN),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                     c
               S=T9RSOCK+2,       for this socket descriptor       c
               BUF=*T9RBUFA,      to this buffer                   c
               NBYTE=T9RBUFL,     with this length                 c
               TASK=T9SKRTIE,     own task storage                 c
               ERRNO=T9RERRNO,    own ERRNO                        c
               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
T9SKRSAV 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
T9RRETCD 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 TIECLEN 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. When 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.

```
►►──EZASMI──TYPE=GSKSSOCREAD──,──SSOCDATA=──┬─address──┬─,BUF=──┬─address──┬─,──────────►
                                            ├─*indaddr─┤        ├─*indaddr─┤
                                            └─(reg)────┘        └─(reg)────┘

►─NBYTE=──┬─number───┬─,RETCODE=──┬─address──┬─,──ERRNO=──┬─address──┬──────────────►◄
          ├─address──┤            ├─*indaddr─┤            ├─*indaddr─┤
          ├─*indaddr─┤            └─(reg)────┘            └─(reg)────┘
          └─(reg)────┘
```

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.

```
►►──EZASMI──TYPE=GSKSSOCRESET──,──────────────────────────────────────────────►

►─SSOCDATA=──┬─address──┬─,RETCODE=──┬─address──┬─,──ERRNO=──┬─address──┬──────────►◄
             ├─*indaddr─┤            ├─*indaddr─┤             ├─*indaddr─┤
             └─(reg)────┘            └─(reg)────┘             └─(reg)────┘
```

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.

```
►►──EZASMI──TYPE=GSKSSOCWRITE──,──SSOCDATA=──┬─address──┬─,BUF=──┬─address──┬─,────────►
                                             ├─*indaddr─┤        ├─*indaddr─┤
                                             └─(reg)────┘        └─(reg)────┘

►─NBYTE=──┬─number───┬─,RETCODE=──┬─address──┬─,──ERRNO=──┬─address──┬──────────────►◄
          ├─address──┤            ├─*indaddr─┤            ├─*indaddr─┤
          ├─*indaddr─┤            └─(reg)────┘            └─(reg)────┘
          └─(reg)────┘
```

SSOCDATA Input parameter. Address of GSKSOCDATA 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.

```
►►──EZASMI──TYPE=GSKUNINIT──,──RETCODE=─┬─address──┬─,ERRNO=─┬─address──┬──────────►◄
                                        ├─*indaddr─┤         ├─*indaddr─┤
                                        └─(reg)────┘         └─(reg)────┘
```

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, will generate a default INITAPI call:

- GETCLIENTID
- GETHOSTID
- GETHOSTNAME
- SELECT
- SELECTEX
- SOCKET
- TAKESOCKET

```
►►──EZASMI──TYPE=INITAPI──┬──────────────────────────────┬──────────────────►
                          └─,MAXSOC──=──┬─number───┬─────┘
                                        ├─address──┤
                                        ├─*indaddr─┤
                                        └─(reg)────┘

►──┬───────────────────────────────┬──┬─────────────────────────────┬───────►
   └─,SUBTASK──=──┬─address──┬─────┘  └─,IDENT──=──┬─address──┬─────┘
                  ├─*indaddr─┤                     ├─*indaddr─┤
                  └─(reg)────┘                     └─(reg)────┘

►─,MAXSNO──=──┬─address──┬──,ERRNO──=──┬─address──┬──,RETCODE──=──┬─address──┬──►
              ├─*indaddr─┤             ├─*indaddr─┤               ├─*indaddr─┤
              └─(reg)────┘             └─(reg)────┘               └─(reg)────┘

►──┬───────────────────────────────┬──┬──────────────────────────────┬──────►
   └─,APITYPE──=──┬─'2'──────┬─────┘  └─,UEEXIT──=──┬─address──┬─────┘
                  ├─address──┤                      ├─*indaddr─┤
                  ├─*indaddr─┤                      └─(reg)────┘
                  └─(reg)────┘

►──┬──────────────────────────┬──┬──────────────────────────┬───────────────►
   └─,ASYNC──=──┬─'NO'──┬─────┘  └─,ERROR──=──┬─indaddr─┬───┘
                └─'ECB'─┘                     └─(reg)───┘

►──┬────────────────────────────┬───────────────────────────────────────────►◄
   └─,TASK──=──┬─address──┬─────┘
               ├─*indaddr─┤
               └─(reg)────┘
```

Keyword **Descriptions**

MAXSOC Optional 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.

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 jobname 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)

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 may 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. Please note that this specification can be overwritten with the following JCL statement: // SETPARM EZA$PHA='routine-name'.

MAXSNO Output parameter. A fullword binary field containing 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 82, 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 specifying the APITYPE:

Value **Meaning**

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 will be 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 when the application programming interface (API) processing module cannot be loaded.

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

IOCTL

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

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

Note: IOCTL can only be used with programming languages that support address pointers

```
►►──EZASMI──TYPE=IOCTL──,S──=──┬─number───┬──,COMMAND──=──┬─'FIONBIO'─┬──────────►
                               ├─address──┤               ├─address───┤
                               ├─*indaddr─┤               ├─*indaddr──┤
                               └─(reg)────┘               └─(reg)─────┘

►─,REQARG──=──┬─address──┬──,RETARG──=──┬─address──┬──,ERRNO──=──┬─address──┬────►
              ├─*indaddr─┤              ├─*indaddr─┤             ├─*indaddr─┤
              └─(reg)────┘              └─(reg)────┘             └─(reg)────┘

►─,RETCODE──=──┬─address──┬──┬──────────────────────┬────────────────────────────►
               ├─*indaddr─┤  └─,ECB=──┬─address──┬──┘
               └─(reg)────┘           ├─*indaddr─┤
                                      └─(reg)────┘

►──┬────────────────────────────┬──┬───────────────────────────┬─────────────────►◄
   └─,ERROR──=──┬─address──┬────┘  └─,TASK──=──┬─address──┬────┘
                ├─*indaddr─┤                   ├─*indaddr─┤
                └─(reg)────┘                   └─(reg)────┘
```

Keyword
Description

S
Input parameter. A value, or the address of a halfword binary number specifying the socket to be controlled.

COMMAND
Input parameter. To control an operating characteristic, set this field to one of the following symbolic names. A value in a bit mask is associated with each symbolic name. By specifying one of these names, you are turning on a bit in a mask which communicates the requested operating characteristic to TCP/IP.

Value	Description
'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 7.

Table 7. 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 82 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 when the application programming interface (API) processing module cannot be loaded.

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

LISTEN

Only servers use the LISTEN macro. The LISTEN macro:

- 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 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.

```
►►──EZASMI──TYPE=LISTEN──,S──=──┬─number───┬──,BACKLOG──=──┬─'number'─┬──────────►
                                ├─address──┤               ├─address──┤
                                ├─*indaddr─┤               ├─*indaddr─┤
                                └─(reg)────┘               └─(reg)────┘

►─,ERRNO──=──┬─address──┬──,RETCODE──=──┬─address──┬──┬──────────────────────┬────►
             ├─*indaddr─┤               ├─*indaddr─┤  └─,ECB=──┬─address──┬──┘
             └─(reg)────┘               └─(reg)────┘           ├─*indaddr─┤
                                                               └─(reg)────┘

►──┬─────────────────────────────┬──┬────────────────────────────┬──────────────►◄
   └─,ERROR──=──┬─address──┬─────┘  └─,TASK──=──┬─address──┬─────┘
                ├─*indaddr─┤                    ├─*indaddr─┤
                └─(reg)────┘                    └─(reg)────┘
```

Keyword **Description**

S
Input parameter. A value, or the address of a halfword binary number specifying the socket descriptor.

BACKLOG
Input parameter. A value (enclosed in single quotation marks) or the address of a fullword binary number specifying the number of messages that can be backlogged. 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 82, 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 when the application programming interface (API) processing module cannot be loaded.

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

NTOP

The NTOP macro converts an IP address from its numeric binary form into a standard text presentation form. On successful completion, NTOP returns the converted IP address in the buffer provided.

> **Important Note**
>
> This function call is not available with TCP/IP for VSE/ESA.

```
►►──EZASMI──TYPE=NTOP──,AF──=──┬─'INET'───┬──,SRCADDR──=──┬─address──┬──────────►
                               ├─'INET6'──┤               ├─*indaddr─┤
                               ├─address──┤               └─(reg)────┘
                               ├─*indaddr─┤
                               └─(reg)────┘

►─,DSTADDR──=──┬─address──┬──,DSTLEN──=──┬─address──┬──,ERRNO──=──┬─address──┬──────►
               ├─*indaddr─┤              ├─*indaddr─┤             ├─*indaddr─┤
               └─(reg)────┘              └─(reg)────┘             └─(reg)────┘

►─,RETCODE──=──┬─address──┬──┬──────────────────────────┬──────────────────►◄
               ├─*indaddr─┤  └─,ERROR──=──┬─address──┬──┘
               └─(reg)────┘               ├─*indaddr─┤
                                          └─(reg)────┘
```

AF
Input parameter. Specify one of the following:

Value Description

'INET' or a decimal ′2′
Indicates the address being converted is an IPv4 address.

′INET6′ or a decimal ′19′
Indicates the address being converted is an IPv6 address.

AF can also indicate a fullword binary number specifying the address family.

SRCADDR
Input parameter. A field containing the numeric binary form of the IPv4 or IPv6 address being converted. For an IPv4 address, this field must be a fullword. For an IPv6 address, this field must be 16 bytes. The address must be in network byte order.

DSTADDR
Input parameter. A field used to receive the standard text presentation form of the IPv4 or IPv6 address being converted. For IPv4 the address 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 82, for information about ERRNO return codes.

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

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

ERROR
Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

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 Note

This function call is not available with TCP/IP for VSE/ESA.

```
►►──EZASMI──TYPE=PTON──,AF──=──┬─'INET'───┬──,SRCADDR──=──┬─address──┬───────────►
                               ├─'INET6'──┤               ├─*indaddr─┤
                               ├─address──┤               └─(reg)────┘
                               ├─*indaddr─┤
                               └─(reg)────┘

►─,SRCLEN──=──┬─address──┬──,DSTADDR──=──┬─address──┬──,ERRNO──=──┬─address──┬────►
              ├─*indaddr─┤               ├─*indaddr─┤             ├─*indaddr─┤
              └─(reg)────┘               └─(reg)────┘             └─(reg)────┘

►─,RETCODE──=──┬─address──┬──┬─────────────────────────────┬──────────────────────►◄
               ├─*indaddr─┤  └─,ERROR──=──┬─address──┬─────┘
               └─(reg)────┘               ├─*indaddr─┤
                                          └─(reg)────┘
```

AF
Input parameter. Specify one of the following:

Value **Description**

’INET’ or a decimal ’2’
Indicates the address being converted is an IPv4 address.

’INET6’ or a decimal ’19’
Indicates the address being converted is an IPv6 address.

AF can also indicate a fullword binary number specifying the address family.

SRCADDR
Input parameter. A field containing the standard text presentation form of the IPv4 or IPv6 address being converted. For IPv4 the address must be in dotted-decimal format and for IPv6 the address must be in colon-hex format. The size of the field for an IPv4 address must be 15 bytes and the size for an IPv6 address must be 45 bytes.

SRCLEN
Input parameter. A binary halfword field that must contain the length of the IP address to be converted.

DSTADDR
A field 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 82, for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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

ERROR

Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

READ

The READ macro reads data on a socket and stores it in a buffer. The READ macro applies only to connected sockets.

For datagram sockets, the READ call returns the entire datagram that was sent. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard extra bytes.

```
►►──EZASMI──TYPE=READ──,S──=──┬─number───┬──,NBYTE──=──┬─number───┬──────────────►
                              ├─address──┤              ├─address──┤
                              ├─*indaddr─┤              ├─*indaddr─┤
                              └─(reg)────┘              └─(reg)────┘

►──,BUF──=──┬─address───┬──,ERRNO──=──┬─address───┬──,RETCODE──=──┬─address───┬───►
            ├─*indaddr──┤             ├─*indaddr──┤               ├─*indaddr──┤
            └─(reg)─────┘             └─(reg)─────┘               └─(reg)─────┘

►──┬─────────────────────────┬──┬──────────────────────────┬────────────────────►
   └─,ECB=──┬─address───┬────┘  └─,ERROR──=──┬─address───┬─┘
            ├─*indaddr──┤                    ├─*indaddr──┤
            └─(reg)─────┘                    └─(reg)─────┘

►──┬──────────────────────────┬─────────────────────────────────────────────────►◄
   └─,TASK──=──┬─address───┬──┘
               ├─*indaddr──┤
               └─(reg)─────┘
```

Keyword **Description**

S
Input parameter. A value, or the address of a halfword binary number specifying the socket that is going to read the data.

NBYTE
Input parameter. A fullword binary number set to the size of BUF. READ does not return more than the number of bytes of data in NBYTE even if more data is available.

BUF
On input, a buffer to be filled by completion of the call. The length of BUF must be at least as long as the value of NBYTE.

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 82 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 when the application programming interface (API) processing module cannot be loaded.

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

READ returns up to the number of bytes specified by NBYTE. If less than the number of bytes requested is available, the READ macro returns the number currently available.

If data is not available for the socket and the socket is in blocking mode, the READ macro blocks the caller until data arrives. If data is not available, and the socket is in nonblocking mode, READ returns a -1 and sets ERRNO EWOULDBLOCK. See "IOCTL" on page 365 or "FCNTL" on page 316 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.

```
►►──EZASMI──TYPE=READV──,S──=──┬─number───┬──,IOV──=──┬─address──┬──────────────►
                               ├─address──┤           ├─*indaddr─┤
                               ├─*indaddr─┤           └─(reg)────┘
                               └─(reg)────┘

►─,IOVCNT──=──┬─address──┬─,RETCODE──=──┬─address──┬─,ERRNO──=──┬─address──┬────►
              ├─*indaddr─┤              ├─*indaddr─┤            ├─*indaddr─┤
              └─(reg)────┘              └─(reg)────┘            └─(reg)────┘

                                                           (1)
►──┬───────────────────────────┬──┬──────────────────────────┬──────────────────►◄
   └─,ERROR──=──┬─address──┬───┘  └─,TASK──=──┬─address──┬───┘
                ├─*indaddr─┤                  ├─*indaddr─┤
                └─(reg)────┘                  └─(reg)────┘
```

Notes:

1 The ECB parameter for asynchronous processing is not supported with this call; unlike z/OS.

Keyword **Description**

S
Input parameter. A value, or the address of a halfword binary number specifying the socket that is going to read the data.

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 82 for information about ERRNO return codes.

ERROR Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

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

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.

```
►►──EZASMI──TYPE=RECV──,S──=──┬─number───┬──,NBYTE──=──┬─number───┬────────────────►
                              ├─address──┤              ├─address──┤
                              ├─*indaddr─┤              ├─*indaddr─┤
                              └─(reg)────┘              └─(reg)────┘

►─,BUF──=──┬─address───┬──,ERRNO──=──┬─address───┬──,RETCODE──=──┬─address───┬──────►
           ├─*indaddr──┤             ├─*indaddr──┤               ├─*indaddr──┤
           └─(reg)─────┘             └─(reg)─────┘               └─(reg)─────┘

►─┬──────────────────────┬─┬────────────────────────┬───────────────────────────────►
  └─,ECB=─┬─address──┬───┘ └─,ERROR──=──┬─address──┬─┘
          ├─*indaddr─┤                  ├─*indaddr─┤
          └─(reg)────┘                  └─(reg)────┘

►─┬───────────────────────────┬─────────────────────────────────────────────────────►◄
  └─,TASK──=──┬─address───┬───┘
              ├─*indaddr──┤
              └─(reg)─────┘
```

Keyword	Description
S	Input parameter. A value, or the address of a halfword binary number specifying the socket descriptor.
NBYTE	Input parameter. A fullword binary number set to the size of BUF. RECV does not receive more than the number of bytes of data in NBYTE even if more data is available.
BUF	On input, a buffer to be filled by completion of the call. The length of BUF must be at least as long as the value of NBYTE.
ERRNO	Ouput parameter. A fullword binary field. If RETCODE is negative, this field contains an error number. See "ERRNO Values" on page 82 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 when the application programming interface (API) processing module cannot be loaded.

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

If data is not available for the socket and the socket is in blocking mode, the RECV macro blocks the caller until data arrives. If data is not available and the socket is in nonblocking mode, RECV returns a −1 and sets ERRNO to EWOULDBLOCK. See “FCNTL” on page 316 or “IOCTL” on page 365 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 316 or "IOCTL" on page 365 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 Note

IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply when this program is used.

```
►►──EZASMI──TYPE=RECVFROM──,S──=──┬─number───┬─,NBYTE──=──┬─number───┬──────────►
                                  ├─address──┤             ├─address──┤
                                  ├─*indaddr─┤             ├─*indaddr─┤
                                  └─(reg)────┘             └─(reg)────┘

►─,BUF──=──┬─address──┬─,NAME──=──┬─address──┬─,ERRNO──=──┬─address──┬──────────►
           ├─*indaddr─┤           ├─*indaddr─┤            ├─*indaddr─┤
           └─(reg)────┘           └─(reg)────┘            └─(reg)────┘

►─,RETCODE──=──┬─address──┬──┬──────────────────────────┬───────────────────────►
               ├─*indaddr─┤  └─,ERROR──=──┬─address──┬──┘
               └─(reg)────┘               ├─*indaddr─┤
                                          └─(reg)────┘

►──┬──────────────────────────┬─────────────────────────────────────────────────►◄
   └─,TASK──=──┬─address──┬───┘
               ├─*indaddr─┤
               └─(reg)────┘
```

Keyword — **Description**

S
Input parameter. A value, or the address of a halfword binary number specifying the socket to receive the data.

NBYTE
Input parameter. A value, or the address of a fullword binary number specifying the length of the input buffer. NBYTE must first be initialized to the size of the buffer associated with NAME. On return the NBYTE contains the number of bytes of data received.

BUF
On input, a buffer to be filled by completion of the call. The length of BUF must be at least as long as the value of NBYTE.

NAME
Initially, the IPv4 or IPv6 application provides a pointer to a structure that will contain the peer socket name on completion of the call. If the NAME parameter value is nonzero, the IPv4 or IPv6 source address of the message is filled. 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 when processed as input. The field is set to 0 when processed as output.

FAMILY
A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary 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 82 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 when the application programming interface (API) processing module cannot be loaded.

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

SELECT

In a process where multiple I/O operations can occur it is necessary for the program to be able to wait on one or several of the operations to complete. For example, consider a program that issues a READ to multiple sockets whose blocking mode is set. Because the socket would block on a READ macro, only one socket could be read at a time. Setting the sockets to nonblocking would solve this problem, but would require polling each socket repeatedly until data becomes available. The SELECT macro allows you to test several sockets and to process a later I/O macro only when one of the tested sockets is ready. This ensures that the I/O macro does not block.

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

- Set the read, write, and except arrays to zeros
- Do not specify MAXSOC.

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 RSNDMSK to ‘1’ before issuing the SELECT macro. When the SELECT macro returns, the corresponding bits in the RRETMSK indicate sockets ready for reading.

Write Operations

A socket is selected for writing, ready to be written, when:

- TCP/IP can accept additional outgoing data.
- A connection request is received in response to an ACCEPT macro.
- A CONNECT call for a nonblocking socket, that has previously returned ERRNO EINPROGRESS, completes the connection.

The WRITE, SEND, or SENDTO macros block when the data to be sent exceeds the amount that TCP/IP can accept. To avoid this, you can precede the write operation with a SELECT macro to ensure that the socket is ready for writing.

To determine if a socket is ready for the write operation, set the appropriate bit in WSNDMSK to ‘1’.

Exception Operations

For each socket to be tested, the SELECT macro can check for an exception condition. The exception conditions are:

- The calling program (concurrent server) has issued a GIVESOCKET command and the target subtask has successfully issued the TAKESOCKET call. When this condition is selected, the calling program (concurrent server) should issue CLOSE to dissociate itself from the socket.
- A socket has received out-of-band data. For this condition, a READ macro returns the out-of-band data before the program data.

To determine if a socket has an exception condition, use the ESNDMSK character string and set the appropriate bits to ‘1’.

Returning the Results

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

```
►►──EZASMI──TYPE=SELECT──,MAXSOC──=──┬─address──┬──,ERRNO──=──┬─address──┬──────────►
                                     ├─*indaddr─┤              ├─*indaddr─┤
                                     └─(reg)────┘              └─(reg)────┘

►─,RETCODE──=──┬─address──┬──┬──────────────────────────────────┬──────────────────►
               ├─*indaddr─┤  └─,TIMEOUT──=──┬─address──┬────────┘
               └─(reg)────┘                 ├─*indaddr─┤
                                            └─(reg)────┘

►─┬─────────────────────────────────────────────────────────────┬──────────────────►
  └─,RSNDMSK──=──┬─address──┬──,RRETMSK──=──┬─address──┬────────┘
                 ├─*indaddr─┤               ├─*indaddr─┤
                 └─(reg)────┘               └─(reg)────┘

►─┬─────────────────────────────────────────────────────────────┬──────────────────►
  └─,WSNDMSK──=──┬─address──┬──,WRETMSK──=──┬─address──┬────────┘
                 ├─*indaddr─┤               ├─*indaddr─┤
                 └─(reg)────┘               └─(reg)────┘
```

```
►─┬──────────────────────────────────────────────────────────────────────┬──────────►
  └─,ESNDMSK──=──┬─address──┬─,ERETMSK──=──┬─address──┬─┘
                 ├─*indaddr─┤              ├─*indaddr─┤
                 └─(reg)────┘              └─(reg)────┘

►─┬────────────────────────┬─┬──────────────────────────┬──────────────────────────►
  └─,ECB=─┬─address──┬─┘ └─,ERROR──=──┬─address──┬─┘
          ├─*indaddr─┤                ├─*indaddr─┤
          └─(reg)────┘                └─(reg)────┘

►─┬───────────────────────────┬─────────────────────────────────────────────────►◄
  └─,TASK──=──┬─address──┬─┘
              ├─*indaddr─┤
              └─(reg)────┘
```

Keyword **Description**

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 82, 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.

