z/OS Communications Server

IP IMS Sockets Guide

Version 1 Release 2

SC31-8830-00

z/OS Communications Server

IP IMS Sockets Guide

Version 1 Release 2

SC31-8830-00

Note

Before using this information and the product it supports, be sure to read the general information under “Appendix D. Notices” on page 203.

First Edition (October 2001)

This edition applies to Version 1 Release 2 of z/OS (program number 5694-A01) and to all subsequent releases and modifications until otherwise indicated in new editions.

Publications are not stocked at the address given below. If you want more IBM publications, ask your IBM representative or write to the IBM branch office serving your locality.

A form for your comments is provided at the back of this document. If the form has been removed, you may address comments to:

IBM Corporation
Software Reengineering
Department G71A/ Bldg 503
Research Triangle Park, North Carolina 27709–9990
U.S.A.

If you prefer to send comments electronically, use one of the following methods:

Fax (USA and Canada):
1-800-227-5088

Internet e-mail:
usib2hpd@vnet.ibm.com

World Wide Web:
http://www.ibm.com/servers/eserver/zseries/zos

IBMLink:
CIBMORCF at RALVM17

IBM Mail Exchange:
tkinlaw@us.ibm.com

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

Contents

Figures

Tables

About This Book

This book describes how to use IP Services with IMS Version 7 and above. It describes the IMS call interface and the supporting functions.

This book addresses the following topics:
- IMS client/server application design
- The IMS Listener
- The IMS Assist function
- The IMS socket calls, including call syntax conventions

This book is intended for programmers who have some familiarity with IMS Transaction Manager and IP Services, and who need to develop IMS client/server applications.

To ensure proper interprogram communication, the two halves of a client/server program must be developed together. At a minimum, they must agree on protocol and data formats. To complicate matters (particularly in the case of a UNIX processor talking to an IMS mainframe), the technology differences are so extensive that the two halves will often be coded by different individuals — one a IP socket programmer; the other, an IMS programmer.

This book has been designed to be read by users with a variety of backgrounds and needs:
- Application designers need to know how the various components of IMS TCP/IP interact to provide program-to-program communication. These readers should read “Chapter 3. Principles of Operation” on page 27.
- Experienced IP socket programmers need to know the protocol and message formats necessary to establish communication with the IMS Listener and with the server program. These readers should read “Chapter 4. How to Write an IMS TCP/IP Client Program” on page 39 and “Chapter 7. Using the CALL Instruction Application Programming Interface (API)” on page 61.
- Experienced IMS application programmers will be familiar with IMS input/output calls (GU, GN, ISRT). These programmers have two choices:
  - Programmers with IMS experience and little or no TCP/IP programming experience will probably wish to use the IMS Assist module, which accepts standard IMS I/O calls, and converts them to equivalent socket calls. They should read the chapter on implicit-mode programming.
  - IMS programmers with socket experience can chose to code native C language or use the Sockets Extended API. These programmers should read the chapter on explicit-mode programming and “Chapter 7. Using the CALL Instruction Application Programming Interface (API)” on page 61.
- IMS system programmers and communication programmers are responsible for the IMS system itself. These readers should read “Chapter 6. How to Customize and Operate the IMS Listener” on page 55.

Typographic Conventions Used in This Book

This publication uses the following typographic conventions:
- Commands that you enter verbatim onto the command line are presented in **bold**.
- Variable information and parameters that you enter within commands, such as filenames, are presented in *italic*.

- System responses are presented in `monospace`.

Where to Find More Information

This section contains:

- Pointers to information available on the Internet
- Information about licensed documentation
- Information about LookAt, the online message tool
- A set of tables that describes the books in the z/OS Communications Server (z/OS CS) library, along with related publications

Where to Find Related Information on the Internet

Home Page	Web address
z/OS	http://www.ibm.com/servers/eserver/zseries/zos/
z/OS Internet Library	http://www.ibm.com/servers/eserver/zseries/zos/bkserv/
IBM Communications Server product	http://www.software.ibm.com/network/commserver/
IBM Communications Server support	http://www.software.ibm.com/network/commserver/support/
IBM Systems Center publications	http://www.redbooks.ibm.com/
IBM Systems Center flashes	http://www-1.ibm.com/support/techdocs/atsmastr.nsf
VTAM and TCP/IP	http://www.software.ibm.com/network/commserver/about/csos390.html
IBM	http://www.ibm.com
RFC	http://www.ietf.org/rfc.html

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

DNS Web Sites

For information about DNS, see the following Web sites:

USENET news groups:

comp.protocols.dns.bind

For BIND mailing lists, see:

- http://www.isc.org/ml-archives/
  - BIND Users
    - Subscribe by sending mail to bind-users-request@isc.org
    - Submit questions or answers to this forum by sending mail to bind-users@isc.org
  - BIND 9 Users (Note: This list may not be maintained indefinitely.)
    - Subscribe by sending mail to bind9-users-request@isc.org
    - Submit questions or answers to this forum by sending mail to bind9-users@isc.org

For definitions of the terms and abbreviations used in this book, you can view or download the latest *IBM Glossary of Computing Terms* at the following Web address:

http://www.ibm.com/ibm/terminology

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

Licensed Documents

z/OS Communications Server licensed documentation in PDF format is available on the Internet at the IBM Resource Link Web site at http://www.ibm.com/servers/resourcelink. Licensed books are available only to customers with a z/OS Communications Server license. Access to these books requires an IBM Resource Link Web user ID and password, and a key code. With your z/OS Communications Server order, you received a memo that includes this key code. To obtain your IBM Resource Link Web user ID and password, log on to http://www.ibm.com/servers/resourcelink. To register for access to the z/OS licensed books perform the following steps:

1. Log on to Resource Link using your Resource Link user ID and password.
2. Click on **User Profiles** located on the left-hand navigation bar.
3. Click on **Access Profile**.
4. Click on **Request Access to Licensed books.**
5. Supply your key code where requested and click on the **Submit** button.

If you supplied the correct key code, you will receive confirmation that your request is being processed. After your request is processed, you will receive an e-mail confirmation.

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

1. Log on to Resource Link using your Resource Link user ID and password.
2. Click on **Library.**
3. Click on **zSeries**.
4. Click on **Software**.
5. Click on **z/OS Communications Server**.
6. Access the licensed book by selecting the appropriate element.

LookAt, an Online Message Help Facility

LookAt is an online facility that allows you to look up explanations for z/OS CS messages and system abends.

Using LookAt to find information is faster than a conventional search because LookAt goes directly to the explanation.

LookAt can be accessed from the Internet or from a TSO command line.

To use LookAt as a TSO command, LookAt must be installed on your host system. You can obtain the LookAt code for TSO from the LookAt Web site by clicking on **News and Help** or from the z/OS V1R2 Collection, SK3T-4269.

To find a message explanation from a TSO command line, simply enter **lookat+message ID**, as in the following example:

```
lookat ezz8477i
```

This results in direct access to the message explanation for message EZZ8477I.

You can use LookAt on the Internet at the following Web site:
www.ibm.com/servers/eserver/zseries/zos/bkserv/lookat/lookat.html

To find a message explanation from the LookAt Web site, simply enter the message ID. You can select the release, if applicable.

How to Contact IBM® Service

For telephone assistance in problem diagnosis and resolution (in the United States or Puerto Rico), call the IBM Software Support Center anytime (1-800-237-5511). You will receive a return call within 8 business hours (Monday – Friday, 8:00 a.m. – 5:00 p.m., local customer time).

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

z/OS Communications Server Information

This section contains descriptions of the books in the z/OS Communications Server library.

z/OS Communications Server publications are available:

- Online at the z/OS Internet Library web page at http://www.ibm.com/servers/eserver/zseries/zos/
- In hardcopy and softcopy
- In softcopy only

Softcopy Information

Softcopy publications are available in the following collections:

Titles	Order Number	Description
z/OS V1R2 Collection	SK3T-4269	This is the CD collection shipped with the z/OS product. It includes the libraries for z/OS V1R2, in both BookManager and PDF formats.
z/OS Software Products Collection	SK3T-4270	This CD includes, in both BookManager and PDF formats, the libraries of z/OS software products that run on z/OS but are not elements and features, as well as the *Getting Started with Parallel Sysplex* bookshelf.
z/OS V1R2 and Software Products DVD Collection	SK3T-4271	This collection includes the libraries of z/OS (the element and feature libraries) and the libraries for z/OS software products in both BookManager and PDF format. This collection combines SK3T-4269 and SK3T-4270.
z/OS Licensed Product Library	SK3T-4307	This CD includes the licensed books in both BookManager and PDF format.
System Center Publication IBM S/390 Redbooks Collection	SK2T-2177	This collection contains over 300 ITSO redbooks that apply to the S/390 platform and to host networking arranged into subject bookshelves.

z/OS Communications Server Library

The following abbreviations follow each order number in the tables below.

HC/SC — Both hardcopy and softcopy are available.

SC — Only softcopy is available. These books are available on the CD Rom accompanying z/OS (SK3T-4269 or SK3T-4307). Unlicensed books can be viewed at the z/OS Internet library site.

Updates to books are available on RETAIN and in the document called *OS/390 DOC APARs and ++HOLD DOC data* which can be found at http://www.s390.ibm.com/os390/bkserv/ new_tech_info.html. See “Appendix C. Information Apars” on page 201 for a list of the books and the informational apars (info apars) associated with them.