- 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 when the application programming interface (API) processing module cannot be loaded.

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

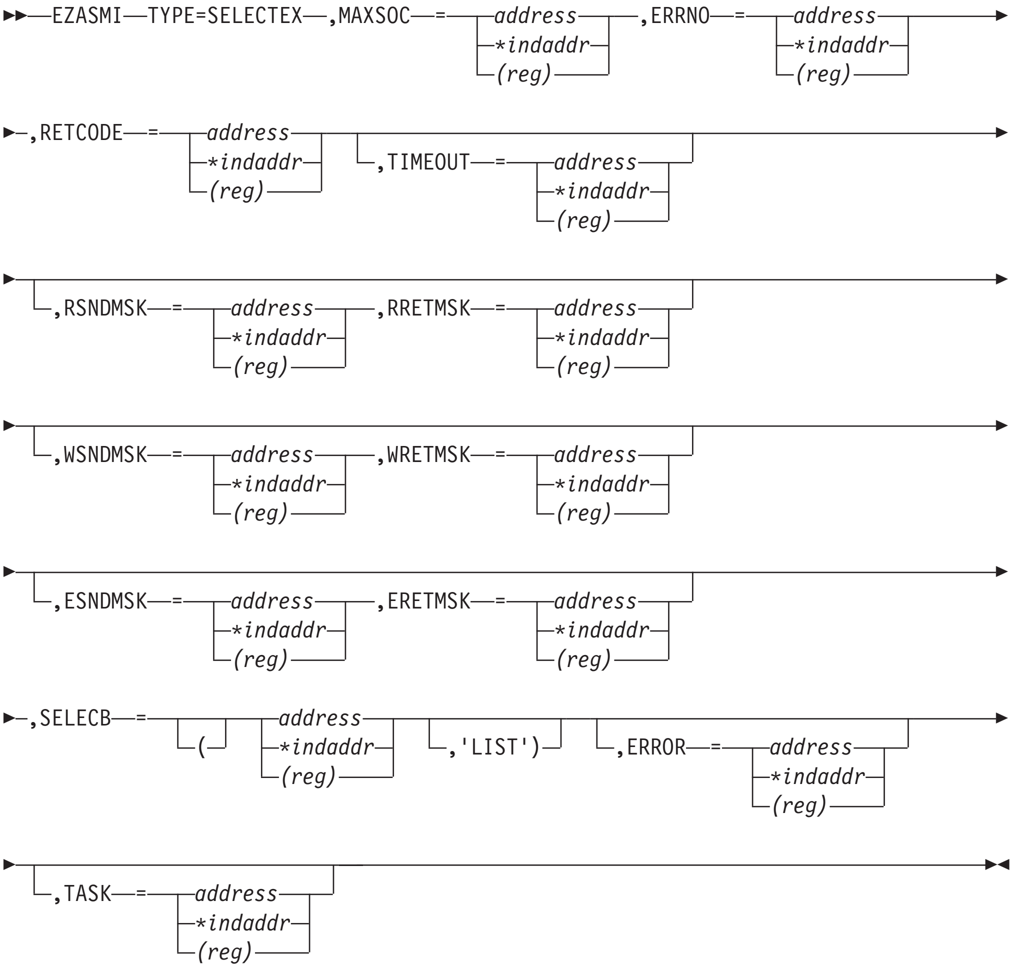
SELECTEX

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

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

- Set the read, write, and except arrays to zeros
- Do not specify MAXSOC.

For a detailed description on testing sockets, refer to the description of "SELECT" on page 382.



Keyword **Description**

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	Meaning
>0	The number of ready sockets.

0 Either the SELECTEX time limit has expired (ECB value will be 0) or one of the caller's ECBs has been posted (ECB value will be nonzero and the caller's descriptor sets will be set to 0). The caller must initialize the ECB values to 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.

RRETMSK Output parameter. The bit-mask array returned by the SELECT if RSNDMSK is specified. The length of this bit-mask array is dependent on the value in MAXSOC and must be a multiple of 4 bytes.

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 when the application programming interface (API) processing module cannot be loaded.

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

SEND

The SEND macro sends datagrams on a specified connected socket.

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.

```
►►──EZASMI──TYPE=SEND──,S──=──┬─number───┬──,NBYTE──=──┬─number───┬───────────────────►
                              ├─address──┤             ├─address──┤
                              ├─*indaddr─┤             ├─*indaddr─┤
                              └─(reg)────┘             └─(reg)────┘

►──,BUF──=──┬─address──┬──,ERRNO──=──┬─address──┬──,RETCODE──=──┬─address──┬───────────►
            ├─*indaddr─┤             ├─*indaddr─┤                ├─*indaddr─┤
            └─(reg)────┘             └─(reg)────┘                └─(reg)────┘

►──┬─────────────────────────┬──┬─────────────────────────────┬──────────────────────►
   └─,ECB=──┬─address──┬──┘  └─,ERROR──=──┬─address──┬──┘
            ├─*indaddr─┤                   ├─*indaddr─┤
            └─(reg)────┘                   └─(reg)────┘

►──┬───────────────────────────┬──────────────────────────────────────────────────────►◄
   └─,TASK──=──┬─address──┬──┘
               ├─*indaddr─┤
               └─(reg)────┘
```

Keyword **Description**

S Input parameter. A value, or the address of a halfword binary number specifying the socket descriptor of the socket that is sending data.

NBYTE Input parameter. A value, or the address of a fullword binary number specifying the number of bytes to transmit.

BUF The address of the data being transmitted. The length of BUF must be at least as long as the value of NBYTE.

ERRNO Output parameter. A fullword binary field. If RETCODE is negative, this field contains an error number. See “ERRNO Values” on page 82, 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 when the application programming interface (API) processing module cannot be loaded.

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

SENDTO

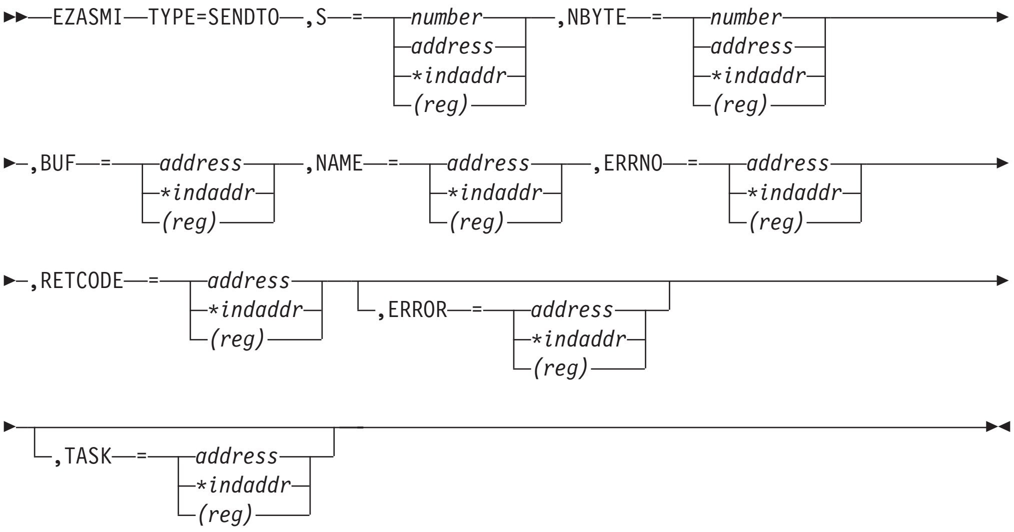
SENDTO is similar to SEND, except that it includes the destination address parameter. You can use the destination address on the SENDTO macro to send datagrams on a UDP socket that is connected or not connected.

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 Note

IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply when this program is used.



```
►►──EZASMI──TYPE=SENDTO──,S──=──┬─number───┬─,NBYTE──=──┬─number───┬────────────►
                                ├─address──┤             ├─address──┤
                                ├─*indaddr─┤             ├─*indaddr─┤
                                └─(reg)────┘             └─(reg)────┘

►─,BUF──=──┬─address──┬─,NAME──=──┬─address──┬─,ERRNO──=──┬─address──┬──────────►
           ├─*indaddr─┤           ├─*indaddr─┤            ├─*indaddr─┤
           └─(reg)────┘           └─(reg)────┘            └─(reg)────┘

►─,RETCODE──=──┬─address──┬──┬──────────────────────────┬──────────────────────►
               ├─*indaddr─┤  └─,ERROR──=──┬─address──┬──┘
               └─(reg)────┘               ├─*indaddr─┤
                                          └─(reg)────┘

►──┬─────────────────────────┬─────────────────────────────────────────────────►◄
   └─,TASK──=──┬─address──┬──┘
               ├─*indaddr─┤
               └─(reg)────┘
```

Keyword	Description
S	Output parameter. A value, or the address of a halfword binary number specifying the socket descriptor of the socket sending the data.
NBYTE	Input parameter. A value, or the address of a fullword binary number specifying the number of bytes to transmit.
BUF	Input parameter. The address of the data being transmitted. The length of BUF must be at least as long as the value of NBYTE.
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 when processed as input. The field is set to 0 when processed as output.

FAMILY
A 1-byte binary field that specifies the IPv6 addressing family. For TCP/IP the value is a decimal 19, indicating AF_INET6.

PORT A halfword binary 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 82 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 when the application programming interface (API) processing module cannot be loaded.

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

SETSOCKOPT

The SETSOCKOPT macro sets the options associated with a socket. SETSOCKOPT can be called only for sockets in the AF_INET domain.

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**.

```
►►──EZASMI──TYPE=SETSOCKOPT──,S──=──┬─number───┬──,OPTLEN──=──┬─address──┬──────────►
                                    ├─address──┤              ├─*indaddr─┤
                                    ├─*indaddr─┤              └─(reg)────┘
                                    └─(reg)────┘

►─,OPTNAME──=──┬─'SO_KEEPALIVE'─┬─,OPTVAL──=──┬─address──┬──────────────────────────►
               ├─'SO_LINGER'────┤             ├─*indaddr─┤
               ├─'SO_REUSEADDR'─┤             └─(reg)────┘
               ├─address────────┤
               ├─*indaddr───────┤
               └─(reg)──────────┘

►─,ERRNO──=──┬─address──┬─,RETCODE──=──┬─address──┬──┬─────────────────────┬──────►
             ├─*indaddr─┤              ├─*indaddr─┤  └─,ECB=─┬─address──┬──┘
             └─(reg)────┘              └─(reg)────┘          ├─*indaddr─┤
                                                             └─(reg)────┘

►──┬─────────────────────────────┬──┬───────────────────────────┬──────────────────►◄
   └─,ERROR──=──┬─address──┬─────┘  └─,TASK──=──┬─address──┬────┘
                ├─*indaddr─┤                    ├─*indaddr─┤
                └─(reg)────┘                    └─(reg)────┘
```

Keyword **Description**

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:

Value **Description**

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 processes data that has not been transmitted when a CLOSE macro is issued for the socket. This option has meaning only for stream sockets.

- When **SO_LINGER** is set and CLOSE is called, the calling program is blocked until the data is successfully transmitted or the connection has timed out.
- When 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 82, 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 when 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

One way to terminate a network connection is to issue a CLOSE macro that attempts to complete all outstanding data transmission requests prior to breaking the connection. The SHUTDOWN macro can be used to close one-way traffic while completing data transfer in the other direction. The HOW parameter determines the direction of the traffic to shutdown.

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

```
►►──EZASMI──TYPE=SHUTDOWN──,S──=──┬─number───┬──,HOW──=──┬─number───┬──────────────►
                                  ├─address──┤            ├─address──┤
                                  ├─*indaddr─┤            ├─*indaddr─┤
                                  └─(reg)────┘            └─(reg)────┘

►─,ERRNO──=──┬─address──┬──,RETCODE──=──┬─address──┬──┬──────────────────────┬──────►
             ├─*indaddr─┤               ├─*indaddr─┤  └─,ECB=──┬─address──┬──┘
             └─(reg)────┘               └─(reg)────┘           ├─*indaddr─┤
                                                               └─(reg)────┘

►──┬────────────────────────────┬──┬───────────────────────────┬────────────────►◄
   └─,ERROR──=──┬─address──┬────┘  └─,TASK──=──┬─address──┬────┘
                ├─*indaddr─┤                   ├─*indaddr─┤
                └─(reg)────┘                   └─(reg)────┘
```

Keyword
Description

S
Input parameter. A value, or the address of a halfword binary number specifying the socket to be shutdown.

HOW
Input parameter. A fullword binary field specifying the shutdown method.

Value	Description
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 82, 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 when the application programming interface (API) processing module cannot be loaded.

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

SOCKET

The SOCKET macro creates an endpoint for communication and returns a socket descriptor representing the endpoint. Different types of sockets provide different communication services.

Important Note

IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply when this program is used.

```
►►──EZASMI──TYPE=SOCKET──,AF──=──┬─'INET'───┬──,SOCTYPE──=──┬─'STREAM'───┬────────►
                                 ├─address──┤               ├─'DATAGRAM'─┤
                                 ├─*indaddr─┤               ├─address────┤
                                 └─(reg)────┘               ├─*indaddr───┤
                                                            └─(reg)──────┘

►─,ERRNO──=──┬─address──┬──,RETCODE──=──┬─address──┬─────────────────────────────►
             ├─*indaddr─┤               ├─*indaddr─┤
             └─(reg)────┘               └─(reg)────┘

►─┬──────────────────────────┬─┬─────────────────────────┬───────────────────────►
  └─,PROTO──=──┬─address──┬──┘ └─,ECB=──┬─address──┬──────┘
               ├─*indaddr─┤             ├─*indaddr─┤
               └─(reg)────┘             └─(reg)────┘

►─┬──────────────────────────┬─┬──────────────────────────┬──────────────────────►◄
  └─,ERROR──=──┬─address──┬──┘ └─,TASK──=──┬─address──┬───┘
               ├─*indaddr─┤                ├─*indaddr─┤
               └─(reg)────┘                └─(reg)────┘
```

Keyword **Description**

AF
Input parameter. Specifies the literal INET, which indicates the internet or TCP/IP. Specify one of the following:

Value **Description**

'INET' or a decimal '2'
Indicates the address being converted is an IPv4 address.

'INET6' or a decimal '19'
Indicates the address being converted is an IPv6 address.

AF can also indicate a fullword binary number specifying the address family.

SOCTYPE
Input parameter. A fullword binary field set to the type of socket required. The types are:

Value **Description**

1 or 'STREAM'
Stream sockets provide sequenced, two-way byte streams that are reliable and connection-oriented. They support a mechanism for out-of-band data. This is the normal type for TCP/IP.

2 or 'DATAGRAM'
Datagram sockets provide datagrams, which are

connectionless messages of a fixed maximum length whose reliability is not guaranteed. Datagrams can be corrupted, received out of order, lost, or delivered multiple times. This type is supported only in the AF_INET domain.

ERRNO Output parameter. A fullword binary field. If RETCODE is negative, this field contains an error number. See "ERRNO Values" on page 82 for information about ERRNO return codes.

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 when 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 198.

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.

Notes:

1. When TAKESOCKET is issued, a new socket descriptor is returned in RETCODE. You should use this new socket descriptor in later macros such as GETSOCKOPT, which require the S (socket descriptor) parameter.
2. Both concurrent servers and iterative servers use this interface. An iterative server handles one client at a time. A concurrent server receives connection requests from multiple clients and creates subtasks that process the client requests. When a subtask is created, the concurrent server gets a new socket, passes the new socket to the subtask, and dissociates itself from the connection. The CICS Listener program is an example of a concurrent server.

Important Note

IPv6 support is not available with TCP/IP for VSE/ESA. Any reference to IPv6 addresses or address structures do not apply when this program is used.

```
►►──EZASMI──TYPE=TAKESOCKET──,CLIENT──=──┬─address───┬──────────────────────────►
                                         ├─*indaddr──┤
                                         └─(reg)─────┘

►─,SOCRECV──=──┬─address───┬──,ERRNO──=──┬─address───┬───────────────────────►
               ├─*indaddr──┤             ├─*indaddr──┤
               └─(reg)─────┘             └─(reg)─────┘

►─,RETCODE──=──┬─address───┬──┬───────────────────────┬─────────────────────►
               ├─*indaddr──┤  └─,ECB=──┬─address───┬──┘
               └─(reg)─────┘           ├─*indaddr──┤
                                       └─(reg)─────┘

►──┬──────────────────────────────┬──┬────────────────────────────┬─────────►◄
   └─,ERROR──=──┬─address───┬─────┘  └─,TASK──=──┬─address───┬────┘
                ├─*indaddr──┤                    ├─*indaddr──┤
                └─(reg)─────┘                    └─(reg)─────┘
```

Keyword **Description**

CLIENT Input parameter. The client data returned by the GETCLIENTID macro.

Field **Description**

DOMAIN
Input parameter. A fullword binary number set to the domain of the program that is giving the socket. For TCP/IP the value is a decimal 2, indicating AF_INET, or a decimal 19, indicating AF_INET6.

Note: The TAKESOCKET can only acquire a socket of the same address family from a GIVESOCKET.

NAME
An 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 82, 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 when the application programming interface (API) processing module cannot be loaded.

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

TASK

The TASK macro allocates a task storage area addressable to all socket users within a task. If more than one module is using sockets within a task, it is your responsibility to provide the task storage address to each module. These modules should use the instruction EZASMI TYPE=TASK with STORAGE=DSECT to define the storage mapping.

If this macro is not named, the default name EZASMTIE is used for the storage mapping.

```
►►──EZASMI──TYPE=TASK──,STORAGE──=──┬─DSECT─┬──────────────────────────────────►◄
                                    └─CSECT─┘
```

Keyword **Description**

STORAGE Input parameter. Defines one of the following storage definitions:

Keyword **Description**

DSECT Generates a DSECT.

CSECT Generates an inline storage definition that can be used within a CSECT or as a part of a larger DSECT.

TERMAPI

The TERMAPI macro ends the session created by the INITAPI macro.

Note: The INITAPI and TERMAPI macros must be issued under the same task.

```
►►──EZASMI──TYPE=TERMAPI──┬──────────────────────────┬──────────────────────►◄
                          └─,TASK──=──┬─address──┬─┘
                                      ├─*indaddr─┤
                                      └─(reg)────┘
```

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

WRITE

The WRITE macro writes data on a connected socket. The WRITE macro is similar to the SEND macro.

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

For stream sockets, the data is processed as streams of information with no boundaries separating the data. For example, if you want to send 1000 bytes of data, each call to the write macro can send 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.

```
►►──EZASMI──TYPE=WRITE──,S──=──┬─number───┬──,NBYTE──=──┬─number───┬──────────────►
                               ├─address──┤             ├─address──┤
                               ├─*indaddr─┤             ├─*indaddr─┤
                               └─(reg)────┘             └─(reg)────┘

►─,BUF──=──┬─address──┬──,ERRNO──=──┬─address──┬──,RETCODE──=──┬─address──┬─────►
           ├─*indaddr─┤             ├─*indaddr─┤                ├─*indaddr─┤
           └─(reg)────┘             └─(reg)────┘                └─(reg)────┘

►──┬──────────────────────┬──┬──────────────────────────┬────────────────────────►
   └─,ECB=─┬─address──┬───┘  └─,ERROR──=──┬─address──┬──┘
           ├─*indaddr─┤                   ├─*indaddr─┤
           └─(reg)────┘                   └─(reg)────┘

►──┬──────────────────────────┬─────────────────────────────────────────────────►◄
   └─,TASK──=──┬─address──┬───┘
               ├─*indaddr─┤
               └─(reg)────┘
```

Keyword **Description**

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 82, 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 when the application programming interface (API) processing module cannot be loaded.

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

This macro writes up to NBYTE bytes of data. If there is not enough available buffer space for the socket data to be transmitted, and the socket is in blocking mode, WRITE blocks the caller until additional buffer space is available. If the socket is in nonblocking mode, WRITE returns a -1 and sets ERRNO to EWOULDBLOCK. See “FCNTL” on page 316 or “IOCTL” on page 365 for a description of how to set the nonblocking mode.

WRITEV

The WRITEV macro writes data on a socket from a set of buffers.

```
►►──EZASMI──TYPE=WRITEV──,S──=──┬─number───┬──,IOV──=──┬─address──┬──────────────────►
                                ├─address──┤           ├─*indaddr─┤
                                ├─*indaddr─┤           └─(reg)────┘
                                └─(reg)────┘

►─,IOVCNT──=──┬─address──┬──,RETCODE──=──┬─address──┬──,ERRNO──=──┬─address──┬──────►
              ├─*indaddr─┤               ├─*indaddr─┤             ├─*indaddr─┤
              └─(reg)────┘               └─(reg)────┘             └─(reg)────┘

                                                             (1)
►──┬──────────────────────────┬──┬─────────────────────────┬──────────────────────►◄
   └─,ERROR──=──┬─address──┬──┘  └─,TASK──=──┬─address──┬──┘
                ├─*indaddr─┤                 ├─*indaddr─┤
                └─(reg)────┘                 └─(reg)────┘
```

Notes:

1 The ECB parameter for asynchronous processing is not supported with this call; unlike z/OS.

Keyword **Description**

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
>=0	The number of bytes sent.
−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 ERRNO.

See “ERRNO Values” on page 82, for information about ERRNO return codes.

ERROR
Input parameter. The location in your program to receive control when the application programming interface (API) processing module cannot be loaded.

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

Part 3. CICS Listener Support

Chapter 12. Setting Up and Configuring CICS Listener Support

Overview

This chapter describes the steps required to configure the CICS Listener Support. The error messages are included in the *z/VSE Messages and Codes, Volume 1*.

Notes:

1. The CICS Listener Support requires CICS/TS. First of all, 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. Please refer to “CICS Considerations for the EZA Interfaces” on page 86 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 Dataset with EZACICD” on page 418
Use the configuration transaction to customize the Configuration dataset	“Customizing the Configuration Dataset” on page 421
Note: You can modify the dataset while CICS is running by using EZAC. See “Configuration Transaction (EZAC)” on page 421.	

CICS — Defining CICS Resources

The following definitions are required for the CICS Listener Support:

- Transactions
- Programs (see “Program Definitions” on page 414)
- Files (see “File Definitions” on page 415)
- Transient data queues (see “Transient Data Definition” on page 415)

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 which 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)* (SC33–0666)

Transaction Definitions

The following four transactions are required to support the CICS Listener:

EZAC Configure the socket interface

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.

Note: For transactions EZAL, EZAO, and EZAP we have suggested a priority of 255. 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.

Note that, 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.

Notes:

1. Use of the IBM-supplied Listener is not required.
2. You may use a transaction name other than EZAL.
3. The TASKDATALoc values for EZAO and EZAP and the TASKDATALoc value for EZAL must all be the same.

Program Definitions

The following programs and one mapset 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 CICS Listener Interface.

EZACIC21
is the initialization module for CICS Listener Interface.

EZACIC22
is the termination module for 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 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.

Note that, 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 may be changed.

If so, it must match the name specified in the ERRORTD parameter of the EZAC DEFINE CICS, the EZACICD TYPE=CICS, or both.(refer to “Configuration Macro” on page 418).

The Listener transaction can start a server using a transient data queue, as described in “Listener Input Format” on page 468. Following is an 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 80. 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.

```
DFHMCT TYPE=EMP,ID=(EZA02.01),CLASS=PERFORM,                          X
       PERFORM=ADDCNT(1,1)
DFHMCT TYPE=EMP,ID=(EZA02.02),CLASS=PERFORM,                          X
       PERFORM=ADDCNT(2,1)
DFHMCT TYPE=EMP,ID=(EZA02.03),CLASS=PERFORM,                          X
       PERFORM=ADDCNT(3,1)
DFHMCT TYPE=EMP,ID=(EZA02.04),CLASS=PERFORM,                          X
       PERFORM=ADDCNT(4,1)
DFHMCT TYPE=EMP,ID=(EZA02.05),CLASS=PERFORM,                          X
       PERFORM=ADDCNT(5,1)
DFHMCT TYPE=EMP,ID=(EZA02.06),CLASS=PERFORM,                          X
       PERFORM=ADDCNT(6,1)
DFHMCT TYPE=EMP,ID=(EZA02.07),CLASS=PERFORM,                          X
       PERFORM=ADDCNT(7,1)
DFHMCT TYPE=EMP,ID=(EZA02.08),CLASS=PERFORM,                          X
       PERFORM=ADDCNT(8,1)
DFHMCT TYPE=EMP,ID=(EZA02.09),CLASS=PERFORM,                          X
       PERFORM=ADDCNT(9,1)
DFHMCT TYPE=EMP,ID=(EZA02.10),CLASS=PERFORM,                          X
       PERFORM=ADDCNT(10,1)
DFHMCT TYPE=EMP,ID=(EZA02.11),CLASS=PERFORM,                          X
       PERFORM=ADDCNT(11,1)
DFHMCT TYPE=EMP,ID=(EZA02.12),CLASS=PERFORM,                          X
       PERFORM=(MLTCNT(1,11)),                                        X
       COUNT=(1,CONN,STARTED,INVALID,DISTRAN,DISPROG,GIVESOKT,SECEXIT)
```

Figure 81. The Monitor Control Table (MCT) for Listener

In the ID parameter, the following specifications are used:

(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 pre-allocated, but empty.
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 421.[1]

[1] The EZAC transaction is modeled after the CEDA transaction used by CICS Resource Definition Online (RDO).

Building the Configuration Dataset with EZACICD

Configuration Macro

The configuration macro (EZACICD) is used to build the configuration dataset. This dataset can then be incorporated into CICS using RDO and modified using the configuration transactions (see "Configuration Transaction (EZAC)" on page 421). The macro is keyword-driven with the TYPE keyword controlling the specific function request. The dataset contains one record for each instance of CICS it supports, and one record for each listener. The following is an example of the macros required to create a configuration file for one instance of the CICS Listener interface using one listener:

```
EZACICD TYPE=INITIAL,    INITIALIZE GENERATION ENVIRONMENT    X
      PRGNAME=EZACONFP,  GENERATE THIS PROGRAM                X
      FILNAME=EZACONF    name of the configuration file
EZACICD TYPE=CICS,       GENERATE CONFIGURATION RECORD        X
      APPLID=DBDCCICS,  APPLID OF CICS                        X
      CACHMIN=10,       MINIMUM REFRESH TIME FOR CACHE        X
      CACHMAX=20,       MAXIMUM REFRESH TIME FOR CACHE        X
      CACHRES=5,        MAXIMUM NUMBER OF ACTIVE RESOLVERS    X
      ERRORTD=EZAM      NAME OF TD QUEUE FOR ERROR MESSAGES
EZACICD TYPE=LISTENER,   CREATE LISTENER RECORD               X
      APPLID=DBDCCICS,  APPLID OF CICS                        X
      TRANID=EZAL,      USE STANDARD TRANSACTION ID           X
      PORT=3010,        USE PORT NUMBER 3010                  X
      BACKLOG=40,       SET BACKLOG VALUE TO 40               X
      ACCTIME=30,       SET TIMEOUT VALUE TO 30 SECONDS       X
      GIVTIME=10,       SET GIVESOCKET TIMEOUT TO 10 SECONDS  X
      REATIME=300,      SET READ TIMEOUT TO 5 MINUTES         X
      NUMSOCK=100,      SUPPORT 99 CONCURRENT CONNECTIONS     X
      MINMSGL=4,        MINIMUM INPUT MESSAGE IS 4 BYTES      X
      IMMED=NO,         DO NOT START LISTENER IMMEDIATELY     X
      FASTRD=NO         READ AFTER ACCEPT (NO SELECT)
EZACICD TYPE=FINAL
```

TYPE Parameter: The TYPE parameter controls the function requests. It may have the following values:

Value **Meaning**

INITIAL
Initialize the generation environment. This value should only be used once per generation and it should be in the first invocation of the macro. For sub-parameters, refer to "TYPE=INITIAL."

CICS Identify a CICS object. This corresponds to a specific instance of CICS and will create a configuration record. For sub-parameters, refer to "TYPE=CICS" on page 419.

LISTENER
Identify a Listener object. This will create a listener record. For sub-parameters, refer to "TYPE=LISTENER" on page 419.

FINAL
indicates the end of the generation. There are no sub-parameters.

TYPE=INITIAL: When TYPE=INITIAL is specified, the following parameters apply:

Value **Meaning**

PRGNAME
The name of the generated initialization program. The default value is EZACONFP.

FILNAME
The file name used for the Configuration File in the execution of the initialization program. The default value is EZACONF.

TYPE=CICS: When TYPE=CICS is specified, the following parameters apply:

Value Meaning

APPLID
The APPLID of the CICS address space in which this instance of CICS Listener is to run. This field is mandatory.

CACHMIN
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 internet address. Higher values improve performance but increase the risk of getting an incorrect (expired) address when resolving a name. The value must be less than cachmax. The default value is 15.

CACHMAX
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 internet 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.

CACHRES
The maximum number of concurrent resolvers desired. If the number of concurrent resolvers is equal to or greater than this value, refresh of cache records will not happen unless their age is greater than the CACHMAX value. The default value is 10.

ERRORTD
The name of a Transient Data destination to which error messages will be written. The default value is EZAM.

TYPE=LISTENER: When TYPE=LISTENER is specified the following parameters apply:

Value Meaning

APPLID
The APPLID value of the CICS object for which this listener is being defined. If this is omitted, the APPLID from the previous TYPE=CICS macro is used.

TRANID
The transaction name for this listener. The default is EZAL.

PORT The port number this listener will use for accepting connections. This parameter is mandatory. The value should be between 2049 and 65535. The ports may be shared.

BACKLOG
The number of unaccepted connections that can be queued to this listener. The default value is 20.

Note: Due to the TCP/IP Stack Implementation a value of 1 will always be used.

ACCTIME

The time in seconds this listener will wait for a connection request before checking for a CICS Listener interface shutdown or CICS shutdown. The default value is 60. Setting this value high will minimize CPU consumption on a lightly loaded system but will lengthen shutdown processing. Setting this value low will use more CPU but facilitate shutdown processing.

GIVTIME

The time in seconds this listener will wait for a response to a GIVESOCKET. If this time expires, the listener will assume that either the server transaction did not start or the TAKESOCKET failed. At this time, the listener will send the client a message indicating the server failed to start and close the socket (connection). If this parameter is not specified, the ACCTIME value is used.

REATIME

The time in seconds this listener will wait for a response to a READ request. If this time expires, the listener will assume that the client has failed and will terminate the connection by closing the socket. If this parameter is not specified, no checking for read timeout is done.

NUMSOCK

The number of sockets supported by this listener. One socket is the listening socket. The others are used to pass connections to the servers using the GIVESOCKET call so, in effect, one less than this number is the maximum number of concurrent GIVESOCKET requests that can be active. The default value is 50.

MINMSGL

The minimum length of the Transaction Initial Message from the client to the listener. The default value is 4. The listener will continue to read on the connection until this length of data has been received. FASTRD (below) handles blocking.

IMMED

Specify YES or NO. YES indicates this listener is to be started when the interface starts. No indicates this Listener is to be started independently using the EZAO transaction. The default is YES.

FASTRD

Specify YES or NO. YES indicates this listener will issue a READ immediately after completion of the ACCEPT, i.e. without issuing an intervening SELECT. NO indicates this listener will issue a SELECT between the ACCEPT and the READ. YES improves performance but relies on the client sending data immediately after its connect request. NO assumes the client may connect without sending data immediately.

The default is YES.

Note: FASTRD=YES acts as a blocking (synchronous) read. FASTRD=NO causes the system to ensure that data is present before issuing a read (that is, it does not block).

TRANTRN

Specify YES or NO. 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. With NO 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 if and only if the first character is not upper case EBCDIC.

TRANUSR

Specify YES or NO. NO indicates that the user data from the Transaction Initial Message should not be translated from ASCII to EBCDIC. YES indicates that the user data may be translated depending on TRANTRN and whether the transaction code is upper case EBCDIC. The default value 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 of multiple listeners on multiple ports is recommended.

Table 8 shows how the listener handles translation with different combinations of TRANTRN, TRANSUSR, and character format of the transaction code:

Table 8. 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

SECEXIT

The name of the security exit used by this listener. The default is no security exit.

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.

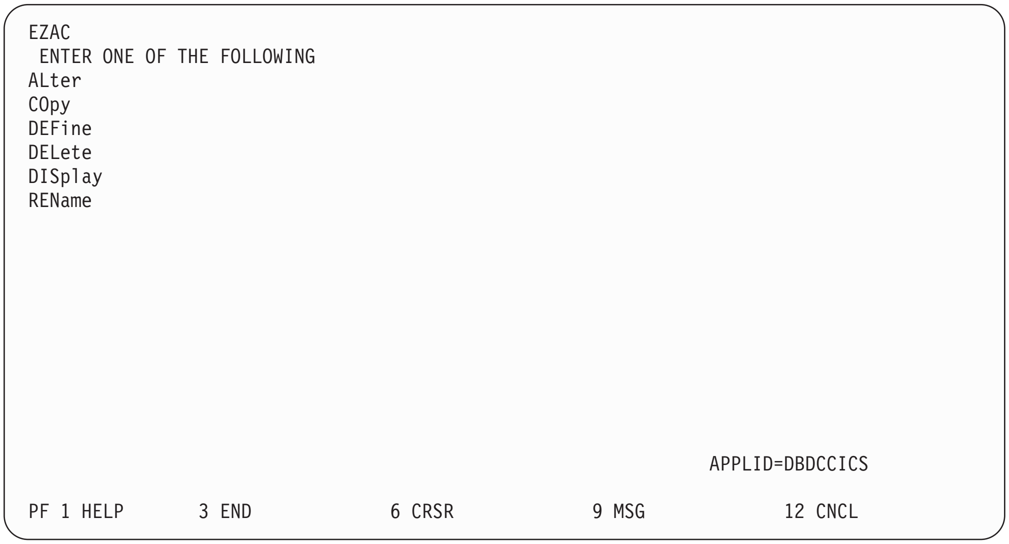
There is a Listener object for each Listener defined for a CICS. It is possible that a CICS may have no Listener but this is not common practice. A CICS may have multiple listeners which are either multiple instances of the supplied Listener with different specifications, multiple user-written listeners 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	CICS/Listener	Modifies the attributes of an existing resource definition
COPY	CICS/Listener	• 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.
DEFINE	CICS/Listener	Create a new resource definition
DELETE	CICS/Listener	• CICS - Deletes the CICS object and all of its associated listeners. • Listener - Deletes the Listener object.
DISPLAY	CICS/Listener	Shows the parameters specified for the CICS/Listener object.
RENAME	CICS/Listener	Performs a COPY followed by a DELETE of the original object.

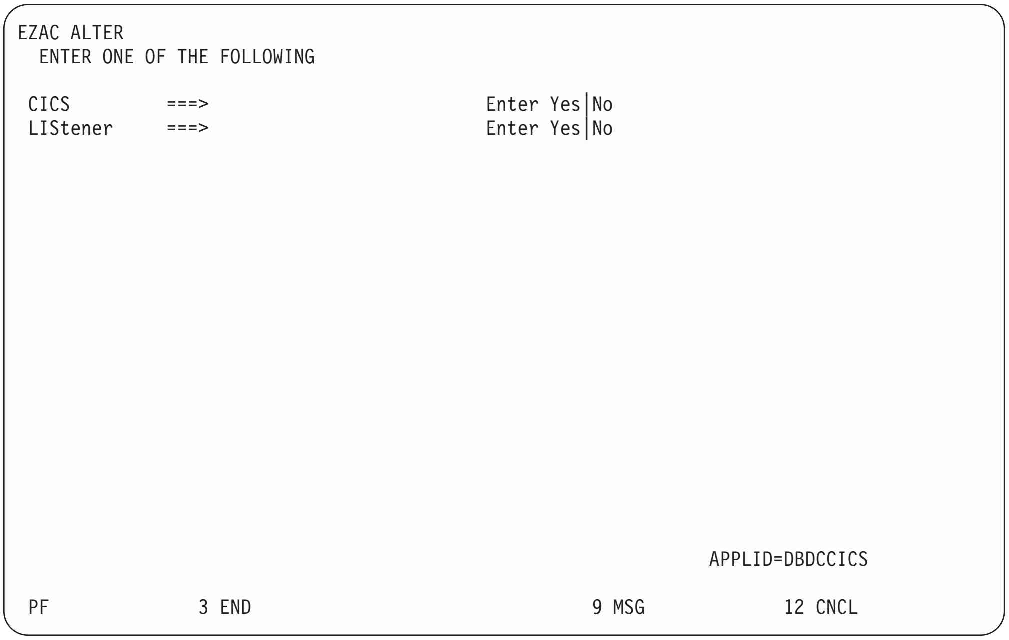
If you enter EZAC, the following screen is displayed:



```
EZAC
 ENTER ONE OF THE FOLLOWING
ALter
COpy
DEFine
DELete
DISplay
REName















                                                                APPLID=DBDCCICS

PF 1 HELP       3 END            6 CRSR            9 MSG            12 CNCL
```

Figure 82. EZAC Initial Screen

ALTER Function: The ALTER function is used to change CICS objects, their Listener objects, or both. If you specify ALter on the EZAC Initial Screen or enter EZAC AL on a blank screen, the following screen is displayed:

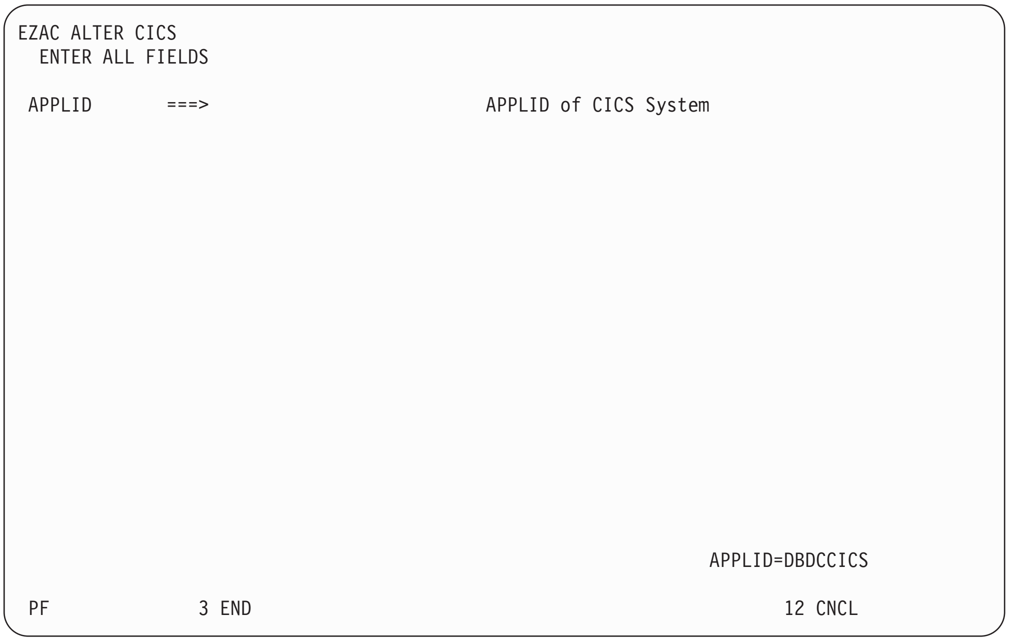


```
EZAC ALTER
  ENTER ONE OF THE FOLLOWING

 CICS         ===>                          Enter Yes|No
 LIStener     ===>                          Enter Yes|No




















                                                                 APPLID=DBDCCICS

 PF             3 END                                 9 MSG            12 CNCL
```

Figure 83. EZAC ALTER Screen

Note: You can short-cut this by entering either EZAC ALTER CICS or EZAC ALTER LISTENER.

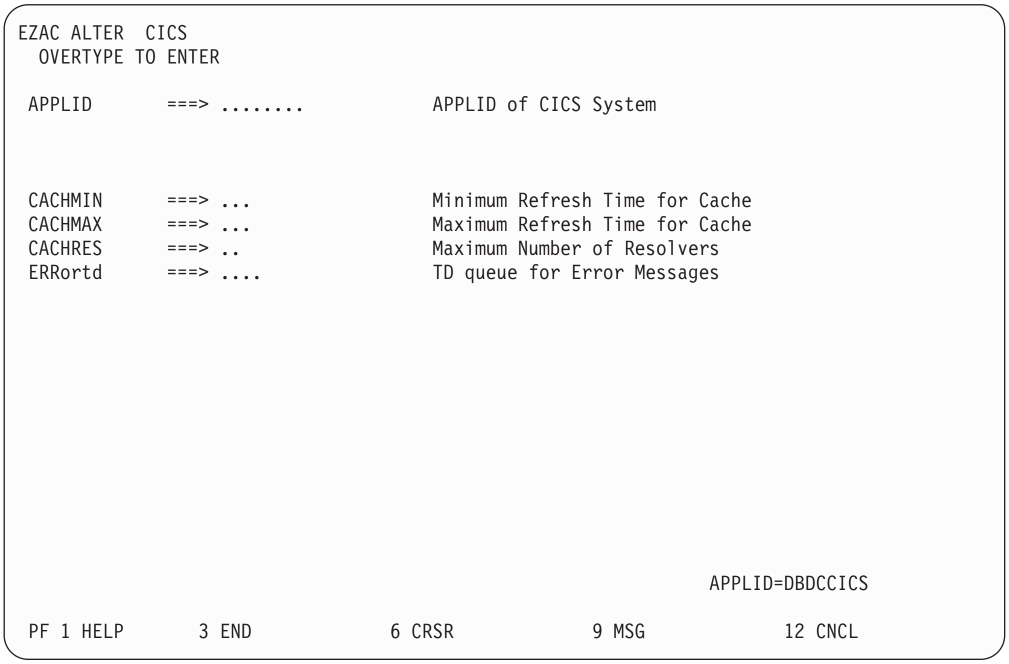
ALTER CICS: For alteration of a CICS object, the following screen is displayed:



```
EZAC ALTER CICS
  ENTER ALL FIELDS

 APPLID       ===>                          APPLID of CICS System





















                                                                 APPLID=DBDCCICS

 PF             3 END                                                  12 CNCL
```

Figure 84. EZAC ALTER CICS screen

After the APPLID is entered, the following screen is displayed.