Planning and Migration:

Title	Number	Format	Description
z/OS Communications Server: SNA Migration	GC31-8774	HC/SC	This book is intended to help you plan for SNA, whether you are migrating from a previous version or installing SNA for the first time. This book also identifies the optional and required modifications needed to enable you to use the enhanced functions provided with SNA.
z/OS Communications Server: IP Migration	GC31-8773	HC/SC	This book is intended to help you plan for TCP/IP Services, whether you are migrating from a previous version or installing IP for the first time. This book also identifies the optional and required modifications needed to enable you to use the enhanced functions provided with TCP/IP Services.

Resource Definition, Configuration, and Tuning:

Title	Number	Format	Description
z/OS Communications Server: IP Configuration Guide	SC31-8775	HC/SC	This book describes the major concepts involved in understanding and configuring an IP network. Familiarity with the z/OS operating system, IP protocols, z/OS UNIX System Services, and IBM Time Sharing Option (TSO) is recommended. Use this book in conjunction with the *z/OS Communications Server: IP Configuration Reference*.
z/OS Communications Server: IP Configuration Reference	SC31-8776	HC/SC	This book presents information for people who want to administer and maintain IP. Use this book in conjunction with the *z/OS Communications Server: IP Configuration Guide*. The information in this book includes: • TCP/IP configuration data sets • Configuration statements • Translation tables • SMF records • Protocol number and port assignments
z/OS Communications Server: SNA Network Implementation Guide	SC31-8777	HC/SC	This book presents the major concepts involved in implementing an SNA network. Use this book in conjunction with the *z/OS Communications Server: SNA Resource Definition Reference*.
z/OS Communications Server: SNA Resource Definition Reference	SC31-8778	HC/SC	This book describes each SNA definition statement, start option, and macroinstruction for user tables. It also describes NCP definition statements that affect SNA.Use this book in conjunction with the *z/OS Communications Server: SNA Network Implementation Guide*.

Title	Number	Format	Description
z/OS Communications Server: SNA Resource Definition Samples	SC31-8836	SC	This book contains sample definitions to help you implement SNA functions in your networks, and includes sample major node definitions.
z/OS Communications Server: AnyNet SNA over TCP/IP	SC31-8832	SC	This guide provides information to help you install, configure, use, and diagnose SNA over TCP/IP.
z/OS Communications: Server AnyNet Sockets over SNA	SC31-8831	SC	This guide provides information to help you install, configure, use, and diagnose sockets over SNA. It also provides information to help you prepare application programs to use sockets over SNA.

Operation:

Title	Number	Format	Description
z/OS Communications Server: IP User's Guide and Commands	SC31-8780	HC/SC	This book describes how to use TCP/IP applications. It contains requests that allow a user to: log on to a remote host using Telnet, transfer data sets using FTP, send and receive electronic mail, print on remote printers, and authenticate network users.
z/OS Communications Server: IP System Administrator's Commands	SC31-8781	HC/SC	This book describes the functions and commands helpful in configuring or monitoring your system. It contains system administrator's commands, such as NETSTAT, PING, TRACERTE and their UNIX counterparts. It also includes TSO and MVS commands commonly used during the IP configuration process.
z/OS Communications Server: SNA Operation	SC31-8779	HC/SC	This book serves as a reference for programmers and operators requiring detailed information about specific operator commands.
z/OS Communications Server: Operations Quick Reference	SX75-0124	HC/SC	This book contains essential information about SNA and IP commands.

Customization:

Title	Number	Format	Description
z/OS Commmunications Server: SNA Customization	LY43-0092	SC	This book enables you to customize SNA, and includes the following: • Communication network management (CNM) routing table • Logon-interpret routine requirements • Logon manager installation-wide exit routine for the CLU search exit • TSO/SNA installation-wide exit routines • SNA installation-wide exit routines
z/OS Communications Server: IP Network Print Facility	SC31-8833	SC	This book is for system programmers and network administrators who need to prepare their network to route SNA, JES2, or JES3 printer output to remote printers using TCP/IP Services.

Writing Application Programs:

Title	Number	Format	Description
z/OS Communications Server: IP Application Programming Interface Guide	SC31-8788	SC	This book describes the syntax and semantics of program source code necessary to write your own application programming interface (API) into TCP/IP. You can use this interface as the communication base for writing your own client or server application. You can also use this book to adapt your existing applications to communicate with each other using sockets over TCP/IP.
z/OS Communications Server: IP CICS Sockets Guide	SC31-8807	SC	This book is for people who want to set up, write application programs for, and diagnose problems with the socket interface for CICS using z/OS TCP/IP.
z/OS Communications Server: IP IMS Sockets Guide	SC31-8830	SC	This book is for programmers who want application programs that use the IMS TCP/IP application development services provided by IBM's TCP/IP Services.
z/OS Communications Server: IP Programmer's Reference	SC31-8787	SC	This book describes the syntax and semantics of a set of high-level application functions that you can use to program your own applications in a TCP/IP environment. These functions provide support for application facilities, such as user authentication, distributed databases, distributed processing, network management, and device sharing. Familiarity with the z/OS operating system, TCP/IP protocols, and IBM Time Sharing Option (TSO) is recommended.
z/OS Communications Server: SNA Programming	SC31-8829	SC	This book describes how to use SNA macroinstructions to send data to and receive data from (1) a terminal in either the same or a different domain, or (2) another application program in either the same or a different domain.
z/OS Communications Server: SNA Programmers LU 6.2 Guide	SC31-8811	SC	This book describes how to use the SNA LU 6.2 application programming interface for host application programs. This book applies to programs that use only LU 6.2 sessions or that use LU 6.2 sessions along with other session types. (Only LU 6.2 sessions are covered in this book.)
z/OS Communications Server: SNA Programmers LU 6.2 Reference	SC31-8810	SC	This book provides reference material for the SNA LU 6.2 programming interface for host application programs.
z/OS Communications Server: CSM Guide	SC31-8808	SC	This book describes how applications use the communications storage manager.
z/OS Communications Server: CMIP Services and Topology Agent Guide	SC31-8828	SC	This book describes the Common Management Information Protocol (CMIP) programming interface for application programmers to use in coding CMIP application programs. The book provides guide and reference information about CMIP services and the SNA topology agent.

Diagnosis:

Title	Number	Format	Description
z/OS Communications Server: IP Diagnosis	GC31-8782	HC/SC	This book explains how to diagnose TCP/IP problems and how to determine whether a specific problem is in the TCP/IP product code. It explains how to gather information for and describe problems to the IBM Software Support Center.
z/OS Communications Server: SNA Diagnosis V1 Techniques and Procedures and *z/OS Communications Server: SNA Diagnosis V2 FFST Dumps and the VIT*	LY43-0088 LY43-0089	HC/SC	These books help you identify an SNA problem, classify it, and collect information about it before you call the IBM Support Center. The information collected includes traces, dumps, and other problem documentation.
z/OS Communications Server: SNA Data Areas Volume 1 and *z/OS Communications Server: SNA Data Areas Volume 2*	LY43-0090 LY43-0091	SC	These books describe SNA data areas and can be used to read an SNA dump. They are intended for IBM programming service representatives and customer personnel who are diagnosing problems with SNA.

Messages and Codes:

Title	Number	Format	Description
z/OS Communications Server: SNA Messages	SC31-8790	HC/SC	This book describes the ELM, IKT, IST, ISU, IUT, IVT, and USS messages. Other information in this book includes: • Command and RU types in SNA messages • Node and ID types in SNA messages • Supplemental message-related information
z/OS Communications Server: IP Messages Volume 1 (EZA)	SC31-8783	HC/SC	This volume contains TCP/IP messages beginning with EZA.
z/OS Communications Server: IP Messages Volume 2 (EZB)	SC31-8784	HC/SC	This volume contains TCP/IP messages beginning with EZB.
z/OS Communications Server: IP Messages Volume 3 (EZY)	SC31-8785	HC/SC	This volume contains TCP/IP messages beginning with EZY.
z/OS Communications Server: IP Messages Volume 4 (EZZ-SNM)	SC31-8786	HC/SC	This volume contains TCP/IP messages beginning with EZZ and SNM.
z/OS Communications Server: IP and SNA Codes	SC31-8791	HC/SC	This book describes codes and other information that appear in z/OS Communications Server messages.

APPC Application Suite:

Title	Number	Format	Description
z/OS Communications Server: APPC Application Suite User's Guide	GC31-8809	SC	This book documents the end-user interface (concepts, commands, and messages) for the AFTP, ANAME, and APING facilities of the APPC application suite. Although its primary audience is the end user, administrators and application programmers may also find it useful.

Title	Number	Format	Description
z/OS Communications Server APPC Application Suite Administration	SC31-8835	SC	This book contains the information that administrators need to configure the APPC application suite and to manage the APING, ANAME, AFTP, and A3270 servers.
z/OS Communications Serverz: APPC Application Suite Programming	SC31-8834	SC	This book provides the information application programmers need to add the functions of the AFTP and ANAME APIs to their application programs.

Redbooks

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

Title	Number
TCP/IP Tutorial and Technical Overview	GG24–3376
SNA and TCP/IP Integration	SG24–5291
IBM Communication Server for OS/390 V2R10 TCP/IP Implementation Guide: Volume 1: Configuration and Routing	SG24–5227
IBM Communication Server for OS/390 V2R10 TCP/IP Implementation Guide: Volume 2: UNIX Applications	SG24–5228
IBM Communication Server for OS/390 V2R10 TCP/IP Implementation Guide: Volume 3: MVS Applications	SG24–5229
OS/390 Secureway Communication Server V2R8 TCP/IP Guide to Enhancements	SG24–5631
TCP/IP in a Sysplex	SG24–5235
Managing OS/390 TCP/IP with SNMP	SG24–5866
Security in OS/390–based TCP/IP Networks	SG24–5383
IP Network Design Guide	SG24–2580

Related Information

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

The table below lists books that may be helpful to readers.

Title	Number
z/OS SecureWay Security Server Firewall Technologies	SC24-5922
S/390: OSA-Express Customer's Guide and Reference	SA22-7403
z/OS MVS Diagnosis: Procedures	GA22-7587
z/OS MVS Diagnosis: Reference	GA22-7588
z/OS MVS Diagnosis: Tools and Service Aids	GA22-7589

Determining If a Publication Is Current

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

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

Summary of Changes

Summary of Changes
for SC31-8830-00
z/OS Version 1 Release 2

This book contains information previously presented in *OS/390 V2R5 eNetwork Communications Server: IP IMS Sockets Guide*, SC31-8519.

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

Part 1. IMS Overview

Chapter 1. Using TCP/IP with IMS

This chapter includes a discussion of the kind of applications for which IMS TCP/IP is intended and an overview of its components.

The Role of IMS TCP/IP

The IMS/ESA database and transaction management facility is used throughout the world. For many enterprises, IMS is the data processing backbone, supporting large personnel and financial databases, manufacturing control files, and inventory management facilities. IMS backup and recovery features protect valuable data assets, and the IMS Transaction Manager provides high-speed access for thousands of concurrent users.

Traditionally, many IMS users have used 3270-type protocol to communicate with the IMS Transaction Manager. In that environment, all of the processing, including display screen formatting, is done by the IMS mainframe. During the decade of the 1980s, users began to move some of the processing outboard into personal computers. However, these PCs were typically connected to IMS via SNA 3270 protocol.

During that period, although most IMS users were focused on 3270 PC emulation, many non-IMS users were busy building a network based on a different protocol, called TCP/IP. As this trend developed, the need for an access path between TCP/IP-communicating devices and the still-indispensable processing power of IMS became clear. IMS TCP/IP provides that access path. Its role can be more easily understood when one distinguishes between traditional 3270 applications (in which the IMS processor does all the work), and the more complex client/server applications (in which the application logic is divided between the IMS processor and another programmable device such as a TCP/IP host).

MVS TCP/IP supports both application types:

- When a TCP/IP host needs access to a traditional 3270 Message Format Service (MFS) application, it does not need to use the IMS TCP/IP feature; it can connect to IMS directly through Telnet which provides 3270 emulation services for TCP/IP-connected clients. Telnet is a part of the base TCP/IP Services product. (See *z/OS Communications Server: IP User's Guide and Commands* for more information).
- When a TCP/IP host needs to support a client/server application, it should use the IMS TCP/IP feature of TCP/IP Services. This feature is specifically designed to support two-way client/server communication between an IMS message processing program (MPP) and a TCP/IP host.

As used in this book, the term *client* refers to a program that requests services of another program. That other program is known as the *server*. The client is often a UNIX-based program; however, DOS-, OS/2*-, CMS-, and MVS-based programs can also act as clients. Similarly, as used in this book, the term *server* refers to a program that is often an IMS MPP; however, the server can be a TCP/IP host, responding to an IMS MPP client.

Introduction to IMS TCP/IP

For peer-to-peer applications that use SNA communication facilities, remote programmable devices communicate with IMS through the advanced program-to-program communication (APPC) API. For peer-to-peer applications that use TCP/IP communication facilities, remote programmable devices communicate with IMS through facilities provided by IMS TCP/IP.

The IMS TCP/IP feature provides the services necessary to establish and maintain connection between a TCP/IP-connected host and an IMS MPP. In addition, it allows client/server applications to be developed using the TCP/IP socket application programming interface.

In operation, when a TCP/IP client requires program-to-program communication with an IMS server message processing program (MPP), the client sends its request to TCP/IP Services. TCP/IP passes the request to the IMS Listener, which schedules the requested MPP and transfers control of the connection to it. Once control of the connection is passed, data transfer between the server and the remote client is performed using socket calls.

IMS TCP/IP Feature Components

The IMS TCP/IP feature consists of the following components:

- The IMS Listener, which provides connectivity
- The IMS Assist module, which simplifies TCP/IP communications programming
- The Sockets Extended application programming interface (API) [1]

The IMS Listener

The purpose of the Listener is to provide clients with a single point of contact to IMS. The IMS Listener is a batch program (BMP) that waits for connection requests from remote TCP/IP-connected hosts. When a request arrives, the Listener schedules the appropriate transaction (the server) and passes a TCP/IP socket (representing the connection) to that server.

The IMS Listener maintains connection requests until the requested MPP takes control of the socket. The Listener is capable of maintaining a variable number of concurrent connection requests.

The IMS Assist Module

The Assist module is a subroutine that is a part of the server program. Its use is optional. Its purpose is to allow the use of conventional IMS calls for TCP/IP communication between client and server. In use, the Assist module intercepts the IMS calls and issues the corresponding socket commands; consequently, IMS MPP programmers who use the IMS Assist module require no TCP/IP skills.

Programs that do use the Assist module are known as *implicit-mode* programs because the socket calls are issued implicitly by the Assist module.

Programs that do not use the Assist module issue socket calls directly. Such programs are known as *explicit-mode* programs because of their explicit use of the calls.

1. Shipped with the TCP/IP V3R2 for MVS base product

The MVS TCP/IP Socket Application Programming Interface (Sockets Extended)

The socket call interface provides a set of programming calls that can be used in an IMS message processing program to conduct a conversation with a peer program in another TCP/IP processor. The interface is derived from BSD 4.3 socket, a commonly used communications programming interface in the TCP/IP environment. Socket calls include connection, initiation, and termination functions, as well as basic read/write communication. The MVS TCP/IP socket call interface makes it possible to issue socket calls from programs written in COBOL, PL/I, and assembler language.

The IMS socket calls are a subset of the TCP/IP socket calls. They are designed to be used in programs written in other than C language; hence the term Sockets Extended.

Chapter 2. Introduction to TCP/IP for IMS

This chapter presents an overview of TCP/IP as it is used with MVS.

What IMS TCP/IP Does

The **IMS TCP/IP** feature allows remote users to access IMS client/server applications over TCP/IP internets. It is a feature of TCP/IP Services. Figure 1 shows how IMS TCP/IP gives a variety of remote users peer-to-peer communication with IMS applications.

It is important to understand that IMS TCP/IP is primarily intended to support *peer-to-peer* applications, as opposed to the traditional IMS mainframe interactive applications in which the IMS system contained all programmable logic, and the remote terminal was often referred to as a "dumb" terminal. To connect a TCP/IP host to one of those traditional applications, you should first consider the use of Telnet, a function of TCP/IP Services which provides 3270 emulation. With Telnet, you can access existing 3270-style Message Format Services applications without modification. You should consider IMS TCP/IP only when developing new peer-to-peer applications in which both ends of the connection are programmable.

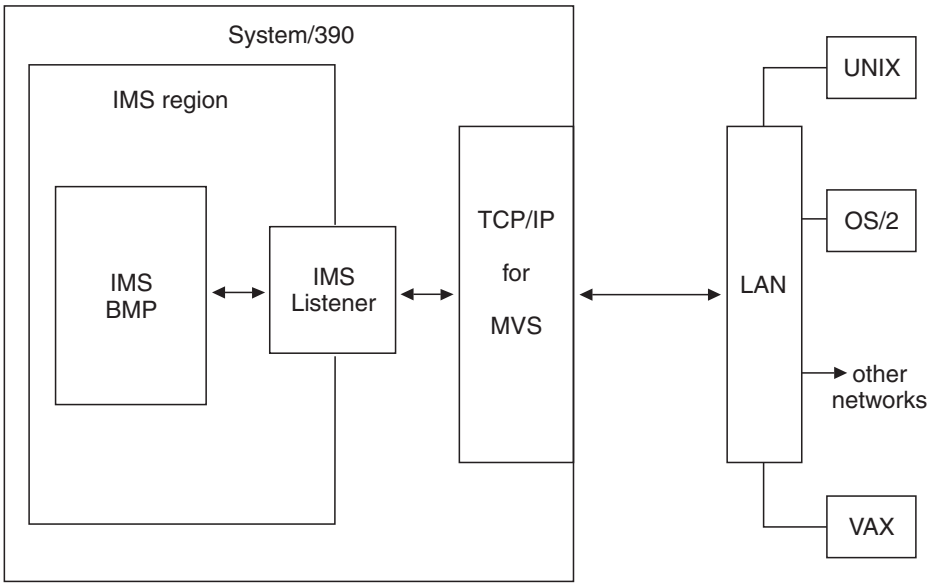


Figure 1. The Use of TCP/IP with IMS

IMS TCP/IP provides a variant of the BSD 4.3 Socket interface, which is widely used in TCP/IP networks and is based on the UNIX** system and other operating systems. The socket interface consists of a set of calls that IMS application programs can use to set up connections, send and receive data, and perform general communication control functions. The programs can be written in COBOL, PL/I, assembler language, or C.