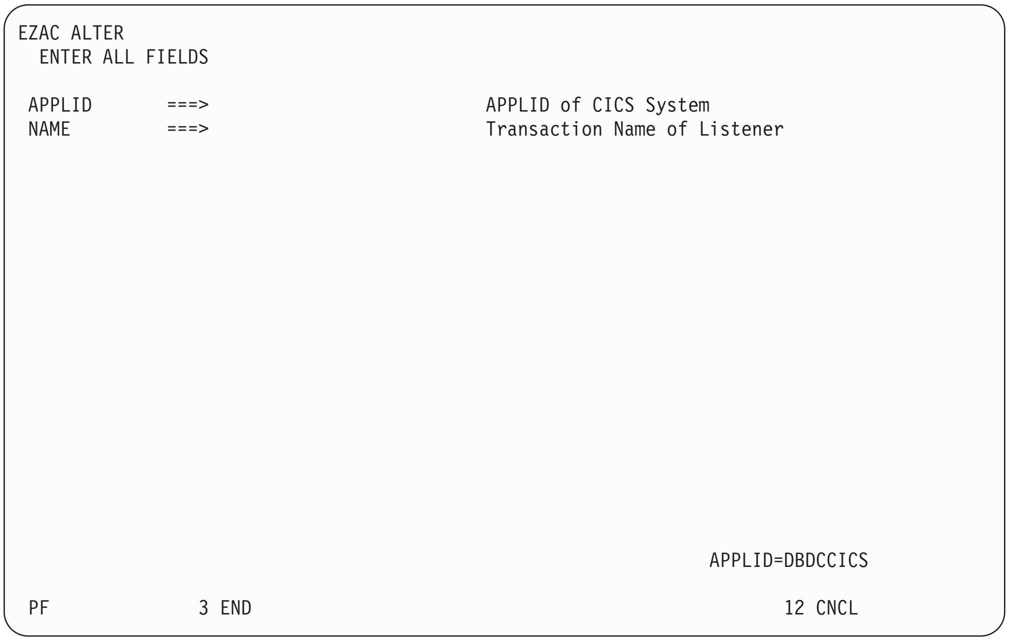


```
EZAC ALTER  CICS
  OVERTYPE TO ENTER

 APPLID      ===> ........           APPLID of CICS System



 CACHMIN     ===> ...                Minimum Refresh Time for Cache
 CACHMAX     ===> ...                Maximum Refresh Time for Cache
 CACHRES     ===> ..                 Maximum Number of Resolvers
 ERRortd     ===> ....               TD queue for Error Messages

















                                                            APPLID=DBDCCICS

 PF 1 HELP      3 END          6 CRSR          9 MSG          12 CNCL
```

Figure 85. EZAC ALTER CICS Detail Screen

The system will request a confirmation of the values displayed. After the changes are confirmed, the changed values will be in effect for the next initialization of the CICS sockets interface.

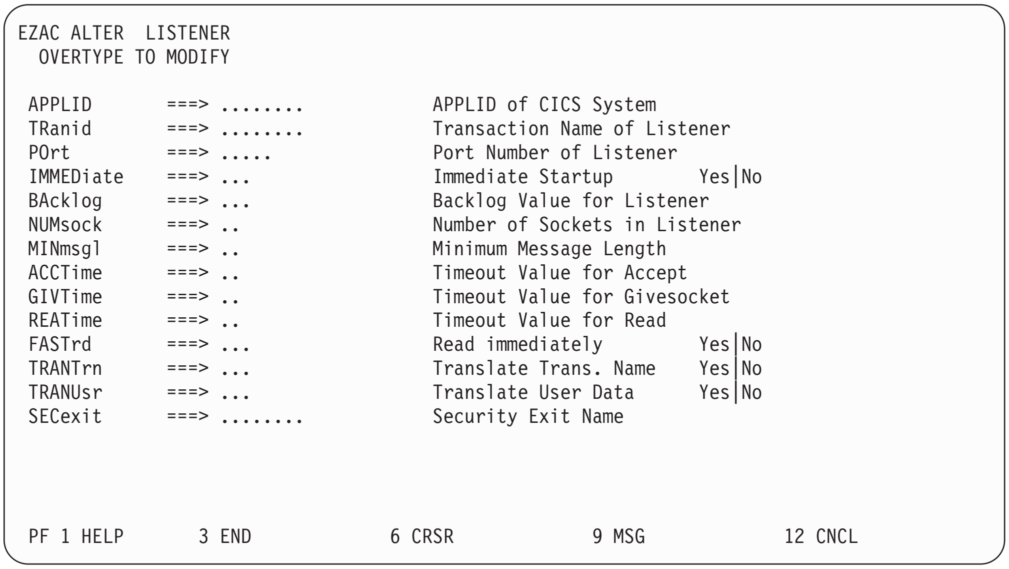
ALTER LISTENER: For alteration of a Listener, the following screen is displayed:



```
EZAC ALTER
  ENTER ALL FIELDS

 APPLID      ===>                         APPLID of CICS System
 NAME        ===>                         Transaction Name of Listener




















                                                            APPLID=DBDCCICS

 PF             3 END                                         12 CNCL
```

Figure 86. ALTER LISTENER screen

After the names are entered, the following screen is displayed:

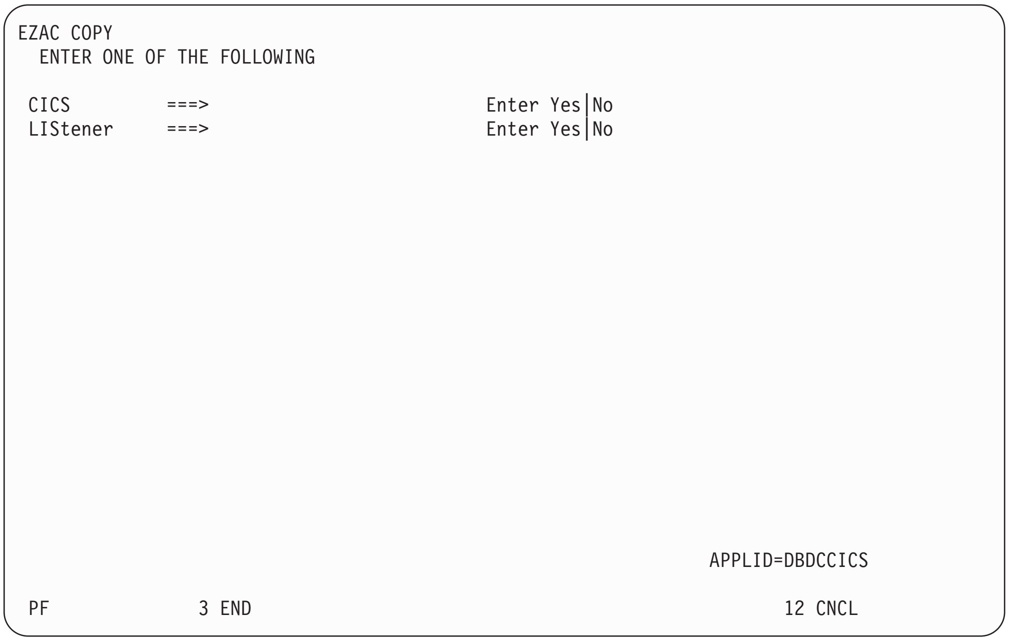


```
EZAC ALTER  LISTENER
  OVERTYPE TO MODIFY

 APPLID       ===> ........           APPLID of CICS System
 TRanid       ===> ........           Transaction Name of Listener
 POrt         ===> .....              Port Number of Listener
 IMMEDiate    ===> ...                Immediate Startup       Yes|No
 BAcklog      ===> ...                Backlog Value for Listener
 NUMsock      ===> ..                 Number of Sockets in Listener
 MINmsgl      ===> ..                 Minimum Message Length
 ACCTime      ===> ..                 Timeout Value for Accept
 GIVTime      ===> ..                 Timeout Value for Givesocket
 REATime      ===> ..                 Timeout Value for Read
 FASTrd       ===> ...                Read immediately        Yes|No
 TRANTrn      ===> ...                Translate Trans. Name   Yes|No
 TRANUsr      ===> ...                Translate User Data     Yes|No
 SECexit      ===> ........           Security Exit Name




 PF 1 HELP       3 END            6 CRSR            9 MSG            12 CNCL
```

Figure 87. EZAC ALTER LISTENER Detail Screen

The system will request a confirmation of the values displayed. After the changes are confirmed, the changed values will be 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:

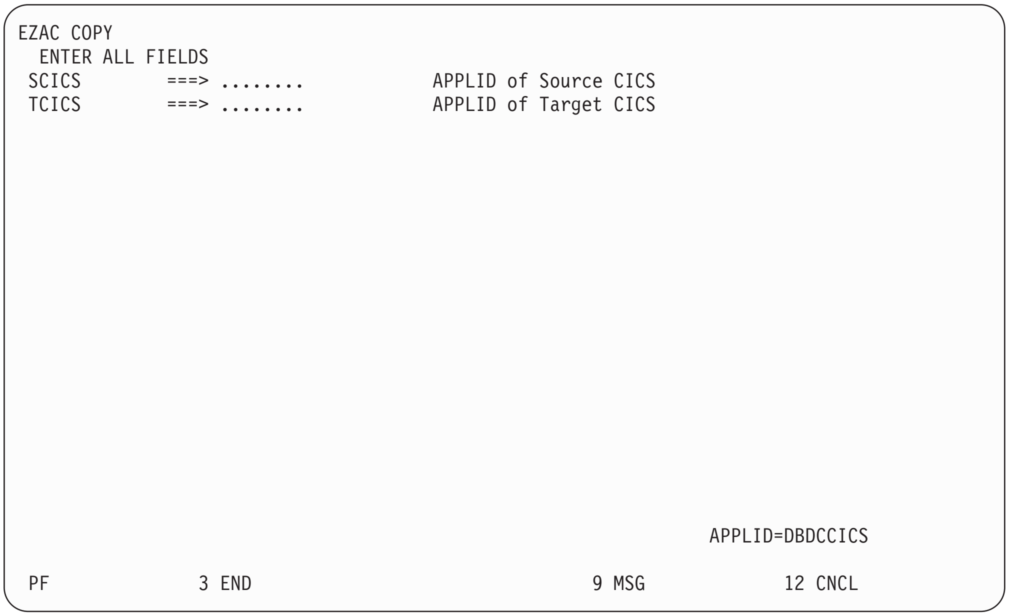


```
EZAC COPY
  ENTER ONE OF THE FOLLOWING

 CICS         ===>                         Enter Yes|No
 LIStener     ===>                         Enter Yes|No




















                                                                APPLID=DBDCCICS

 PF              3 END                                                12 CNCL
```

Figure 88. EZAC COPY Screen

Note: You can short-cut 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:

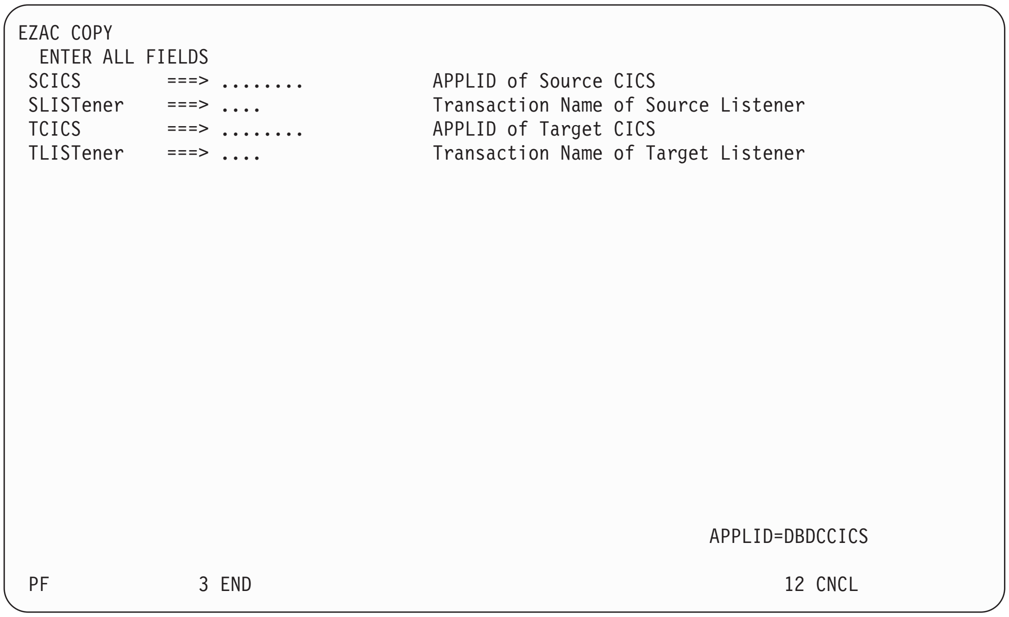


```
EZAC COPY
  ENTER ALL FIELDS
 SCICS       ===> ........           APPLID of Source CICS
 TCICS       ===> ........           APPLID of Target CICS




















                                                                  APPLID=DBDCCICS

 PF             3 END                              9 MSG            12 CNCL
```

Figure 89. EZAC COPY Screen

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:



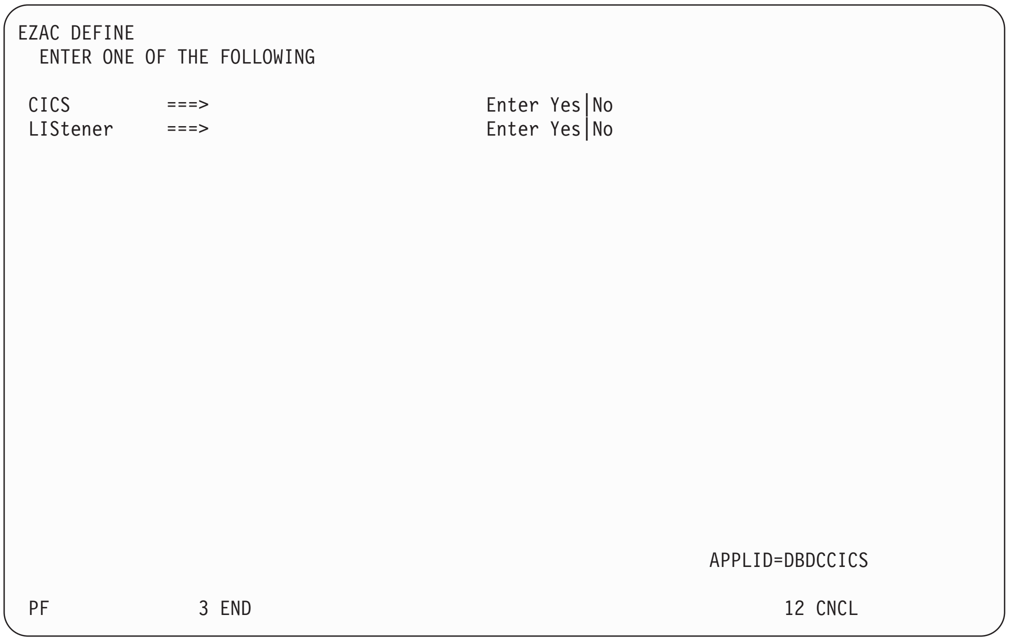
```
EZAC COPY
  ENTER ALL FIELDS
 SCICS       ===> ........           APPLID of Source CICS
 SLISTener   ===> ....               Transaction Name of Source Listener
 TCICS       ===> ........           APPLID of Target CICS
 TLISTener   ===> ....               Transaction Name of Target Listener


















                                                                  APPLID=DBDCCICS

 PF             3 END                                               12 CNCL
```

Figure 90. EZAC COPY Screen

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.

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:

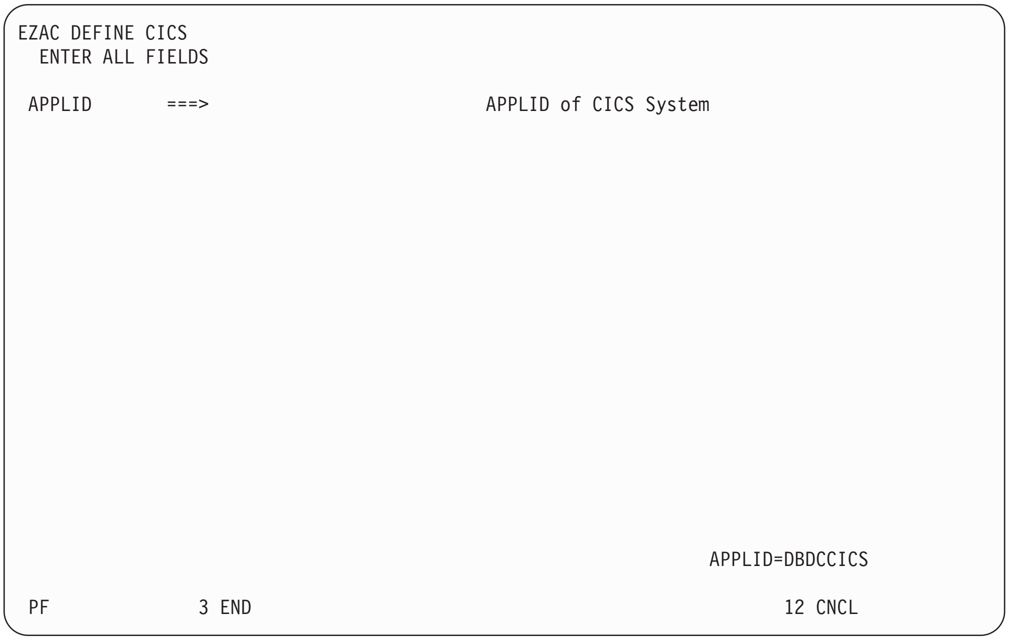


```
EZAC DEFINE
  ENTER ONE OF THE FOLLOWING

 CICS        ===>                         Enter Yes|No
 LIStener    ===>                         Enter Yes|No














                                                                 APPLID=DBDCCICS

 PF             3 END                                                   12 CNCL
```

Figure 91. EZAC DEFINE Screen

Note: You can short-cut this by entering either EZAC DEFINE CICS or EZAC DEFINE LISTENER.

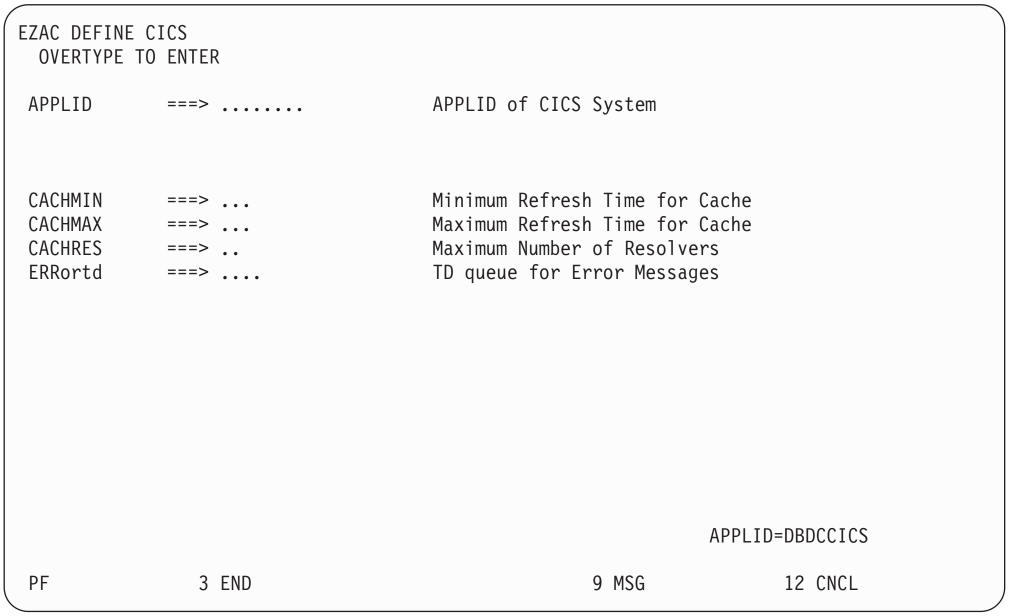
DEFINE CICS: For definition of a CICS object, the following screen is displayed:



```
EZAC DEFINE CICS
  ENTER ALL FIELDS

 APPLID      ===>                         APPLID of CICS System
















                                                                 APPLID=DBDCCICS

 PF             3 END                                                   12 CNCL
```

Figure 92. EZAC DEFINE CICS screen

After the APPLID is entered, the following screen is displayed.