Using IMS with SNA or TCP/IP

IMS is an online transaction processing system. This means that application programs using IMS can handle large numbers of data transactions from large networks of computers and terminals.

Communication throughout these networks has often been based on the Systems Network Architecture (SNA) family of protocols. IMS TCP/IP offers IMS users an alternative to SNA — the TCP/IP family of protocols for those users whose native communications protocol is TCP/IP.

TCP/IP Internets

This section describes some of the basic ideas behind the TCP/IP family of protocols.

Like SNA, TCP/IP is a set of communication protocols used between physically separated computer systems. Unlike SNA and most other protocols, TCP/IP is not designed for a particular hardware technology. TCP/IP can be implemented on a wide variety of physical networks, and is specially designed for communicating between systems on different physical networks (local and wide area). This is called *internetworking*.

Mainframe Interactive Processing

TCP/IP Services supports traditional 3270 mainframe interactive (MFI) applications with an emulator function called Telnet (TN3270). For these applications, all program logic runs in the mainframe, and the remote host uses only that amount of logic necessary to provide basic communications services. Thus, if your requirement is simply to provide access from a remote TCP/IP host to existing IMS MFI applications, you should consider Telnet rather than IMS TCP/IP as the communications vehicle. Telnet 3270-emulation functions allow your TCP/IP host to communicate with traditional applications without modification.

Client/Server Processing

TCP/IP also supports *client/server* processing, where processes are either:

- **Servers** that provide a particular service and respond to requests for that service
- **Clients** that initiate the requests to the servers

With IMS TCP/IP, remote client systems can initiate communications with IMS and cause an IMS transaction to start. It is anticipated that this will be the most common mode of operation. (Alternatively, the remote system can act as a server with IMS initiating the conversation.)

TCP, UDP, and IP

TCP/IP is a family of protocols that is named after its two most important members. Figure 2 on page 9 shows the TCP/IP protocols used by IMS TCP/IP, in terms of the layered Open Systems Interconnection (OSI) model, which is widely used to describe data communication systems. For IMS users who might be more accustomed to SNA, the left side of Figure 2 shows the SNA layers, which correspond very closely to the OSI layers.

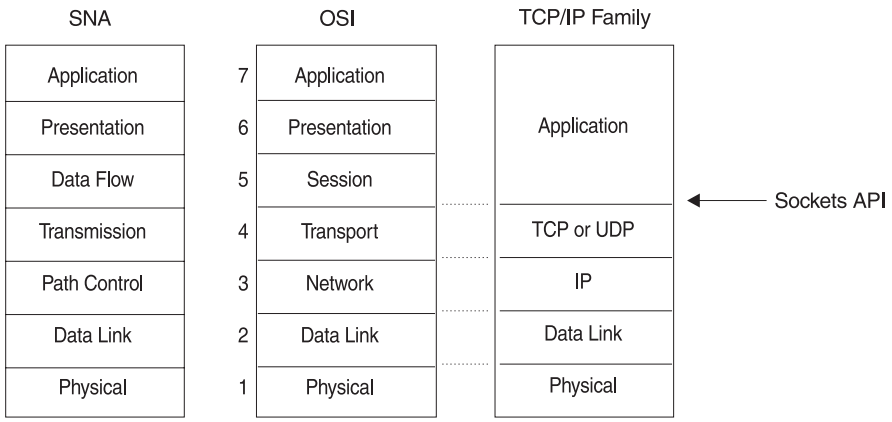


Figure 2. TCP/IP Protocols when compared to the OSI Model and SNA

The protocols implemented by TCP/IP Services and used by IMS TCP/IP, are highlighted in Figure 2:

Transmission Control Protocol (TCP)
: In terms of the OSI model, TCP is a transport-layer protocol. It provides a reliable virtual-circuit connection between applications; that is, a connection is established before data transmission begins. Data is sent without errors or duplication and is received in the same order as it is sent. No boundaries are imposed on the data; TCP treats the data as a stream of bytes.

User Datagram Protocol (UDP)
: UDP is also a transport-layer protocol and is an alternative to TCP. It provides an unreliable datagram connection between applications (that is, data is transmitted link by link; there is no end-to-end connection). The service provides no guarantees: data can be lost or duplicated, and datagrams can arrive out of order.

Internet Protocol (IP)
: In terms of the OSI model, IP is a network-layer protocol. It provides a datagram service between applications, supporting both TCP and UDP.

The Socket API

What the Socket API Provides

The socket API is a collection of socket calls that enable you to perform the following primary communication functions between application programs:

- Set up and establish connections to other users on the network
- Send and receive data to and from other users
- Close down connections

In addition to these basic functions, the API enables you to:

- Interrogate the network system to get names and status of relevant resources
- Perform system and control functions as required

IMS TCP/IP provides two TCP/IP socket application program interfaces (APIs), similar to those used on UNIX systems. One interfaces to C language programs, the other to COBOL, PL/I, and System/370* assembler language programs.

- **C language**. Historically, TCP/IP has been associated with the C language and the UNIX operating system. Textbook descriptions of socket calls are usually given in C, and most socket programmers are familiar with the C interface to TCP/IP. For these reasons, TCP/IP Services includes a C language API. If you are writing new TCP/IP applications and are familiar with C language

programming, you might prefer to use this interface. See *z/OS Communications Server: IP Application Programming Interface Guide* for the C language socket calls supported by MVS TCP/IP.

- **Sockets Extended API (COBOL, PL/I, Assembler Language)**. The Sockets Extended API (Sockets Extended) is for those who want to write in COBOL, PL/I, or assembler language, or who have COBOL, PL/I, or assembler language programs that need to be modified to run with TCP/IP. The Sockets Extended API enables you to do this by using CALL statements. If you are writing new TCP/IP applications in COBOL, PL/I, or assembler language, you might prefer to use the Sockets Extended API. With this interface, C language is not required. See "Chapter 7. Using the CALL Instruction Application Programming Interface (API)" on page 61 for details of this interface.

Programming with Sockets

The original UNIX socket interface was designed to hide the physical details of the network. It included the concept of a *socket*, which would represent the connection to the programmer, yet shield the program (as much as possible) from the details of communication programming. **A socket is an end-point for communication that can be named and addressed in a network.** From an application program perspective, a socket is a resource that is allocated by the TCP/IP address space. A socket is represented to the program by an integer called a *socket descriptor*.

Socket types

The MVS socket APIs provide a standard interface to the transport and internetwork layer interfaces of TCP/IP. They support three socket types: *stream, datagram, and raw*. Stream and datagram sockets interface to the transport layer protocols, and raw sockets interface to the network layer protocols. All three socket types are discussed here for background purposes.

Stream sockets transmit data between TCP/IP hosts that are already connected to one another. Data is transmitted in a continuous stream; in other words, there are no record length or newline character boundaries between data. Communicating processes [2] must agree on a scheme to ensure that both client and server have received all data. One way of doing this is for the sending process to send the *length* of the data, followed by the data itself. The receiving process reads the length and then loops, accepting data until all of it has been transferred.

In TCP/IP terminology, the stream socket interface defines a reliable connection-oriented service. In this context, the word *reliable* means that data is sent without error or duplication and is received in the same order as it is sent. Flow control is built in to avoid data overruns.

The **datagram** socket interface defines a connectionless service. Datagrams are sent as independent packets. The service provides no guarantees; data can be lost or duplicated, and datagrams can arrive out of order. The size of a datagram is limited to the size that can be sent in a single transaction (currently the default is 8192 and the maximum is 65507). No disassembly and reassembly of packets is performed by TCP/IP.

The **raw** socket interface allows direct access to lower layer protocols, such as IP and Internet Control Message Protocol (ICMP). This interface is often used for testing new protocol implementations.

2. In TCP/IP terminology, a *process* is essentially the same as an application program.

Addressing TCP/IP hosts

The following section describes how one TCP/IP host addresses another TCP/IP host. [3]

Address Families

An address family defines a specific addressing format. Applications that use the same addressing family have a common scheme for addressing socket end-points. TCP/IP for IMS supports the AF_INET address family.

Socket Addresses

A socket address in the AF_INET family comprises 4 fields: the name of the address family itself (AF_INET), a port, an internet address, and an eight-byte reserved field. In COBOL, a socket address looks like this:

```
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).
```

You will find this structure in every call that addresses another TCP/IP host.

In this structure, FAMILY is a half-word that defines which addressing family is being used. In IMS, FAMILY is always set to a value of 2, which specifies the AF_INET internet address family. [4] The PORT field identifies the application port number; it must be specified in network byte order. The IP_ADDRESS field is the internet address of the network interface used by the application. It also must be specified in network byte order. The RESERVED field should be set to all zeros.

Internet (IP) Addresses

An internet addresses (otherwise known as an IP address) is a 32-bit field that represents a network interface. An IP address is commonly represented in *dotted decimal* notation such as *129.5.25.1*. Every internet address within an administered AF_INET domain must be unique. A common misunderstanding is that a host must have only one internet address. In fact, a single host may have several internet addresses — one for each network interface.

Ports

A port is a 16-bit integer that defines a specific application, within an IP address, in which several applications use the same network interface. The port number is a qualifier that TCP/IP uses to route incoming data to a specific application within an IP address. Some port numbers are reserved for particular applications and are called *well-known ports*, such as Port 23, which is the well-known port for Telnet.

As an example, an MVS system with an IP address of 129.9.12.7 might have IMS as port 2000, and Telnet as port 23. In this example, a client desiring connection to IMS would issue a CONNECT call, requesting port 2000 at IP address 129.9.12.7.

Sockets and Ports:

Note: It is important to understand the difference between a socket and a port. TCP/IP defines a port to represent a certain process on a certain machine

3. In TCP/IP terminology, a host is simply a computer that is running TCP/IP. There is no connotation of ″mainframe″ or large processor within the TCP/IP definition of the word *host*.

4. Note that sockets support many address families, but TCP/IP for IMS only supports the internet address family.

(network interface). A port represents the location of one process in a host that can have many processes. A bound socket represents a specific port and the IP address of its host.

Domain Names

Because dotted decimal IP addresses are difficult to remember, TCP/IP also allows you to represent host interfaces on the network as alphabetic names, such as Alana.E04.IBM.COM, or CrFre@AOL.COM. Every Domain Name has an equivalent IP address or set of addresses. TCP/IP includes service functions (GETHOSTBYNAME and GETHOSTBYADDR) that will help you convert from one notation to another.

Network Byte Order

In the open environment of TCP/IP, internet addresses must be defined in terms of the architecture of the machines. Some machine architectures, such as IBM mainframes, define the lowest memory address to be the high-order bit, which is called *big endian*. However, other architectures, such as IBM PCs, define the lowest memory address to be the low-order bit, which is called *little endian*.

Network addresses in a given network must all follow a consistent addressing convention. This convention, known as Network Byte Order, defines the bit-order of network addresses as they pass through the network. The TCP/IP standard Network Byte Order is big-endian. In order to participate in a TCP/IP network, little-endian systems usually bear the burden of conversion to Network Byte Order.

Note: The socket interface does not handle application data bit-order differences. Application writers must handle these bit order differences themselves.

A Typical Client Server Program Flow Chart

Stream-oriented socket programs generally follow a prescribed sequence. See Figure 3 on page 13 for a diagram of the logic flow for a typical client and server. As you study this diagram, keep in mind the fact that a concurrent server typically starts before the client does, and waits for the client to request connection at step **3** . It then continues to wait for additional client requests after the client connection is closed.

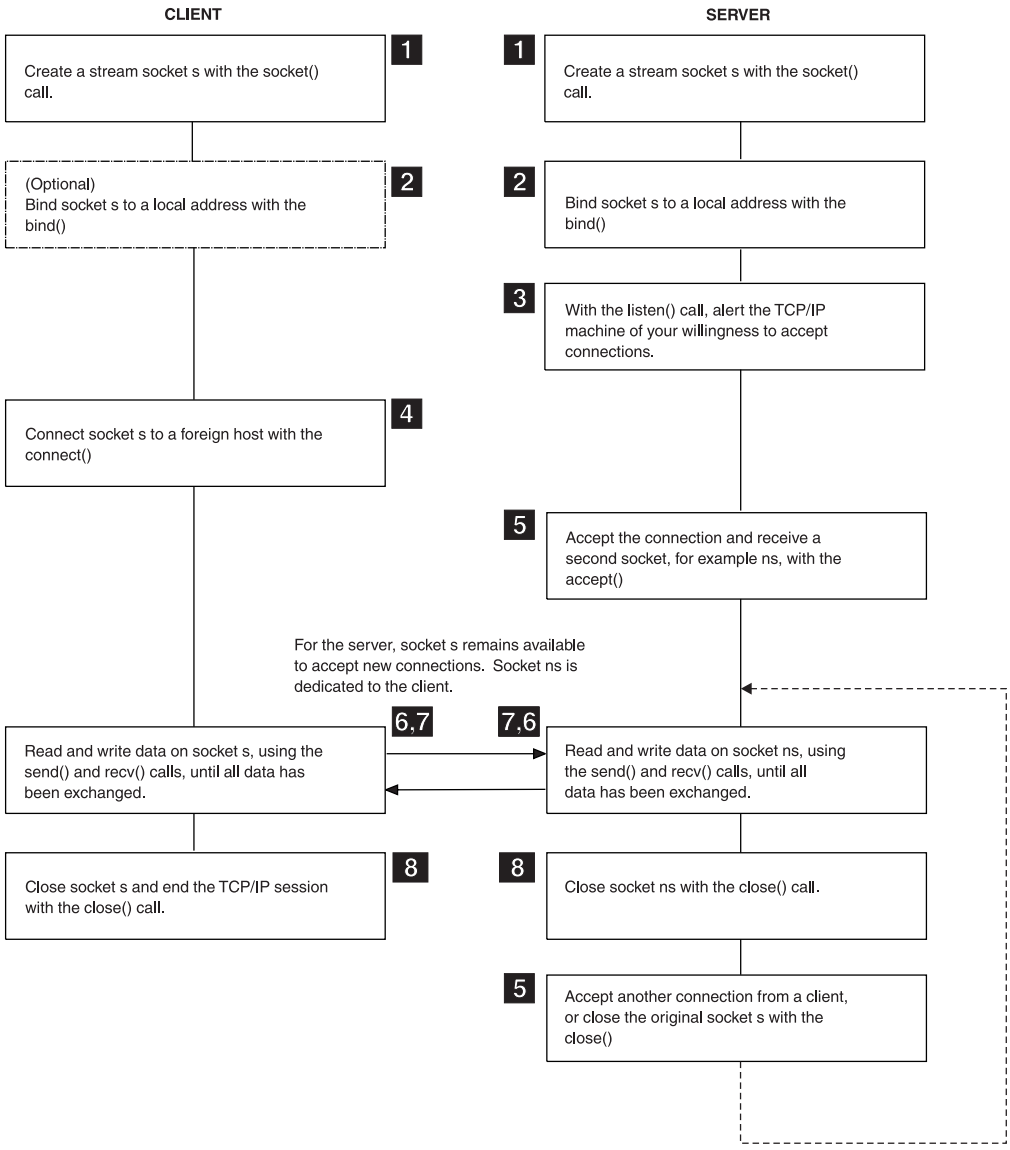


Figure 3. A Typical Client Server Session

Concurrent and Iterative Servers

An *iterative server* handles both the connection request and the transaction involved in the call itself. Iterative servers are fairly simple and are suitable for transactions that do not last long.

However, if the transaction takes more time, queues can build up quickly. In Figure 4 on page 14, once Client A starts a transaction with the server, Client B cannot make a call until A has finished.

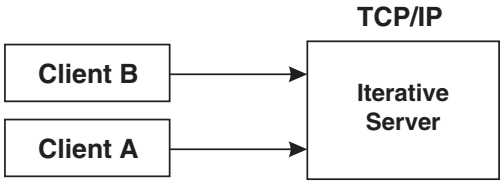


Figure 4. An Iterative Server

So, for lengthy transactions, a different sort of server is needed — the *concurrent server*, as shown in Figure 5. Here, Client A has already established a connection with the server, which has then created a *child server process* to handle the transaction. This allows the server to process Client B's request without waiting for A's transaction to complete. More than one child server can be started in this way.

TCP/IP provides a concurrent server program called the **IMS Listener**. It is described in "Chapter 6. How to Customize and Operate the IMS Listener" on page 55.

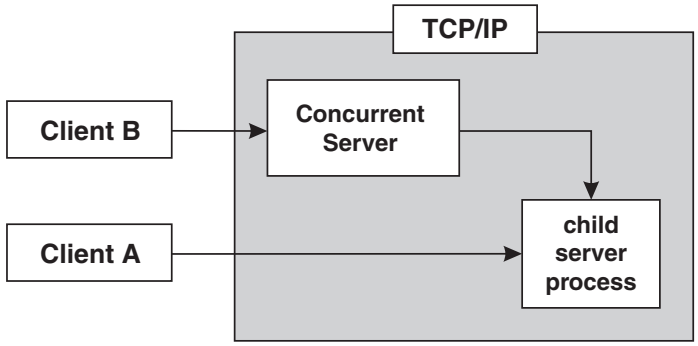


Figure 5. A Concurrent Server

Figure 3 on page 13 illustrates a concurrent server at work.

The Basic Socket Calls

The following is an overview of the basic socket calls.

The following calls are used by the server:

SOCKET
Obtains a socket to read from or write to.

BIND Associates a socket with a port number.

LISTEN
Tells TCP/IP that this process is listening for connections on this socket.

SELECT
Waits for activity on a socket.

ACCEPT
Accepts a connection from a client.

The following calls are used by a concurrent server to pass the socket from the parent server task (Listener) to the child server task (user-written application).

GIVESOCKET
Gives a socket to a child server task.

TAKESOCKET
Accepts a socket from a parent server task.

GETCLIENTID
Optionally used by the parent server task to determine its own address space name (if unknown) prior to issuing the GIVESOCKET.

The following calls are used by the client:

SOCKET
Allocates a socket to read from or write to.

CONNECT
Allows a client to open a connection to a server's port.

The following calls are used by both the client and the server:
WRITE
Sends data to the process on the other host.
READ Receives data from the other host.
CLOSE
Terminates a connection, deallocating the socket.

For full discussion and examples of these calls, see "Chapter 7. Using the CALL Instruction Application Programming Interface (API)" on page 61.

Server TCP/IP calls

To understand Socket programming, the client program and the server program must be considered separately. In this section the call sequence for the *server* is described; the next section discusses the typical call sequence for a *client*. This is the logical presentation sequence because the server is usually already in execution before the client is started. The step numbers (such as **5**) in this section refer to the steps in Figure 3 on page 13.