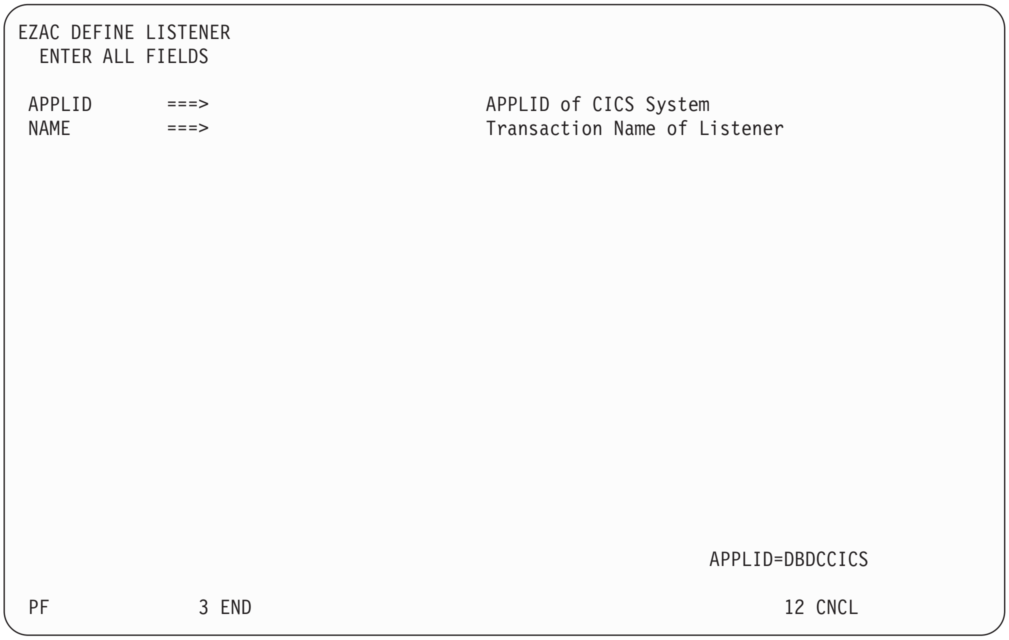


```
EZAC DEFINE CICS
  OVERTYPE TO ENTER

 APPLID       ===> ........           APPLID of CICS System



 CACHMIN      ===> ...                Minimum Refresh Time for Cache
 CACHMAX      ===> ...                Maximum Refresh Time for Cache
 CACHRES      ===> ..                 Maximum Number of Resolvers
 ERRortd      ===> ....               TD queue for Error Messages










                                                          APPLID=DBDCCICS

 PF             3 END                              9 MSG            12 CNCL
```

Figure 93. EZAC DEFINE CICS Detail Screen

After the definition is entered, confirmation is requested. When confirmation is entered, the object is created on the configuration file.

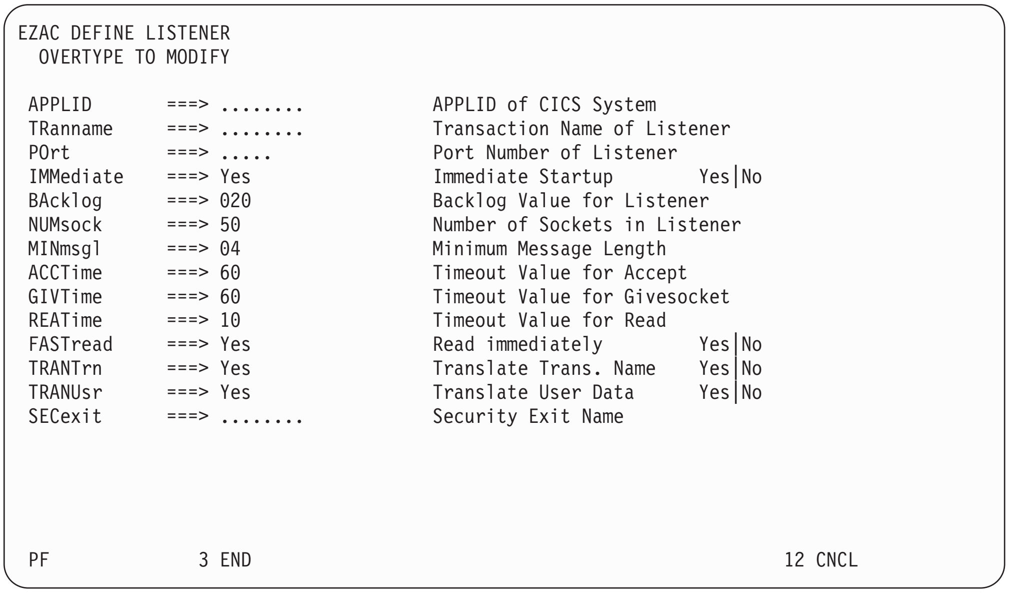
DEFINE LISTENER: For definition of a Listener, the following screen is displayed:



```
EZAC DEFINE LISTENER
  ENTER ALL FIELDS

 APPLID       ===>                         APPLID of CICS System
 NAME         ===>                         Transaction Name of Listener

















                                                          APPLID=DBDCCICS

 PF             3 END                                               12 CNCL
```

Figure 94. EZAC DEFINE LISTENER screen

After the names are entered, the following screen is displayed:

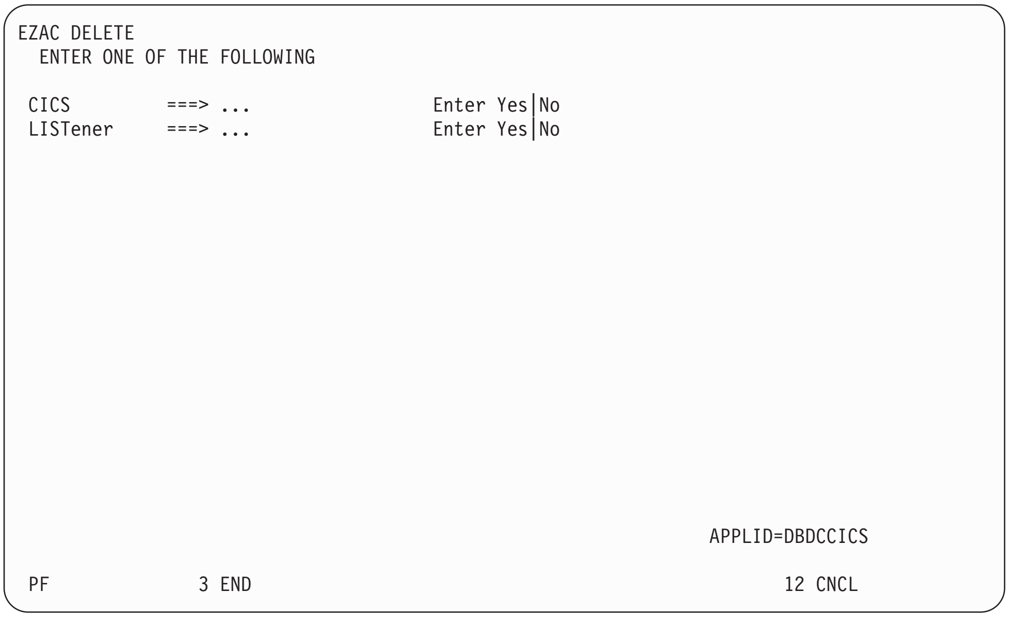


```
EZAC DEFINE LISTENER
  OVERTYPE TO MODIFY

 APPLID       ===> ........           APPLID of CICS System
 TRanname     ===> ........           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
```

Figure 95. EZAC DEFINE LISTENER Detail Screen

After the definition is entered, confirmation is requested. When confirmation is entered, the object is created on the configuration file.

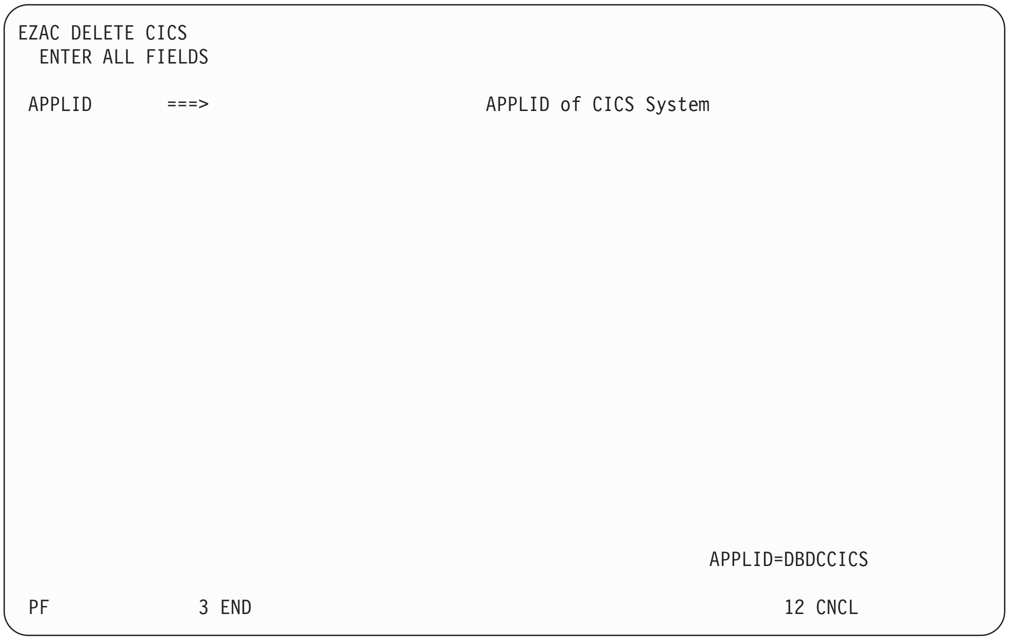
DELETE Function: The DELETE function is used to delete a CICS object or a Listener object. Deleting a CICS object deletes all Listener objects within that CICS object. If you specify DELete on the EZAC initial screen or enter EZAC DEL on a blank screen, the following screen is displayed:



```
EZAC DELETE
  ENTER ONE OF THE FOLLOWING

 CICS         ===> ...                Enter Yes|No
 LISTener     ===> ...                Enter Yes|No

















                                                                APPLID=DBDCCICS

 PF             3 END                                                  12 CNCL
```

Figure 96. EZAC DELETE Screen

DELETE CICS: If you specify DELETE CICS, the following screen is displayed:

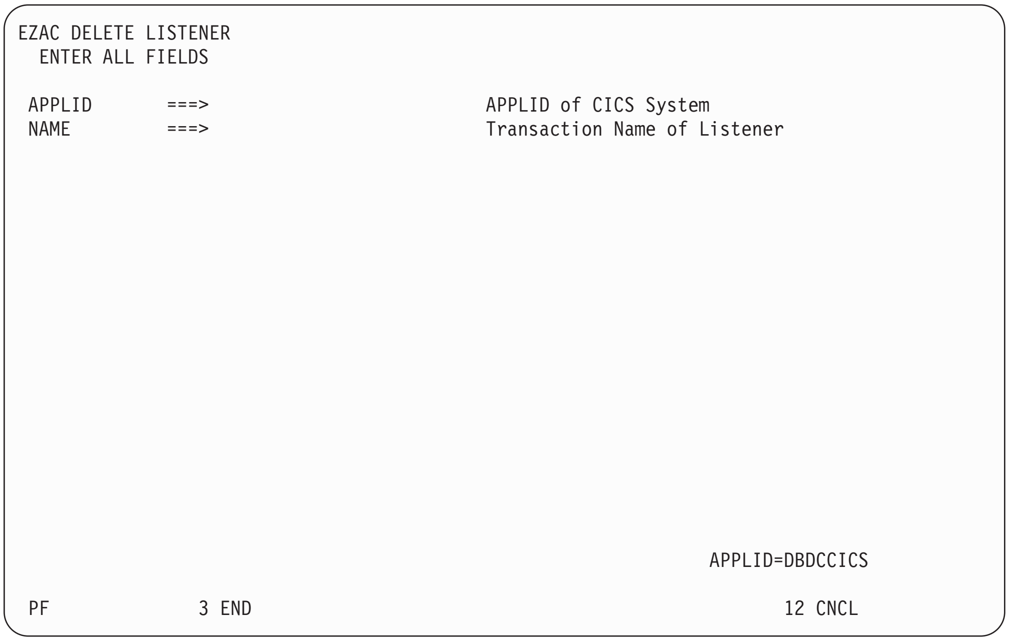


```
EZAC DELETE CICS
  ENTER ALL FIELDS

 APPLID      ===>                         APPLID of CICS System
















                                                              APPLID=DBDCCICS

 PF             3 END                                               12 CNCL
```

Figure 97. EZAC DELETE CICS screen

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:

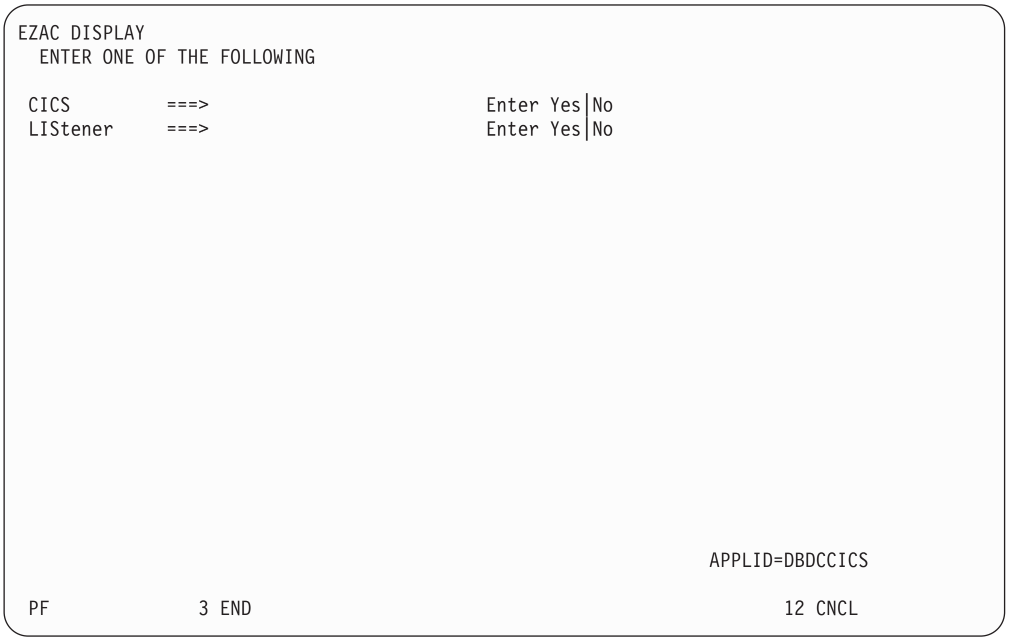


```
EZAC DELETE LISTENER
  ENTER ALL FIELDS

 APPLID      ===>                         APPLID of CICS System
 NAME        ===>                         Transaction Name of Listener















                                                              APPLID=DBDCCICS

 PF             3 END                                               12 CNCL
```

Figure 98. EZAC DELETE LISTENER screen

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. If you specify DISplay on the initial EZAC screen or enter EZAC DIS on a blank screen, the following screen is displayed:

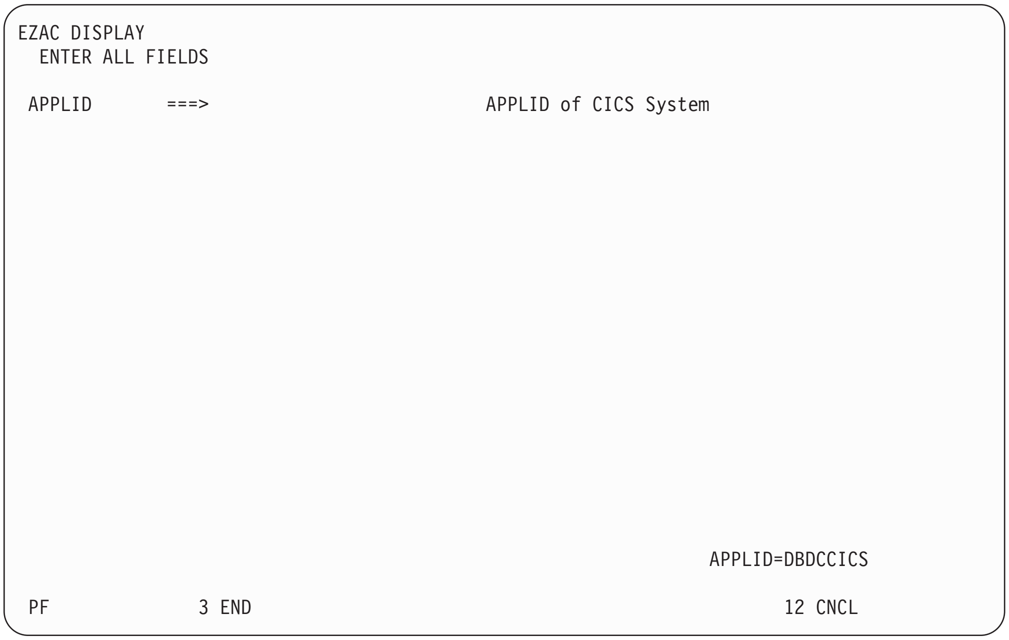


```
EZAC DISPLAY
  ENTER ONE OF THE FOLLOWING

 CICS         ===>                          Enter Yes|No
 LIStener     ===>                          Enter Yes|No




















                                                                 APPLID=DBDCCICS

 PF              3 END                                                  12 CNCL
```

Figure 99. EZAC DISPLAY Screen

Note: You can short-cut this by entering either EZAC DISPLAY CICS or EZAC DISPLAY LISTENER.

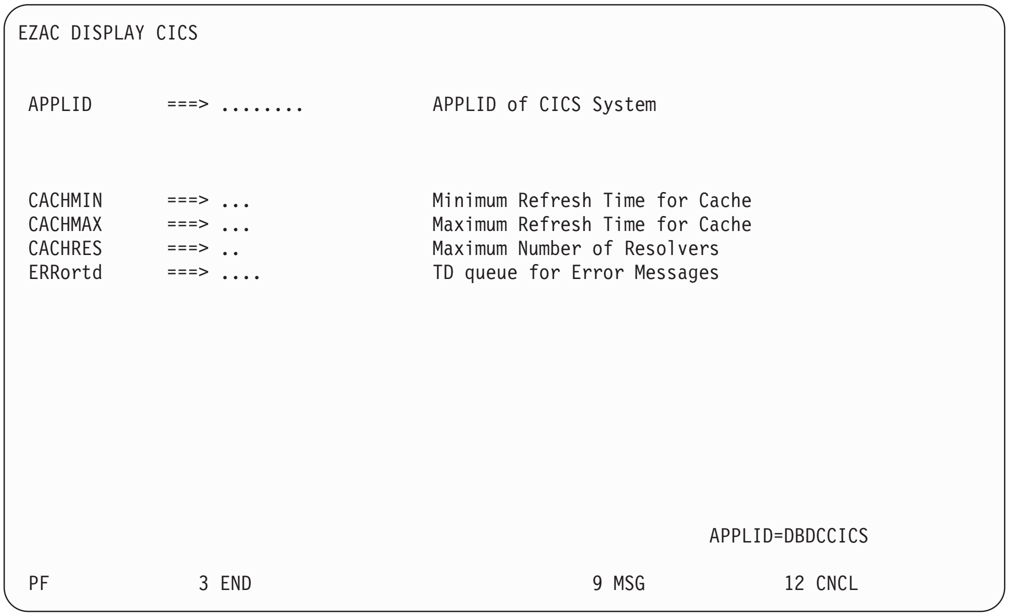
DISPLAY CICS: If you specify DISPLAY CICS, the following screen is displayed:



```
EZAC DISPLAY
  ENTER ALL FIELDS

 APPLID       ===>                          APPLID of CICS System





















                                                                 APPLID=DBDCCICS

 PF              3 END                                                  12 CNCL
```

Figure 100. EZAC DISPLAY CICS screen

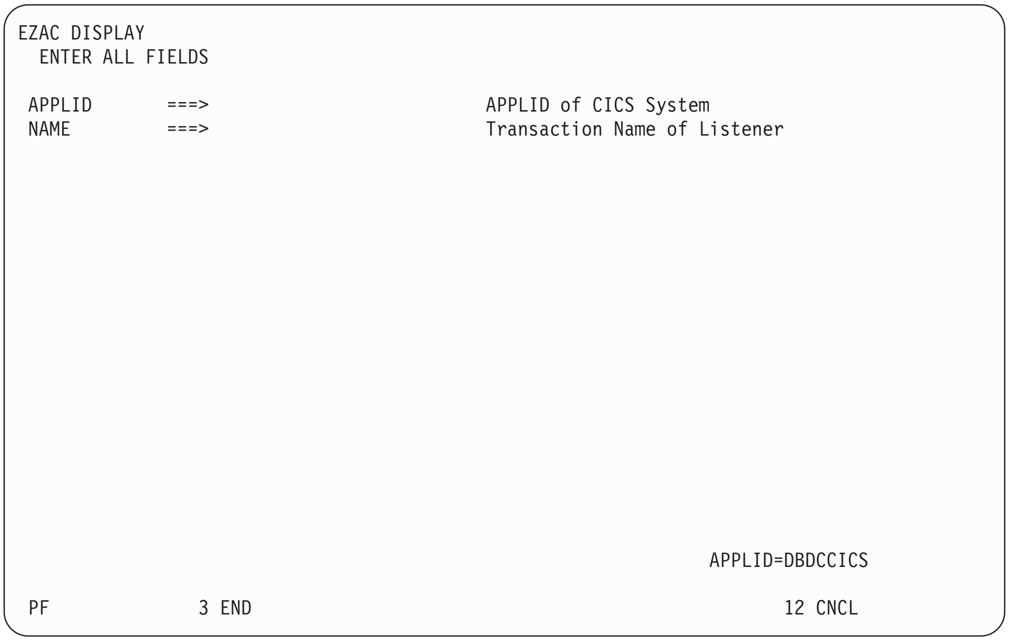
After the APPLID is entered, the following screen is displayed:



```
EZAC DISPLAY CICS


 APPLID       ===> ........               APPLID of CICS System




 CACHMIN      ===> ...                    Minimum Refresh Time for Cache
 CACHMAX      ===> ...                    Maximum Refresh Time for Cache
 CACHRES      ===> ..                     Maximum Number of Resolvers
 ERRortd      ===> ....                   TD queue for Error Messages










                                                                 APPLID=DBDCCICS

 PF              3 END                                9 MSG             12 CNCL
```