Socket

The server must first obtain a socket **1** . This socket provides an end-point to which clients can connect.

A socket is actually an index into a table of connections in the TCP/IP address space, so TCP/IP usually assigns socket numbers in ascending order. In COBOL, the programmer uses the SOCKET call to obtain a new socket.

The socket function specifies the address family (AF_INET), the type of socket (STREAM), and the particular networking protocol (PROTO) to use. (When PROTO is set to zero, the TCP/IP address space automatically uses the appropriate protocol for the specified socket type). Upon return, the newly allocated socket's descriptor is returned in RETCODE.

For an example of the SOCKET call, see "SOCKET" on page 130.

Bind

At this point **2** , an entry in the table of communications has been reserved for the application. However, the socket has no port or IP address associated with it until the BIND call is issued. The BIND function requires 3 parameters:

- The socket descriptor that was just returned by the SOCKET call.
- The number of the port on which the server wishes to provide its service
- The IP address of the network connection on which the server is listening. If the application wants to receive connection requests from any network interface, the IP address should be set to zeros.

For an example of the BIND call, see “BIND” on page 66.

Listen

After the bind, the server has established a specific IP address and port upon which other TCP/IP hosts can request connection. Now it must notify the TCP/IP address space that it intends to listen for connections on this socket. The server does this with the LISTEN **3** call, which puts the socket into passive open mode. *Passive open mode* describes a socket that can accept connection requests, but cannot be used for communication. A passive open socket is used by a listener program like the IMS Listener to await connection requests. Sockets that are directly used for communication between client and server are known as *active open* sockets. In passive open mode, the socket is open for client contacts; it also establishes a backlog queue of pending connections.

This LISTEN call tells the TCP/IP address space that the server is ready to begin accepting connections. Normally, only the number of requests specified by the BACKLOG parameter will be queued.

For an example of the LISTEN call, see “LISTEN” on page 99.

Accept

At this time **5** , the server has obtained a socket, bound the socket to an IP address and port, and issued a LISTEN to open the socket. The server main task is now ready for a client to request connection **4** . The ACCEPT call temporarily blocks further progress. [5]

The default mode for Accept is blocking. Accept behavior changes when the socket is non-blocking. The FCNTL() or IOCTL() calls can be used to disable blocking for a given socket. When this is done, calls that would normally block continue regardless of whether the I/O call has completed. If a socket is set to non-blocking and an I/O call issued to that socket would otherwise block (because the I/O call has not completed) the call returns with ERRNO 35 (EWOULDBLOCK).

When the ACCEPT call is issued, the server passes its socket descriptor, S, to TCP/IP. When the connection is established, the ACCEPT call returns a new socket descriptor (in RETCODE) that represents the connection with the client. **This is the socket upon which the server subtask communicates with the client**. Meanwhile, the original socket (S) is still allocated, bound and ready for use by the main task to accept subsequent connection requests from other clients.

To accept another connection, the server calls ACCEPT again. By repeatedly calling ACCEPT, a concurrent server can establish simultaneous sessions with multiple clients.

For an example of the ACCEPT call, see “ACCEPT” on page 64.

5. Blocking is a UNIX concept in which the requesting process is suspended until the request is satisfied. It is roughly analogous to the MVS wait. A socket is blocked while an I/O call waits for an event to complete. If a socket is set to block, the calling program is suspended until the expected event completes.

GIVESOCKET and TAKESOCKET

The GIVESOCKET and TAKESOCKET functions are not supported with the IMS TCP/IP OTMA Connection server. A server handling more than one client simultaneously acts like a dispatcher at a messenger service. A messenger dispatcher gets telephone calls from people who want items delivered and the dispatcher sends out messengers to do the work. In a similar manner, the server receives client requests, and then spawns tasks to handle each client.

In UNIX**-based servers, the *fork()* system call is used to dispatch a new subtask after the initial connection has been established. When the *fork()* command is used, the new process automatically inherits the socket that is connected to the client.

Because of architectural differences, CICS sockets does not implement the *fork()* system call.Tasks use the GIVESOCKET and TAKESOCKET functions to pass sockets from parent to child. The task passing the socket uses GIVESOCKET, and the task receiving the socket uses TAKESOCKET. See “GIVESOCKET and TAKESOCKET Calls” on page 21 for more information about these calls.

Read and Write

Once a client has been connected with the server, and the socket has been transferred from the main task (parent) to the subtask (child), the client and server exchange application data, using various forms of READ/WRITE calls. See “Read/Write Calls — the Conversation” on page 18 for details about these calls.

Client TCP/IP Calls

The TCP/IP call sequence for a client is simpler than the one for a concurrent server. A client only has to support one connection and one conversation. A concurrent server obtains a socket upon which it can listen for connection requests, and then creates a new socket for each new connection.

The Socket Call

In the same manner as the server, the first call **1** issued by the client is the SOCKET call. This call causes allocation of the socket on which the client will communicate.

```
CALL 'EZASOKET' USING SOCKET-FUNCTION SOCTYPE PROTO ERRNO RETCODE.
```

See “SOCKET” on page 130 for a sample of the SOCKET call.

The Connect Call

Once the SOCKET call has allocated a socket to the client, the client can then request connection on that socket with the server through use of the CONNECT call **4** .

The CONNECT call attempts to connect socket descriptor (S) to the server with an IP address of NAME. The CONNECT call blocks until the connection is accepted by the server. On successful return, the socket descriptor (S) can be used for communication with the server.

This is essentially the same sequence as that of the server; however, the client need not issue a BIND command because the port of a client has little significance. The client need only issue the CONNECT call, which issues an implicit BIND. When the CONNECT call is used to bind the socket to a port, the port number is assigned by the system and discarded when the connection is closed. Such a port is known

as an *ephemeral* port because its life is very short as compared with that of a concurrent server, whose port remains available for a prolonged time.

See “CONNECT” on page 70 for an example of the CONNECT call.

Read/Write Calls — the Conversation

A variety of I/O calls is available to the programmer. The READ and WRITE, READV and WRITEV, and SEND 6 and RECV 6 calls can be used only on sockets that are in the connected state. The SENDTO and RECVFROM, and SENDMSG and RECVMSG calls can be used regardless of whether a connection exists.

The WRITEV, READV, SENDMSG, and RECVMSG calls provide the additional features of scatter and gather data. Scattered data can be located in multiple data buffers. The WRITEV and SENDMSG calls gather the scattered data and send it. The READV and RECVMSG calls receive data and scatter it into multiple buffers.

The WRITE and READ calls specify the socket S on which to communicate, the address in storage of the buffer that contains, or will contain, the data (BUF), and the amount of data transferred (NBYTE). The server uses the socket that is returned from the ACCEPT call.

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

- For an example of the READ call, see “READ” on page 101.
- For an example of the WRITE call, see “WRITE” on page 134.

The Close Call

When the conversation is over, both the client and server call CLOSE to end the connection. The CLOSE call also deallocates the socket, freeing its space in the table of connections. For an example of the CLOSE call, see “CLOSE” on page 68

Other Socket Calls

Several other calls that are often used — particularly in servers — are the SELECT call, the GIVESOCKET/TAKESOCKET calls, and the IOCTL and FCTL calls. These calls are discussed next.

The SELECT Call

Applications such as concurrent servers often handle multiple sockets at once. In such situations, the SELECT call can be used to simplify the determination of which sockets have data to be read, which are ready for data to be written, and which have pending exceptional conditions. An example of how the SELECT call is used can be found in Figure 6 on page 19.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'SELECT'.
    01  MAXSOC          PIC 9(8) BINARY VALUE 50.
    01  TIMEOUT.
        03  TIMEOUT-SECONDS   PIC 9(8) BINARY.
        03  TIMEOUT-MILLISEC  PIC 9(8) BINARY.
    01  RSNDMASK        PIC X(50).
    01  WSNDMASK        PIC X(50).
    01  ESNDMASK        PIC X(50).
    01  RRETMASK        PIC X(50).
    01  WRETMASK        PIC X(50).
    01  ERETMASK        PIC X(50).
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

PROCEDURE
     CALL 'EZASOKET' USING SOC-FUNCTION MAXSOC TIMEOUT
                    RSNDMASK WSNDMASK ESNDMASK
                    RRETMASK WRETMASK ERETMASK
                    ERRNO RETCODE.
```

Figure 6. The SELECT Call

In this example, the application *sends* bit sets (the xSNDMASK sets) to indicate which sockets are to be tested for certain conditions, and *receives* another set of bits (the xRETMASK sets) from TCP/IP to indicate which sockets meet the specified conditions.

The example also indicates a time-out. If the time-out parameter is NULL, this is the C language API equivalent of a wait forever. (In Sockets Extended, a negative timeout value is a wait forever.) If the time-out parameter is nonzero, SELECT only waits the timeout amount of time for at least one socket to become ready on the indicated conditions. This is useful for applications servicing multiple connections that cannot afford to wait for data on a single connection. If the xSNDMASK bits are all zero, SELECT acts as a timer.

With the Socket SELECT call, you can define which sockets you want to test (the xSNDMASKs) and then wait (block) until one of the specified sockets is ready to be processed. When the SELECT call returns, the program knows only that some event has occurred, and it must test a set of bit masks (xRETMASKs) to determine which of the sockets had the event, and what the event was.

To maximize performance, a server should only test those sockets that are active. The SELECT call allows an application to select which sockets will be tested, and for what. When the Select call is issued, it blocks until the specified sockets are ready to be serviced (or, optionally) until a timer expires. When the select call returns, the program must check to see which sockets require service, and then process them.

To allow you to test any number of sockets with just one call to SELECT, place the sockets to test into a bit set, passing the bit set to the select call. A bit set is a string of bits where each possible member of the set is represented by a 0 or a 1. If the member's bit is 0, the member is not to be tested. If the member's bit is 1, the member is to be tested. Socket descriptors are actually small integers. If socket 3 is a member of a bit set, then bit 3 is set; otherwise, bit 3 is zero.

Therefore, the server specifies 3 bit sets of sockets in its call to the SELECT function: one bit set for sockets on which to receive data; another for sockets on which to write data; and any sockets with exception conditions. The SELECT call

tests each selected socket for activity and returns only those sockets that have completed. On return, if a socket's bit is raised, the socket is ready for reading data or for writing data, or an exceptional condition has occurred.

The format of the bit strings is a bit awkward for an assembler programmer who is accustomed to bit strings that are counted from left to right. Instead, these bit strings are counted from right to left.

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

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

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

Look at the first fullword you pass in a bit string in Table 1.

Table 1. First Fullword Passed in a Bit String in Select

Socket Descriptor Numbers Represented by Byte	Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
Byte 0	31	30	29	28	27	26	25	24
Byte 1	23	22	21	20	19	18	17	16
Byte 2	15	14	13	12	11	10	9	8
Byte 3	7	6	5	4	3	2	1	0

In these examples, we use standard assembler numbering notation; the left-most bit or byte is relative zero.

If you want to test socket descriptor number 5 for pending read activity, you raise bit 2 in byte 3 of the first fullword (X'00000020'). If you want to test both socket descriptor 4 and 5, you raise both bit 2 and bit 3 in byte 3 of the first fullword (X'00000030').

If you want to test socket descriptor number 32, you must pass two fullwords, where the numbering scheme for the second fullword resembles that of the first. Socket descriptor number 32 is bit 7 in byte 3 of the second fullword. If you want to test socket descriptors 5 and 32, you pass two fullwords with the following content: X'0000002000000001'.

The bits in the second fullword represents the socket descriptor numbers shown in Table 2.

Table 2. Second Fullword Passed in a Bit String in Select

Socket Descriptor Numbers Represented by Byte	Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
Byte 4	63	62	61	60	59	58	57	56

Table 2. Second Fullword Passed in a Bit String in Select (continued)

Socket Descriptor Numbers Represented by Byte	Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
Byte 5	55	54	53	52	51	50	49	48
Byte 6	47	46	45	44	43	42	41	40
Byte 7	39	38	37	36	35	34	33	32

If you develop your program in COBOL or PL/I, you may find that the EZACIC06 routine, which is provided as part of TCP/IP for MVS, will make it easier for you to build and test these bit strings. This routine translates between a character string mask (one byte per socket) and a bit string mask (one bit per socket).

In addition to its function of reporting completion on Read/Write events, the SELECT call can also be used to determine completion of events associated with the LISTEN and GIVESOCKET calls.

- When a connection request is pending on the socket for which the main process issued the LISTEN call, it will be reported as a pending read.
- When the parent process has issued a GIVESOCKET, and the child process has taken the socket, the parent's socket descriptor is selected with an exception condition. The parent process is expected to `close` the socket descriptor when this happens.

IOCTL and FCNTL Calls

In addition to SELECT, applications can use the IOCTL or FCNTL calls to help perform asynchronous (nonblocking) socket operations. An example of the use of the IOCTL call is shown in "IOCTL" on page 95.

The IOCTL call has many functions; establishing blocking mode is only one of its functions. The value in COMMAND determines which function IOCTL will perform. The REQARG of 0 specifies non-blocking (a REQARG of 1 would request that socket S be set to blocking mode). When this socket is passed as a parameter to a call that would block (such as RECV when data is not present), the call returns with an error code in RETCODE, and ERRNO set to `EWOULDBLOCK`. Setting the mode of the socket to nonblocking allows an application to continue processing without becoming blocked.

GIVESOCKET and TAKESOCKET Calls

The GIVESOCKET and TAKESOCKET functions are not supported with the IMS TCP/IP OTMA Connection server. Tasks use the GIVESOCKET and TAKESOCKET functions to pass sockets from parent to child.

For programs using TCP/IP for MVS, each task has its own unique 8-byte name. The main server task passes three arguments to the GIVESOCKET call:

- The socket number it wants to give
- Its own name [6]
- The name of the task to which it wants to give the socket

6. If a task does not know its address space name, it can use the GETCLIENTID function call to determine its unique name.

If the server does not know the name of the subtask that will receive the socket, it blanks out the name of the subtask. [7] The first subtask calling TAKESOCKET with the server's unique name receives the socket.

The subtask that receives the socket must know the main task's unique name and the number of the socket that it is to receive. This information must be passed from main task to subtask in a work area that is common to both tasks.

- In IMS, the parent task name and the number of the socket descriptor are passed from parent (Listener) to child (MPP) through the message queue.
- IN CICS, the parent task name and the socket descriptor number are passed from the parent (Listener) to the transaction program by means of′ the EXEC CICS START and EXEC CICS RETREIVE function.

Because each task has its own socket table, the socket descriptor obtained by the main task is not the socket descriptor that the subtask will use. When TAKESOCKET accepts the socket that has been given, the TAKESOCKET call assigns a new socket number for the subtask to use. This new socket number represents the same connection as the parent's socket. (The transferred socket might be referred to as socket number 54 by the parent task and as socket number 3 by the subtask; however, both socket descriptors represent the same connection.)

Once the socket has successfully been transferred, the TCP/IP address space posts an exceptional condition on the parent's socket. The parent uses the SELECT call to test for this condition. When the parent task SELECT call returns with the exception condition on that socket (indicating that the socket has been successfully passed) the parent issues CLOSE to complete the transfer and deallocate the socket from the main task.

To continue the sequence, when another client request comes in, the concurrent server (Listener) gets another new socket, passes the new socket to the new subtask, and dissociates itself from that connection. And so on.

Summary