Figure 101. EZAC DISPLAY CICS Detail Screen

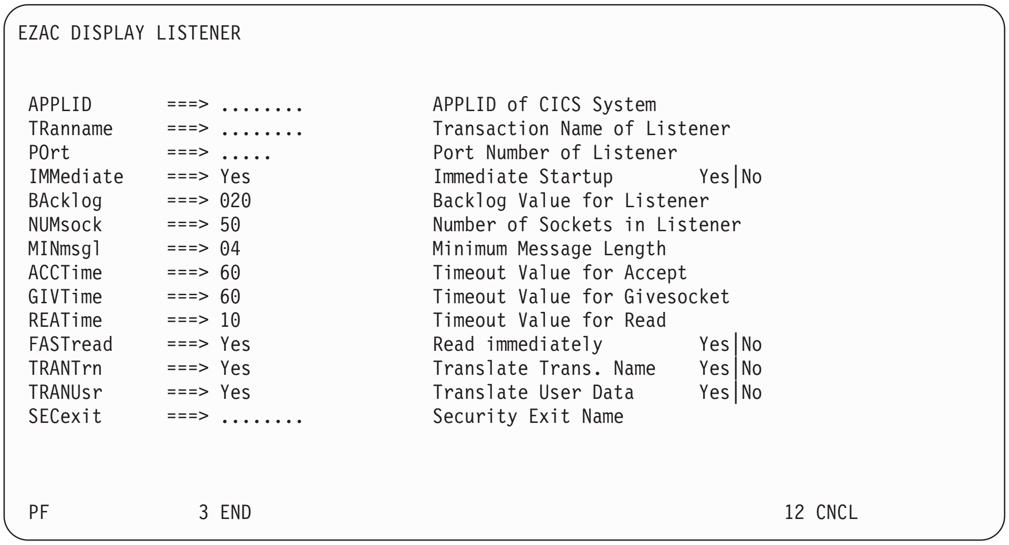
DISPLAY LISTENER: If you specify DISPLAY LISTENER, the following screen is displayed:



```
EZAC DISPLAY
  ENTER ALL FIELDS

 APPLID       ===>                             APPLID of CICS System
 NAME         ===>                             Transaction Name of Listener
















                                                                 APPLID=DBDCCICS

 PF              3 END                                                  12 CNCL
```

Figure 102. EZAC DISPLAY LISTENER screen

After the APPLID and name are entered, the following screen is displayed

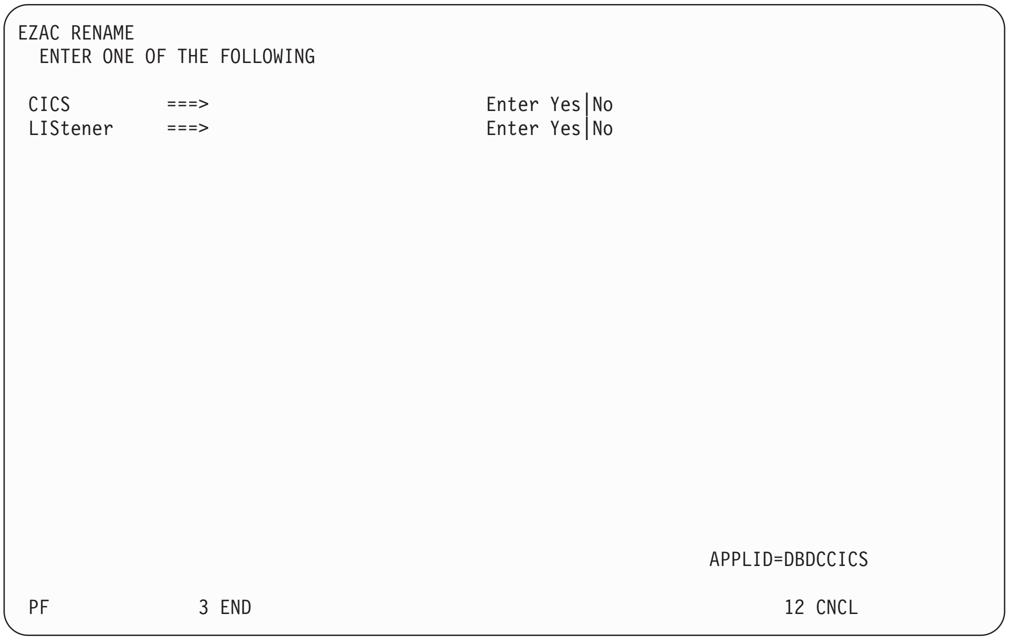


```
EZAC DISPLAY LISTENER


 APPLID       ===> ........           APPLID of CICS System
 TRanname     ===> ........           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
```

Figure 103. EZAC DISPLAY LISTENER Detail Screen

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:

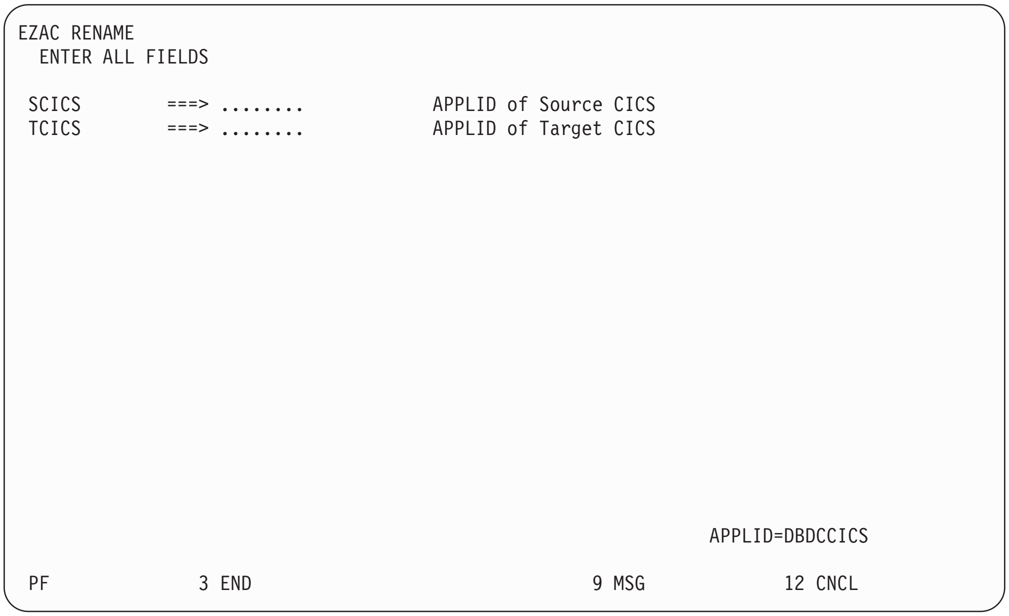


```
EZAC RENAME
  ENTER ONE OF THE FOLLOWING

 CICS         ===>                         Enter Yes|No
 LIStener     ===>                         Enter Yes|No

















                                                          APPLID=DBDCCICS

 PF             3 END                                                  12 CNCL
```

Figure 104. EZAC RENAME Screen

Note: You can short-cut 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:

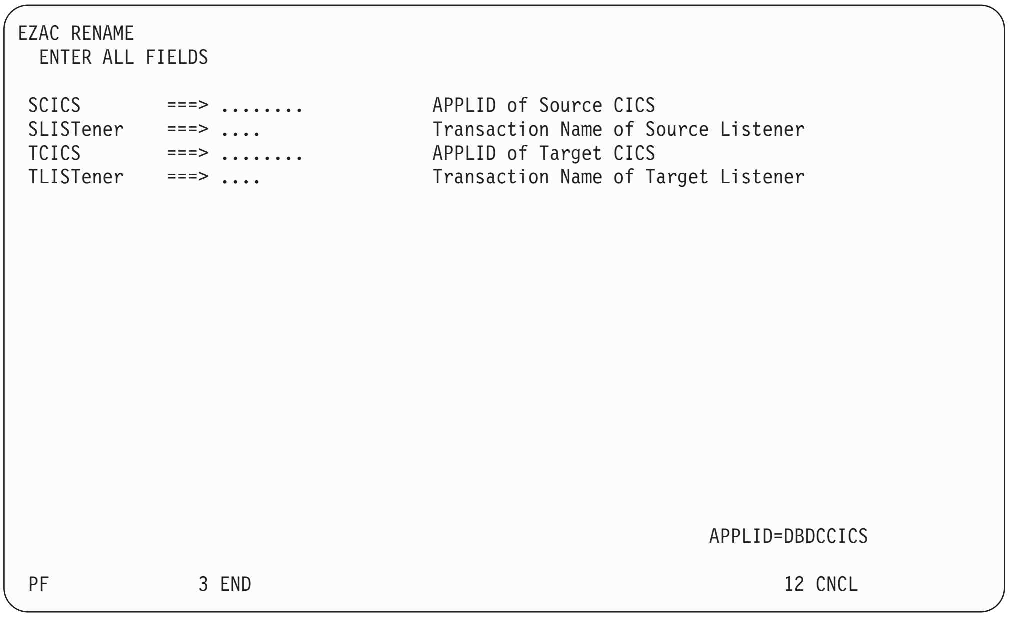


```
EZAC RENAME
  ENTER ALL FIELDS

 SCICS       ===> ........           APPLID of Source CICS
 TCICS       ===> ........           APPLID of Target CICS





















                                                         APPLID=DBDCCICS

 PF             3 END                            9 MSG            12 CNCL
```

Figure 105. EZAC RENAME CICS Screen

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
  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



















                                                         APPLID=DBDCCICS

 PF             3 END                                             12 CNCL
```

Figure 106. EZAC RENAME LISTENER Screen

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 13. 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 437.

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 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

There are three steps to using 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” on page 437.

2. Define the cache files to CICS. See “Step 2: Define the Cache File to CICS” on page 439.
3. Use EZACIC25 to replace gethostbyname calls in CICS application modules. See “Step 3: Execute EZACIC25” on page 439.

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.

The specifications for the EZACICR macro are as follows:

Operand
Meaning

TYPE There are three acceptable values:

Value **Meaning**

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 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:

Field Name
Description

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:

Field Name
: **Description**

Return Code
: A fullword binary variable specifying the results of the function:

Value	Meaning
-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.

Value	Meaning
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:

Value	Meaning
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.

HOSTENT Structure

The returned HOSTENT structure is shown in Figure 107.

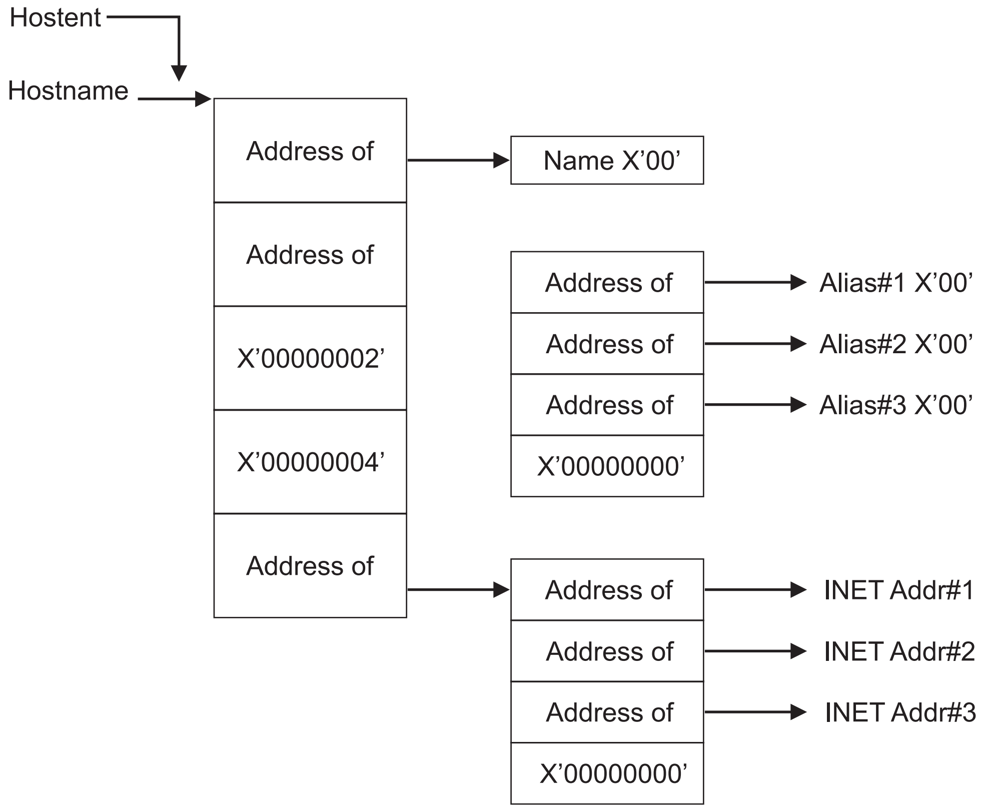


Figure 107. The DNS Hostent

Chapter 14. Starting and Stopping the CICS Listener Support

Overview

This chapter 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 446.

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.

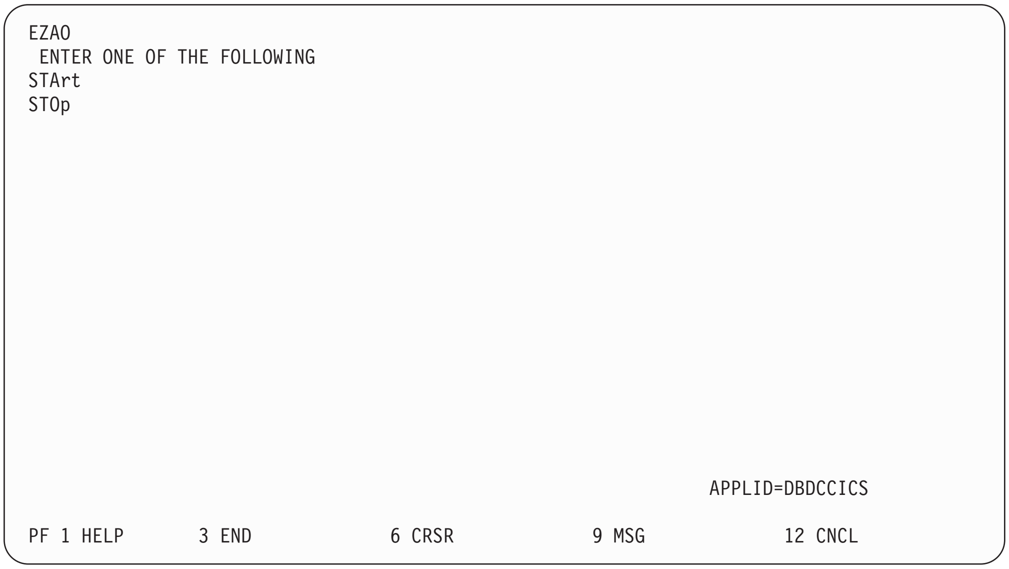
Note: The EZAO transaction **must** be running on the CICS where you want to start the CICS Listener Support. You may not start a CICS Listener Support from a different CICS.

CICS Listener Support Shutdown
Stops the interface in a CICS address space.

Listener Startup
Starts a Listener in a CICS address space.

Listener Shutdown
Stops a Listener in a CICS address space.

When you enter EZAO, the following screen displays.

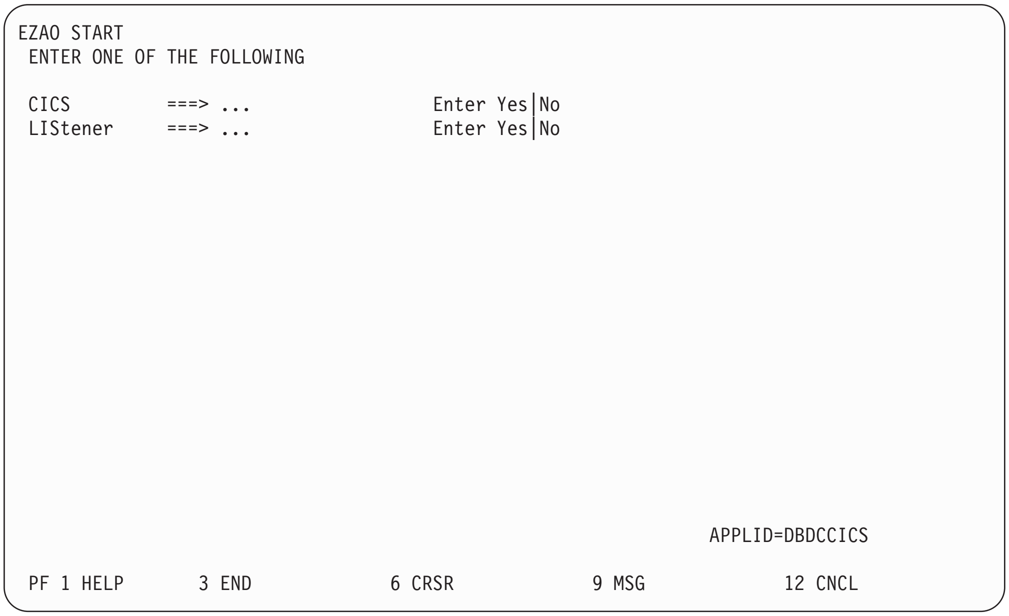


```
EZAO
 ENTER ONE OF THE FOLLOWING
STArt
STOp

                                                               APPLID=DBDCCICS

PF 1 HELP      3 END            6 CRSR            9 MSG            12 CNCL
```

Figure 108. EZAO Initial Screen

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.

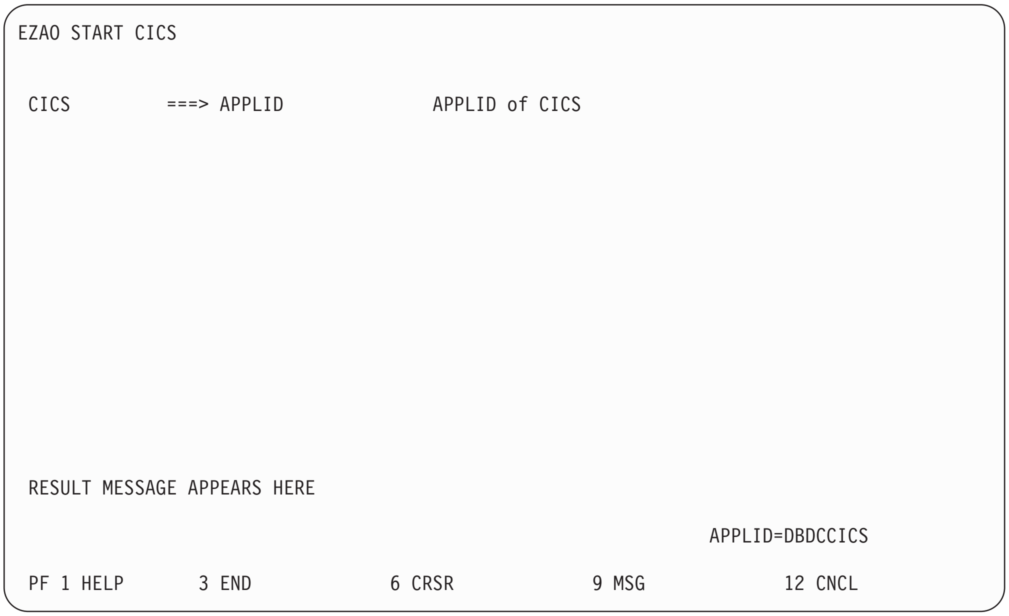


```
EZAO START
 ENTER ONE OF THE FOLLOWING

 CICS        ===> ...                Enter Yes|No
 LIStener    ===> ...                Enter Yes|No

                                                                APPLID=DBDCCICS

 PF 1 HELP      3 END            6 CRSR            9 MSG            12 CNCL
```

Figure 109. EZAO START Screen

START CICS Listener Support

If you enter EZAO START CICS, the following screen displays.

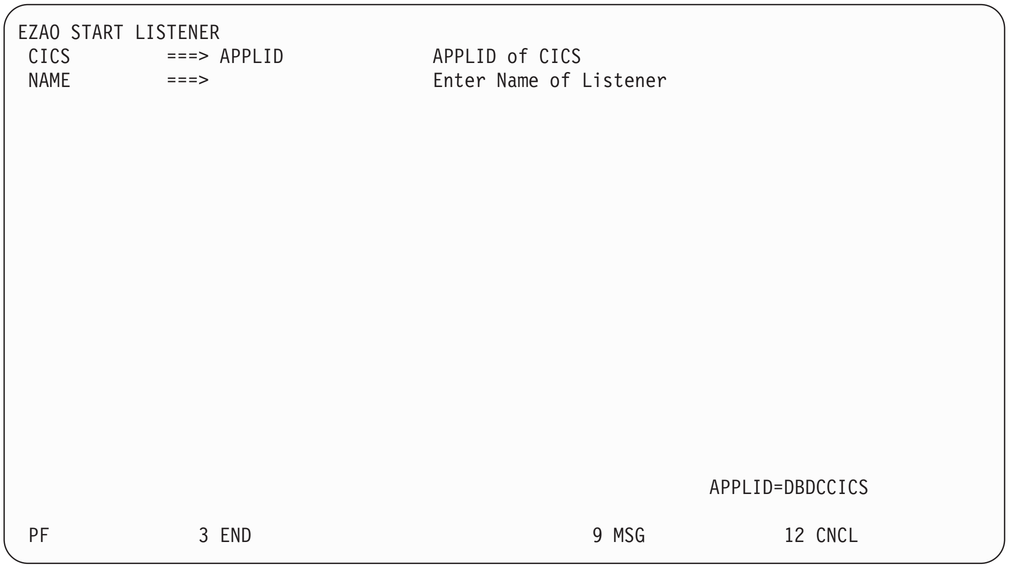


```
EZAO START CICS

 CICS         ===> APPLID              APPLID of CICS













 RESULT MESSAGE APPEARS HERE

                                                                 APPLID=DBDCCICS

 PF 1 HELP       3 END             6 CRSR            9 MSG            12 CNCL
```

Figure 110. EZAO START CICS Response Screen

START A LISTENER

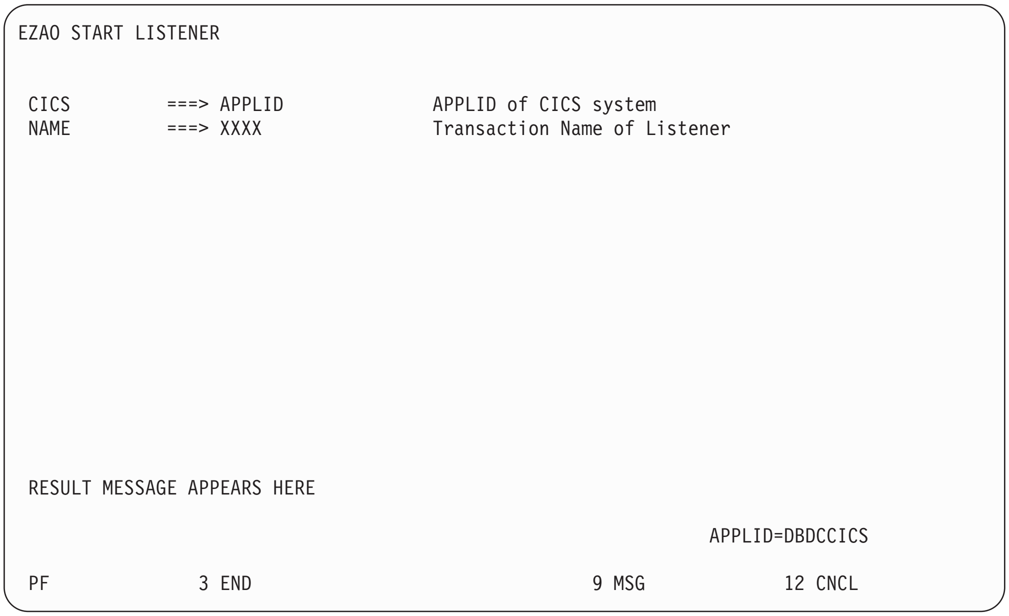
If you enter EZAO START LISTENER, the following screen displays.



```
EZAO START LISTENER
 CICS         ===> APPLID              APPLID of CICS
 NAME         ===>                     Enter Name of Listener
















                                                                 APPLID=DBDCCICS

 PF              3 END                               9 MSG            12 CNCL
```

Figure 111. EZAO START LISTENER Screen

After you enter the listener name, the listener is started. The following screen displays; the results appear in the message area.

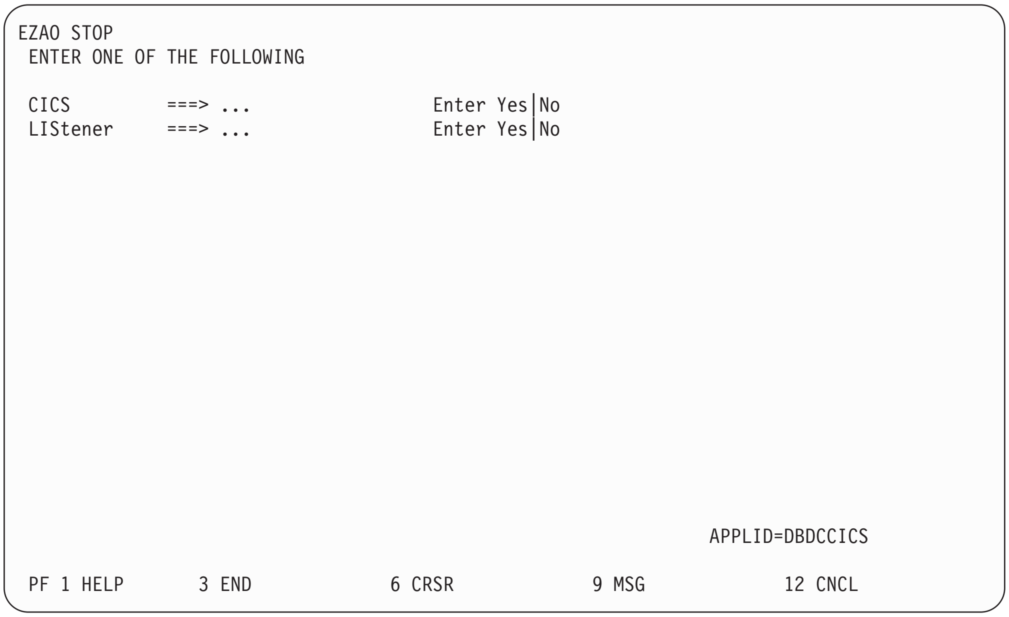


```
EZAO START LISTENER


 CICS         ===> APPLID              APPLID of CICS system
 NAME         ===> XXXX                Transaction Name of Listener














 RESULT MESSAGE APPEARS HERE

                                                                 APPLID=DBDCCICS

 PF              3 END                                9 MSG            12 CNCL
```

Figure 112. EZAO START LISTENER Result Screen

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:

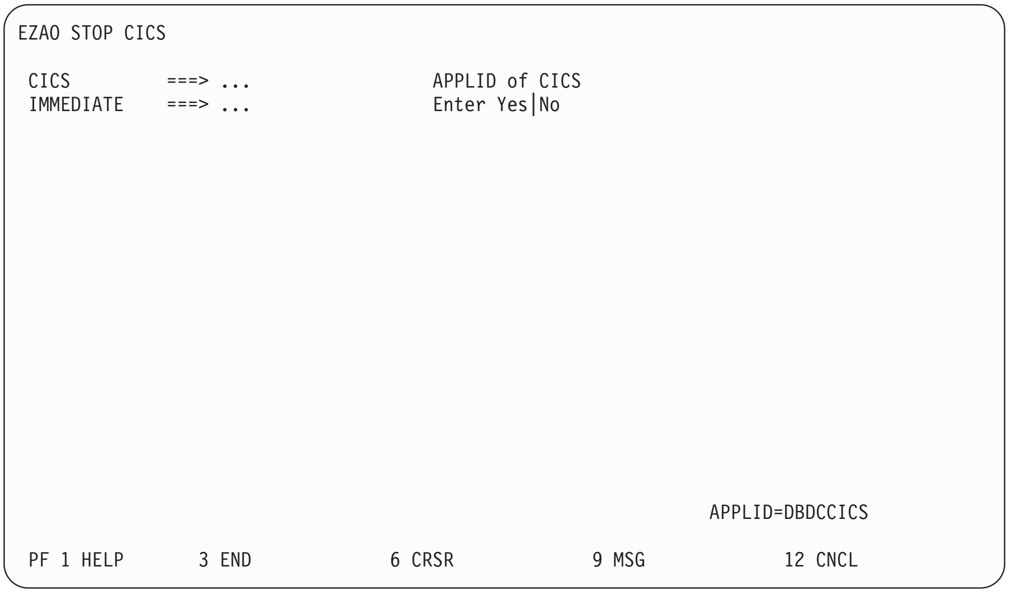


```
EZAO STOP
 ENTER ONE OF THE FOLLOWING

 CICS         ===> ...                 Enter Yes|No
 LIStener     ===> ...                 Enter Yes|No

















                                                                 APPLID=DBDCCICS

 PF 1 HELP       3 END             6 CRSR             9 MSG            12 CNCL
```

Figure 113. EZAO STOP Screen

STOP CICS Listener Support

If you specify EZAO STOP CICS, the following screen is displayed



```
EZAO STOP CICS

 CICS        ===> ...               APPLID of CICS
 IMMEDIATE   ===> ...               Enter Yes|No














                                                        APPLID=DBDCCICS

 PF 1 HELP      3 END          6 CRSR          9 MSG          12 CNCL
```

Figure 114. EZAO STOP CICS Screen

Two options are available to stop CICS Listener Support:

IMMEDIATE=NO
This should be used in most cases, because it causes the graceful 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.

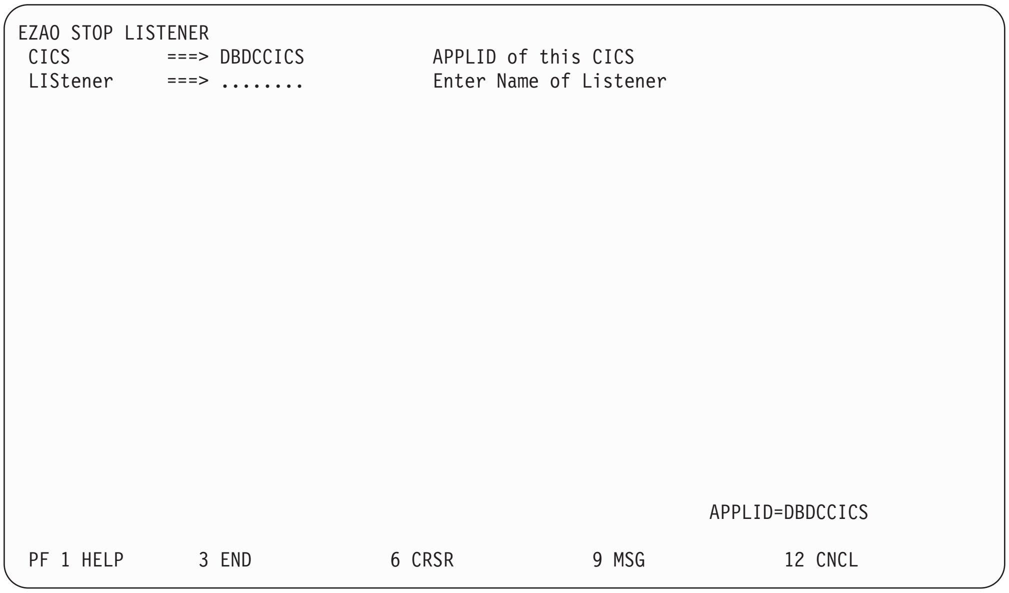
IMMEDIATE=YES
This option is reserved for unusual situations and causes the abrupt termination of the CICS Listener Support. It has the following effect on applications using this API:

- 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.

STOP A LISTENER

If you specify STOP LISTENER, the following screen displays.



```
EZAO STOP LISTENER
 CICS        ===> DBDCCICS           APPLID of this CICS
 LIStener    ===> ........           Enter Name of Listener




















                                                                    APPLID=DBDCCICS

 PF 1 HELP      3 END            6 CRSR            9 MSG            12 CNCL
```

Figure 115. EZAO STOP LISTENER Screen

When 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.

Note: The following user abend codes may 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.

Chapter 15. Writing Your Own Listener

Basic Requirements

The CICS Listener Support provides a structure which supports up to 255 listeners. These listeners may be multiple copies of the IBM-supplied listener, user-written listeners, or a combination of the two. You may 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 468.

Pre-Requisites

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 multi-threading 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 may 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 16, “External Data Structures,” on page 453:

- 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=CFG,TYPE=CSECT
```

If you are going to read the configuration file using LOCATE mode you can define a DSECT for the area as follows:

```
EZACICA AREA=CFG,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.

Chapter 16. 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=CFG,TYPE=DSECT
EZACICA AREA=CFG,TYPE=CSECT
```

Table 9. 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 • C = CICS Object Record • L = Listener Object Record	
(Reserved)	3 byte hex	Reserved for IBM Use	00
CICS Record Format			
CFCTRAN	4 byte char	Transaction name for Configuration Record	<<<<
CFCTCPIP	8-byte char	Address space name of TCP/IP (1)	
CFCNOTSK	Halfword bin	Number of reusable tasks (1)	20
CFCSTIME	Halfword bin	Resolver Cache minimum refresh time	15
CFCLTIME	Halfword bin	Resolver Cache maximum refresh time	30
CFCNORES	Halfword bin	Resolver Cache number of concurrent resolvers	10
CFCDPRTY	Halfword bin	Limit Priority of Subtask (LPMOD value in ATTACH macro) (1)	0
CFCENAME	4-byte character	Name of Transient Data Message Queue	EZAM
Listener Record Format			
CFLTRAN	4 byte char	Transaction name for this listener	EZAL
CFLPORT	Halfword bin	Port number for this listener	
CFLBKLOG	Halfword bin	Backlog value for listen call	10
CFLNSOCK	Halfword bin	Number of sockets used by listener	50
CFLMMIN	Halfword bin	Minimum length of input message	4
CFLLTIM	Halfword bin	Timeout value (seconds) for accept	60
CFLRTIM	Halfword bin	Timeout value (seconds) for read	0
CFLGTIM	Halfword bin	Timeout value (seconds) for givesocket	30

Table 9. Configuration File Format (continued)

Field Name	*Field Type*	*Description*	*Default Value*
CFLOPT	1 byte hex	Listener Options **Value** **Description** **B'00000001'** Immediate Startup **B'00000110'** Translate entire message **B'00000010'** Translate trans code only **B'00000100'** Translate user data only **B'00001000'** Issue READ immediately after ACCEPT	B'00001111'
CFLSECXT	8-byte char	Name of Security Exit (1)	
CFLWLMN1	12-byte char	WLM Group Name 1 (1)	
CFLWLMN2	12-byte char	WLM Group Name 2 (1)	
CFLWLMN3	12-byte char	WLM Group Name 3	

(1) Not used within 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 10. Global Work Area Format

Field Name	**Field Type**	**Description**	**Default Value**
Beginning of Global Work Area Eyecatcher			
GWACMDSC	8 byte char	Identifier	EZACICGW
Beginning of Startup Module Heritage			
GWACMNAM	8 byte hex	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	
End of Startup Module Heritage			
Beginning of Task-Related User Exit Heritage			
GWATRNAM	8 byte hex	Task Related User Exit Module Name	EZACIC01

Table 10. Global Work Area Format (continued)

Field Name	Field Type	Description	Default Value
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)	
End of Task-Related User Exit Heritage			
Beginning of IBM Listener Heritage			
GWAMSNAM	8 byte hex	IBM Listener Module Name	EZACIC02
GWAMSVER	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	
End of IBM Listener Heritage			
Beginning of Attached Subtask Heritage			
GWASTNAM	8 byte hex	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)	
End of Attached Subtask Heritage			
GWACMIBM	154 byte char	Copyright Statement	
	42 byte char	Reserved Area	
End of Global Work Area Eyecatcher			
GWAUSCNT	Fullword bin	Use count for this GWA	
GWABKWRD	Fullword bin	Attached (non-pool) task chain anchor backward address (1)	
GWAFOWRD	Fullword bin	Attached (non-pool) 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	TCPIP Address Space Name (1)	
GWALCAAD	Fullword bin	Address of First Listener Control Area	
GWA03PSA	Fullword bin	Address of EZASOH03 Load Module (1)	
GWANTASK	Halfword bin	Number of Reusable Tasks (1)	
GWANLIST	Halfword bin	Number of Listeners	

Table 10. Global Work Area Format (continued)

Field Name	Field Type	Description	Default Value
GWATSTAT	1 byte char	Task Related User Exit Status **Value** **Meaning** **E** TRUE is enabled **I** Immediate Shutdown Requested/Processing **Q** Quiescent Shutdown Requested/Processing	
GWARSHUT	1 byte char	EZAO Shutdown Request Indicator **Value** **Meaning** **I** Immediate Shutdown Requested/Processing **Q** Quiescent Shutdown Requested/Processing	
GWACSTAT	1 byte bin	CICS Execution Status (1)	
GWAVOSYS	1 byte bin	MVS Version (1)	
GWAOPREL	2 byte bin	MVS Release	
GWACIVER	2 byte char	CICS Version (1)	
GWACIREL	1 byte char	CICS Release (1)	
GWACIMOD	1 byte char	CICS Modification (1)	
GWATOKEN	8 byte char	Token for OS/390 Registration/Deregistration (1)	
GWAMSGMD	8 byte char	Name of Message Module	
End of Global Work Area Eyecatcher			
GWATDMSG	4 byte char	Name of TD Queue for Message Delivery	
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 11. COMMAREA Format for EZACIC20

Field Name	*Field Type*	*Description*	*Default Value*
P20TYPE	1 byte char	Type of Function **Value** **Meaning** **I** Initialization **T** Immediate Termination **D** Deferred Termination	

Table 11. COMMAREA Format for EZACIC20 (continued)

Field Name	Field Type	Description	Default Value
P20OBJ	1 byte char	Type of Function **Value** **Meaning** **C** CICS Listener Support **L** Listener	
P20LIST	4 byte char	Transaction Name of Listener	
P20RET	1 byte bin	Return Code **Value** **Meaning** **B'00000000'** No Errors Encountered **B'00000001'** Errors in CICS Listener Support Initialization **B'00000010'** Errors in Listener Initialization **B'00000100'** Errors in CICS Listener Support Termination **B'00001000'** Errors in Listener Termination **B'00010000'** Errors in COMMAREA Contents. **B'00100000'** Errors in CICS/TS for VSE/ESA.	

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 12. 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	

Table 12. Listener Control Area (LCA) (continued)

Field Name	*Field Type*	*Description*	*Default Value*
LCASTAT	1 byte bin	Status of this Listener **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	
LCAPHASE	1 byte char	Execution Phase for IBM Listener	

Chapter 17. CICS Listener Programming Considerations

Overview

This chapter describes typical sequences of calls for client, concurrent server (with associated child server processes), and iterative server programs. The contents of the chapter are:

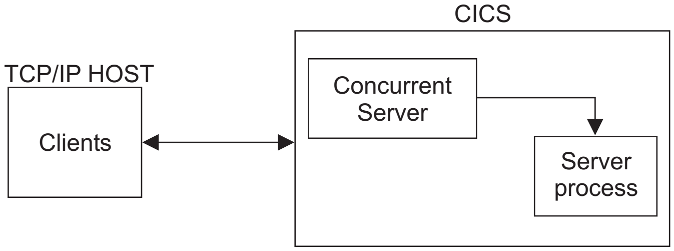
- Four setups for writing CICS TCP/IP applications:
  1. Concurrent server (the supplied Listener transaction) and child server processes run under CICS TCP/IP
  2. The same as 1 but with a user-written concurrent server
  3. An iterative server running under CICS TCP/IP
  4. A client application running under CICS TCP/IP
- Socket addresses
- GETCLIENTID, GIVESOCKET, and TAKESOCKET commands
- The Listener program

Writing CICS TCP/IP Applications

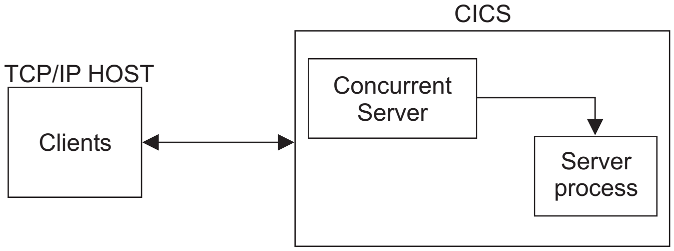
This chapter considers in detail 4 TCP/IP setups in which CICS TCP/IP applications are used in various parts of the client/server system.

The setups are:

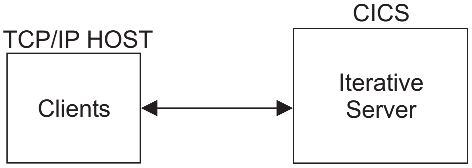
1. **The Client-Listener-Child Server Application Set**. The concurrent server and child server processes run under CICS TCP/IP. The concurrent server is the supplied **Listener** transaction. The client might be running TCP/IP under the OS/2 operating system or one of the various UNIX operating systems such as AIX.



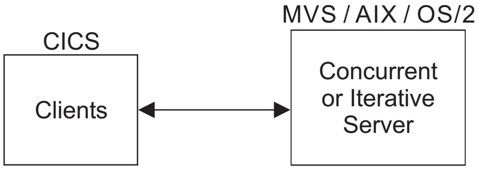
2. **Writing Your Own Concurrent Server**. This is the same setup as the first except that a user-written concurrent server is being used instead of the IBM Listener.



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 116 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.

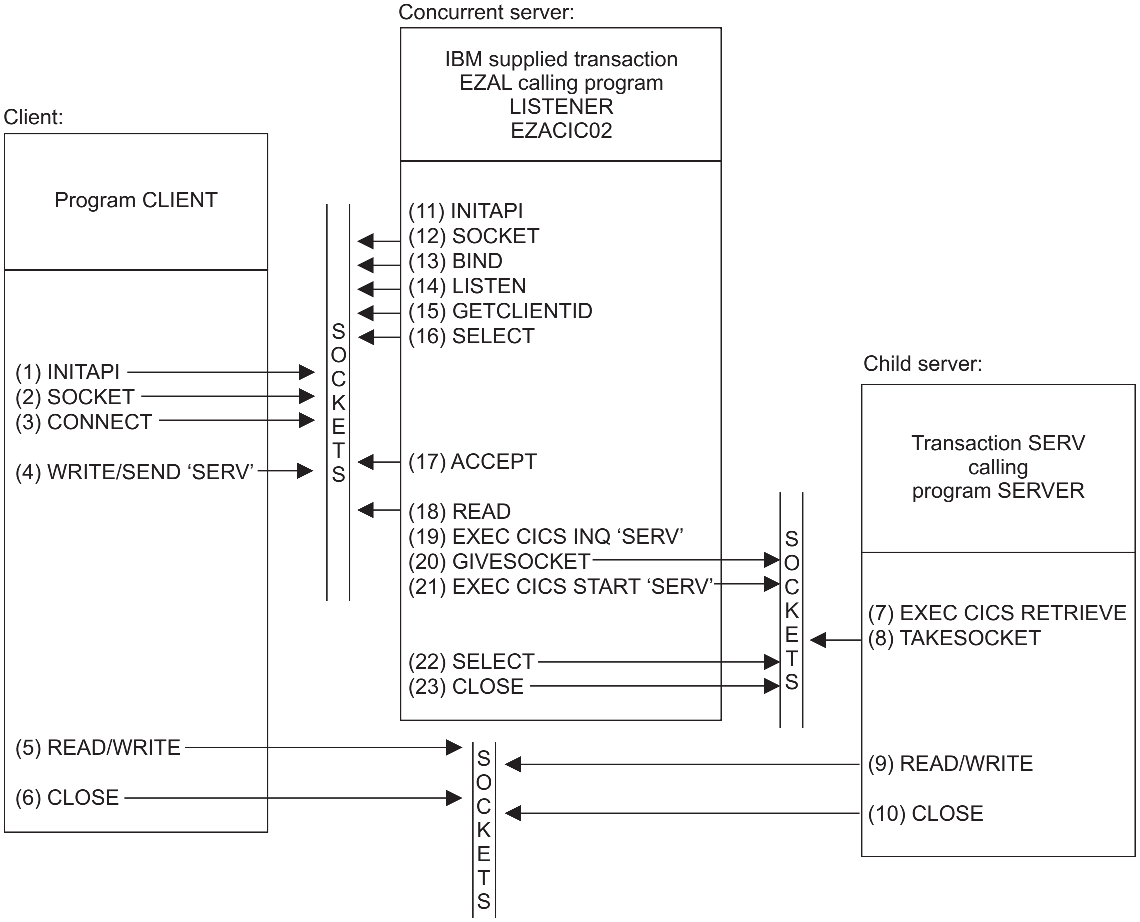


Figure 116. The Sequence of Sockets Calls

Client Call Sequence

Table 13 explains the functions of each of the calls listed in Figure 116 on page 460.

Table 13. 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: • The domain, or addressing family • The type of socket • The protocol 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. If successful, the SOCKET call returns a socket descriptor, s, which is always a small integer. Notice that the socket obtained is not yet attached to any local or destination address.
(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” on page 462, where the Listener calls in Figure 116 are explained.

Child Server Call Sequence

Table 14 explains the functions of each of the calls listed in Figure 116 on page 460.

Table 14. 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.

Table 14. Calls for the Server Application (continued)

(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. Obviously, the client and child server applications have the same functions.

Concurrent Server Call Sequence

Table 15 explains the functions of each of the steps listed in Figure 116 on page 460.

Table 15. Calls for the Concurrent Server Application

(11) INITAPI	Connects the application to TCP/IP, as in Table 13.
(12) SOCKET	This obtains a socket, as in Table 13.
(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 *bound* 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.

Table 15. Calls for the Concurrent Server Application (continued)

(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 469, the parameter `LSTN-CLIENTID` defines the Listener.
(22) SELECT [2]	Again, the SELECT call is used to monitor TCP/IP activity. This time, SELECT returns when the child server issues a TAKESOCKET call.
(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 GIVESOCKET/TAKESOCKET sequence of calls.

3. The Iterative Server CICS TCP/IP Application

Figure 117 shows the sequence of socket calls involved in a simple client-iterative server setup.

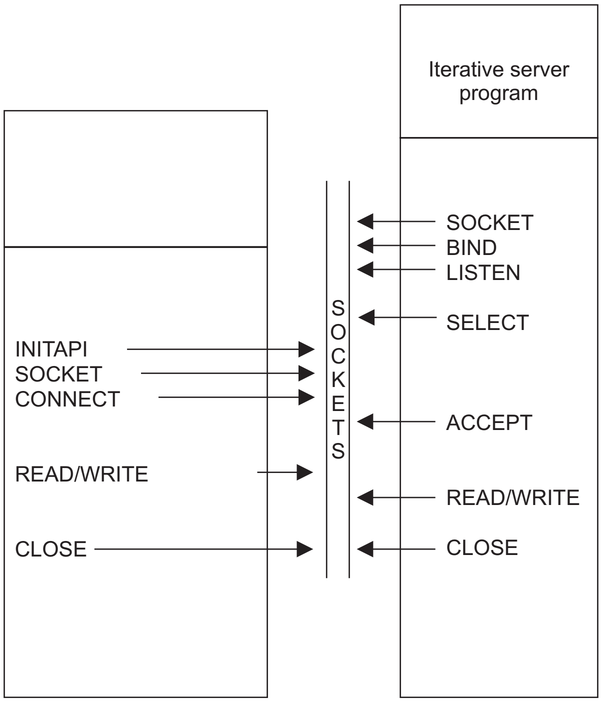


Figure 117. Sequence of Socket Calls with an Iterative Server

[2] This SELECT is the same as the SELECT call in Step 16. They are shown as two calls to clarify the functions being performed.

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 118 shows the sequence of calls in a CICS client-remote server setup. The calls are similar to the previous examples.

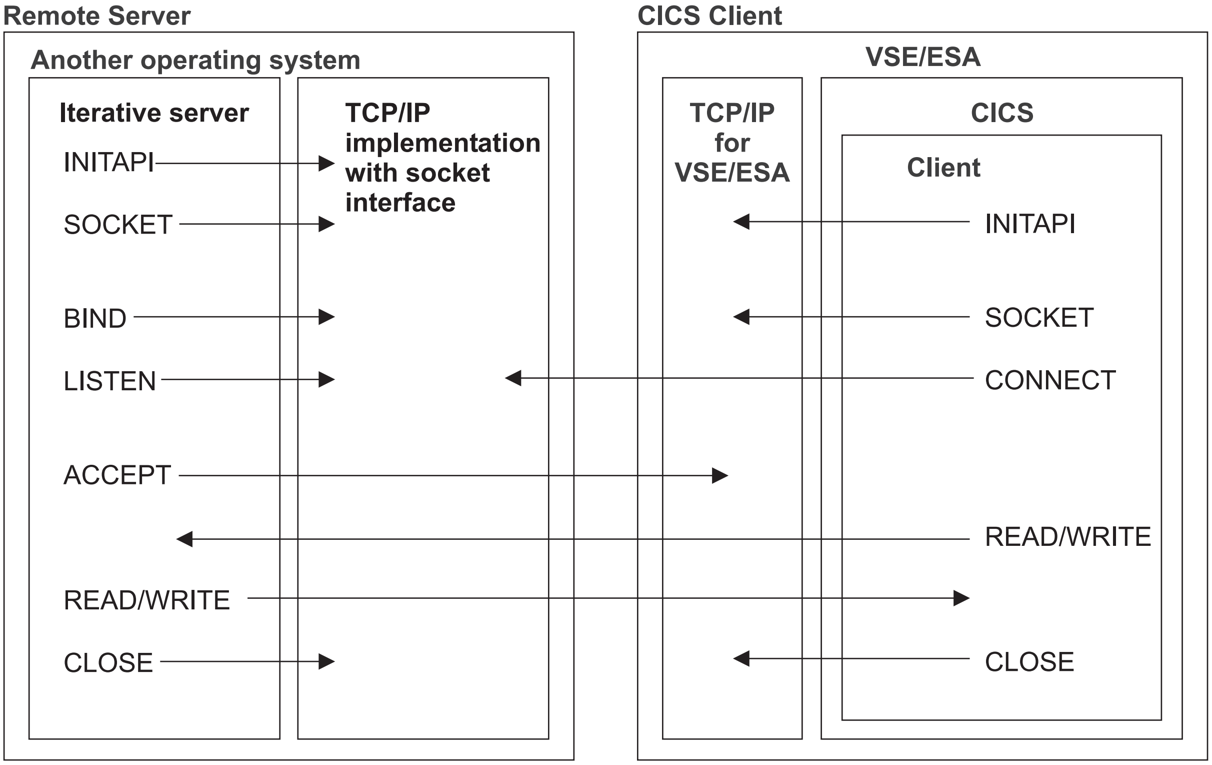


Figure 118. Sequence of Socket Calls between a CICS Client and a Remote Iterative Server

Figure 118 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 116 on page 460.

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 118, 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 118, 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. 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 466).
- 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.

For COBOL and Assembler Language Programs

The address structure of an internet socket address should be defined as follows:

Parameter	Assembler	COBOL
NAME STRUCTURE:		
FAMILY	H	PIC 9(4) BINARY
PORT	H	PIC 9(4) BINARY
ADDRESS	F	PIC 9(8) BINARY
ZEROS	XL8	PIC X(8)

For C Programs

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

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

The socket calls GETCLIENTID, GIVESOCKET, and TAKESOCKET are in CICS 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 119 on page 467.

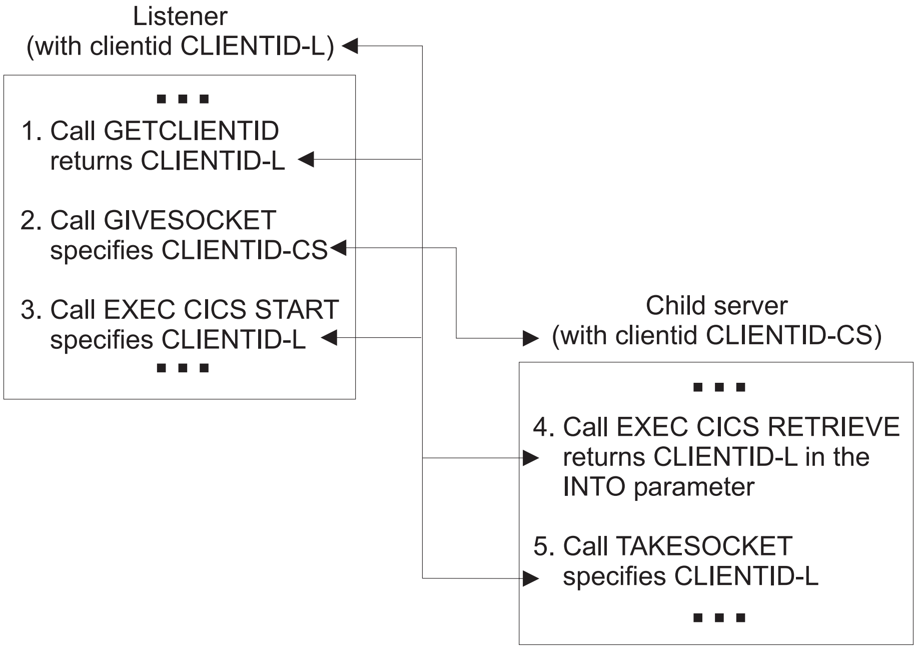


Figure 119. Transfer of CLIENTID Information

Figure 119 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.
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 the 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. It 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.”
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

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.

```
►►──tran──,──┬────────────────┬──,──┬────┬──,──┬────────┬──────────────────────►◄
             └─client-in-data─┘     ├─IC─┤     └─hhmmss─┘
                                    ├─ic─┤
                                    ├─TD─┤
                                    └─td─┘
```

tran
The CICS transaction ID (in uppercase) that the Listener is going to start. This field can be 1 to 4 characters.

client-in-data
Optional. Application data, used by the optional security exit [3] or the server transaction. The maximum length of this field is 35 characters.

IC/TD
Optional. Startup type that can be either `IC` for CICS interval control or `TD` for CICS transient data. These can also be entered in lowercase (`ic` or `td`). If this field is left blank, startup is immediate.

[3] (See “Writing Your Own Security Link Module for the Listener” on page 470)

hhmmss

Optional. Hours, minutes, and seconds for interval time if the transaction is started using interval control. All 6 digits must be given.

Note: TD ignores the timefield.

Examples

The following are examples of client input and the Listener processing that results from them. The data fields referenced can be found in "Listener Output Format." Note that 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 described by the structure TCPSOCKET-PARM, described in "Listener Output Format." The data 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 very frequently and for which the overhead of initiating a separate CICS transaction for each server request could be a performance concern.
`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 passed to the new transaction.

Listener Output Format

Table 16 on page 470 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 16. Listener Output Format

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

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 Figure 95 on page 429 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 17 on page 471 shows the data area used by the security module.

Table 17. Security Exit Data

Description	Format	Value
CICS transaction identifier	4-byte character	CICS transaction requested by the client
Data area	40-byte character	User data received from the client
Action	2-byte character	Method of starting the task: **IC** Interval control **KC** Task control **TD** Transient data
Interval control time	6-byte character	Interval requested for IC start Has the form *hhmmss*
Address family	Halfword binary	Network address family. A value of 2 must be set.
Port	Halfword binary	The port number of the requester's port.
Address	Fullword binary	The IP address of the requester's host.
Switch	1-byte character	Switch: **1** Permit the transaction **Not 1** Prohibit the transaction
Switch-2	1-byte character	Switch: **1** Listener sends message to Client. **Not 1** Security Exit program sends message to client.
Terminal identification	4-byte character	LOW-VALUES and binary zeros if a CICS terminal is not associated with the new task. CICS terminal identifier if a CICS terminal is associated with the new task.
Socket descriptor	Halfword binary	Current socket descriptor
User ID	8-byte character	A USERID value which is used in starting the server transaction.

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 of these routines, refer to EZACIC04, EZACIC05, and EZACIC06 in Figure 120 on page 483.

Part 4. Appendixes

Appendix A. Examples

Autonomous FTP

Overview

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, here 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=FTPAUTNP,CLASS=X,DISP=D
// JOB  FTPAUTNP   TEST   AUTONOMIOUS FTP BATCH
// DLBL TESTNKD,'%A.KRUS',0,VSAM,CAT=ESCAT1,RECSIZE=120,              X
               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
/*
```

```
*
// EXEC FTP,PARM='IP=KRUSE'
KRUS
DAGI
DD
DD
LCD ESCAT1
CD VSE230/TEMP
PUT %TEST,SAM,F,120 FTPPUT.X1
QUIT
/*
IF $RC > 0 THEN
GOTO $EOJ
// EXEC IDCAMS,SIZE=AUTO
 DELETE (%A.KRUS                                  ) -
     CLUSTER        -
     PURGE -
     CATALOG (ESCAT1.USER.CATALOG                          )
/*
/&
* $$ EOJ
```

Notes:

1. In the job listing the workfile has a dynamic name, here

   ```
   PUT %X1SAM,SAM,F,120 FTPPUT.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('LPRTEST') 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.

```
* $$ JOB JNM=CATAL,CLASS=A,DISP=D
// JOB CATAL  CATALOG SCRIPT MEMBER PRTLOCAL.L
// EXEC LIBR
ACC S=PRD2.CONFIG
CAT PRTLOCAL.L  R=Y
SET HOST=REMHOST         SYMBOLIC HOST NAME
SET PRINTER=PRINTER1
/+
/*
/&
* $$ EOJ
```

GPS and RCF

Overview

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*.

Defining to VTAM

```
TCPPRT   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 please refer to the manual *VTAM Programming Guide*.

Defining to CICS

```
CEDA DEFine TYpeterm:  GPSPRT   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 instead simply 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*.

Step 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, let's take 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.

Step 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.

Step 3: Using the catalog

Now that you have the VSE and TCP/IP systems know about the catalog (and if it actually exist!), 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 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

Overview

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.

Notes:

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.

A detailed description of the TCP/IP for VSE/ESA preprocessor can be found in *TCP/IP for VSE 1.5 Programmer's Reference*.

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 RANKIN.
    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).
```

Figure 120. COBOL Example (Part 1 of 3)

```
 PROCEDURE DIVISION.

 BEGIN.
*---------------------------------------*
*                                       *
*            First Test                 *
*                                       *
*---------------------------------------*
*
*    Setup IPADDRESS to hold 172.20.10.10 in binary
*
     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'.
```

Figure 120. COBOL Example (Part 2 of 3)

```
*---------------------------------------*
*                                       *
*            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.
```

Figure 120. COBOL Example (Part 3 of 3)

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)
                     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);
```

Figure 121. PL/I Example (Part 1 of 2)

```
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;
```

Figure 121. PL/I Example (Part 2 of 2)

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 Figure 120 on page 483.

Compiling a COBOL Program for Batch

The first example shows how to compile source COBSRC.C (see Figure 120 on page 483) 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
* $$ PUN CLASS=4,DISP=I
// JOB COMPILE TCPIP PROGRAM
// LIBDEF *,SEARCH=(PRD3.INGO,PRD1.BASE,PRD2.PROD,PRD2.SCEEBASE)
// EXEC PROC=COMSTP1
/&
* $$ 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
$ $$ PUN CLASS=4,DISP=I
// JOB CATAL1 CATALOG OUTPUT OF THE TCPIP preprocessor
// LIBDEF *,SEARCH=(PRD3.INGO,PRD1.BASE,PRD2.PROD,PRD2.SCEEBASE)
// LIBDEF *,CATALOG=PRD3.INGO
// EXEC LIBR
   ACC S=PRD3.INGO
   CATALOG PRETCP.DAT  EOD=/(    REPLACE=YES
* $$ END
// OPTION DECK
*
* Process EXEC TCP source for CICS
*
// EXEC IPNETPRE,SIZE=IPNETPRE,PARM='LANG=COBOL,ENV=BATCH'
* $$ SLI MEM=COBSRC.C,S=PRD3.INGO
/*
// EXEC IESINSRT
/(
/*
// EXEC PROC=COMSTP2
#&
$ $$ EOJ
* $$ 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 Figure 120 on page 483) 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
* $$ PUN CLASS=4,DISP=I
// JOB COMPILE TCPIP PROGRAM
// LIBDEF *,SEARCH=(PRD3.INGO,PRD1.BASE,PRD2.PROD,PRD2.SCEEBASE)
// EXEC PROC=COMSTP1
/&
* $$ 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
$ $$ PUN CLASS=4,DISP=I
// JOB CATAL1 CATALOG OUTPUT OF THE TCPIP preprocessor
// LIBDEF *,SEARCH=(PRD3.INGO,PRD1.BASE,PRD2.PROD,PRD2.SCEEBASE)
// LIBDEF *,CATALOG=PRD3.INGO
// EXEC LIBR
   ACC S=PRD3.INGO
   CATALOG PRETCP.DAT  EOD=/(    REPLACE=YES
* $$ 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
$ $$ PUN CLASS=4,DISP=I
// JOB CATALOG OUTPUT OF THE CICS preprocessor
// LIBDEF *,SEARCH=(PRD3.INGO,PRD1.BASE,PRD2.PROD,PRD2.SCEEBASE)
// LIBDEF *,CATALOG=PRD3.INGO
// EXEC LIBR
   ACC S=PRD3.INGO
   CATALOG PRECICS.DAT  EOD=/(    REPLACE=YES
* $$ 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
*
// 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 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.

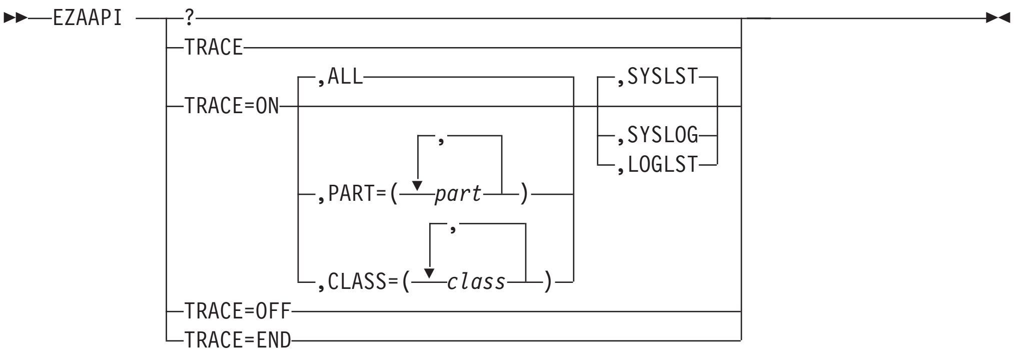
For short, this trace facility is named "EZAAPI trace" in the following.

Requirements for Usage

The EZAAPI trace needs module EZASOHTR loaded into the SVA (which is the default with z/VSE).

Setup

The EZAAPI trace can be activated and controlled with the AR command EZAAPI.



EZAAPI ?
Display the command syntax

EZAAPI TRACE
Display currrent 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

PART=(part,..)
Define and start trace for selected partitions

CLASS=(class,..)
Define and start trace for selected dynamic classes

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 EZASOH99 (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
   ```

With the activation of the EZAAPI trace, the BSD-C trace of TCP/IP for VSE/ESA (called $SOCKDBG trace) is automatically activated in addition.

Index

Readers' Comments — We'd Like to Hear from You

TCP/IP for VSE/ESA
TCP/IP for VSE/ESA – IBM Program Setup and Supplementary Information

Publication No. SC33-6601-11

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: FAX (Germany): 07031+16-3456
  FAX (Other Countries): (+49)+7031-16-3456
- Send your comments via e-mail to: s390id@de.ibm.com

If you would like a response from IBM, please fill in the following information:

Name

Address

Company or Organization

Phone No.

E-mail address

Readers' Comments — We'd Like to Hear from You
SC33-6601-11

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Deutschland Research & Development GmbH
Department 3248
Schoenaicher Strasse 220
D-71032 Boeblingen
Federal Republic of Germany

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

Program Number: 5686-A04

Printed in USA

SC33-6601-11