To summarize, the process of passing the socket is accomplished in the following way:

- After creating a subtask, the server main task issues the GIVESOCKET call to pass the socket to the subtask. If the subtask's address space name and subtask ID are specified in the GIVESOCKET call, (as with CICS) only a subtask with a matching address space and subtask ID can take the socket. If this field is set to blanks, (as with IMS) any MVS address space requesting a socket can take this socket.
- The server main task then passes the socket descriptor and concurrent server's ID to the subtask using some form of commonly addressable technique such as the IMS Message Queue.
- The concurrent server issues the SELECT call to determine when the GIVESOCKET has successfully completed.
- The subtask calls TAKESOCKET with the concurrent server's ID and socket descriptor and uses the resulting socket descriptor for communication with the client.
- When the GIVESOCKET has successfully completed, the concurrent server issues the CLOSE call to complete the handoff.

7. This is the case in IMS because the Listener has no way of knowing which Message Processing Region will inherit the socket.

An example of a concurrent server is the IMS Listener. It is described in “Chapter 6. How to Customize and Operate the IMS Listener” on page 55. Figure 5 on page 14 shows a concurrent server.

What You Need to Run IMS TCP/IP

IMS TCP/IP using the IMS Listener and IMS Assist Module is designed for use on an MVS/SP host system running: IMS/ESA Version 4 or later.

A TCP/IP host can communicate with any remote IMS or non-IMS system that runs TCP/IP. The remote system can, for example, run a UNIX or OS/2 operating system.

TCP/IP Services

TCP/IP Services is not described in this book because it is a prerequisite for IMS TCP/IP. However, much material from the TCP/IP library has been repeated in this book in an attempt to make it independent of that library.

A Summary of What IMS TCP/IP Provides

Figure 7 on page 24 shows how IMS TCP/IP allows IMS applications to access the TCP/IP network. It shows that IMS TCP/IP makes the following facilities available to your application programs:

The sockets calls (1 and 2 in Figure 7 on page 24)

The socket API is available both in the C language and in COBOL, PL/I, or assembler language. It includes the following socket calls:

Basic calls:	*socket, bind, connect, listen, accept, shutdown, close*
Read/write calls:	*send, sendto, recvfrom, read, write*
Advanced calls:	*gethostname, gethostbyaddr, gethostbyname, getpeername, getsockname, getsockopt, setsockopt, fcntl, ioctl, select*
IBM-specific calls:	*initapi, getclientid, givesocket, takesocket*

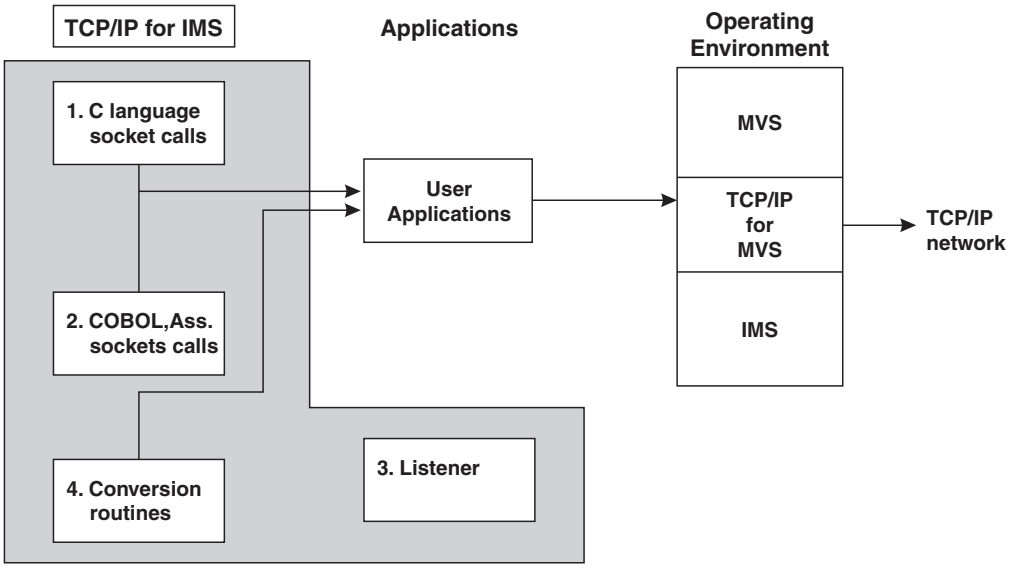


Figure 7. How User Applications Access TCP/IP Networks with IMS TCP/IP

IMS TCP/IP provides for both connection-oriented and connectionless (datagram) services, using the TCP and UDP protocols respectively. TCP/IP does not support the IP (raw socket) protocol.

The Listener (3)

IMS TCP/IP includes a concurrent server application, called the Listener, to which the client makes initial connection requests. The Listener passes the connection request on to the user-written server, which is typically an IMS Message Processing Program.

Conversion routines (4)

IMS TCP/IP provides the following conversion routines, which are part of the base TCP/IP Services product:

- An EBCDIC-to-ASCII conversion routine, used to convert EBCDIC data within IMS to the ASCII format used in TCP/IP networks and workstations. It is run by calling module `EZACIC04`.
- A corresponding ASCII-to-EBCDIC conversion routine (`EZACIC05`).
- A module that converts COBOL character arrays into bit-mask arrays used in TCP/IP. This module, which is run by calling `EZACIC06`, is used with the socket SELECT call.
- A module that interprets a C language structure known as Hostent. (`EZACIC08`).

Part 2. Using The IMS Listener

Chapter 3. Principles of Operation

This chapter describes the operation of the Listener and the Assist module. Its purpose is to explain how a TCP/IP-to-IMS connection is established, and how the client and server exchange application data. For specific data formats and the socket protocols used when coding a TCP/IP client or server, see “Chapter 4. How to Write an IMS TCP/IP Client Program” on page 39 and “Chapter 5. How to Write an IMS TCP/IP Server Program” on page 47.

Overview

The IMS TCP/IP feature consists of 3 components: the IMS Listener, the IMS Assist module, and the Sockets Extended API. [8] The Sockets Extended API can either be used independently, or with the other 2 components. When the Sockets Extended interface is used independently, an IMS MPP can either serve as a client or as a server.

When the IMS Listener is used, the IMS MPP acts as a **server**, and the TCP/IP remote acts as the **client**. The Assist module is dependent upon the IMS Listener; therefore, when the Assist module is used, IMS is the server.

Because the Listener and the Assist module are designed to support IMS as a server, the next several chapters are based on that assumption. For a discussion of IMS as **client**, see “When the Client is an IMS MPP” on page 36, later in this chapter, and the sample program on “Sample Program - IMS MPP Client” on page 171.

The Role of the IMS Listener

Since the IMS Transaction Manager does not support direct connection with TCP/IP, some other program must establish that connection. When IMS is acting as a ***server*** to a TCP/IP-connected ***client***, that program is the IMS Listener — an IMS batch message program (BMP) whose main function it is to establish connection between the client and the requested IMS transaction.

When the client requests the services of an IMS message processing program (MPP), it sends a message to the IMS host containing the transaction code of that MPP. The IMS Listener receives that request and schedules the requested MPP; it then holds the connection until the MPP starts and accepts the connection. Once the MPP owns the connection, the Listener is no longer involved with it.

The Role of the IMS Assist Module

The IMS Assist module is a subroutine, called from an IMS MPP (server) that translates conventional IMS communication calls into the corresponding socket calls. Its use is optional. Its purpose is to shield the programmer from having to understand TCP/IP programming. To exchange data with the client, the server program issues traditional IMS message queue calls (GU, GN, ISRT). These calls are intercepted by the Assist module, which issues the appropriate socket calls.

8. Shipped with the TCP/IP Services base product

Use of the IMS Assist Module — Pros and Cons

The Assist module makes message processing program (MPP) coding easier, but is accompanied by a series of trade-offs. This section discusses the trade-offs between implicit mode and explicit mode.

- Implicit-mode application programmers use conventional IMS Transaction Manager (TM) calls and require no special training; explicit-mode application programmers must understand TCP/IP socket calls and protocols.
- Implicit-mode transactions must adhere to constraints imposed by the IMS Assist module. By contrast, explicit-mode transactions use the TCP/IP socket call interface and have no specific protocol requirements other than the orderly initiation and termination of the transaction.
- Implicit-mode transactions obtain their message input from the IMS message queue. Since the Listener must put the input message segments on the queue before the server begins execution, the client sends all application data with the transaction request. Explicit-mode transactions bypass the message queue for all application data — both input, and output.
- Implicit-mode transactions are limited to a single GU-GN/ISRT iteration (one input of one or more segments, followed by one output of one or more segments) for each message retrieved from the IMS message queue. By contrast, explicit-mode transactions have no such limit. Unlimited read/write sequences make it possible to design conversations in which the two programs talk back and forth without limit. [9]

Client/Server Logic Flow

The following section describes the flow of a client/server application through the system — starting with the client and continuing on through the Listener to the server. The complete transaction, including initiation, execution, and termination is traced.

How the Connection is Established

The following paragraphs describe the functions the Listener performs in coordinating between the client and the server. With the exception of paragraph 6, the Listener performs the same steps for both explicit- and implicit-mode servers. Paragraph numbers correspond to the step numbers in Figure 1.

9. Because of the potential for long running conversations, MPPs with multiple conversational iterations should be carefully designed to avoid the possibility of extended message processing region occupancy.

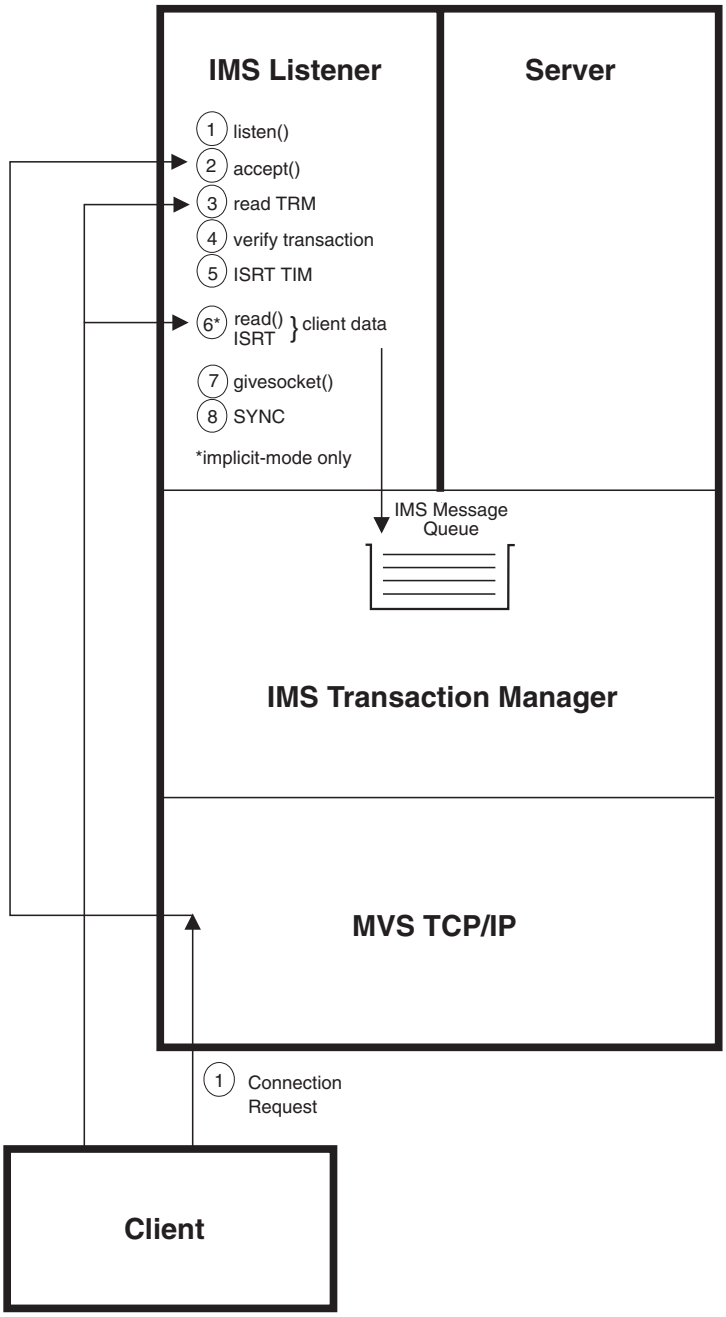


Figure 8. IMS TCP/IP Message Flow for Transaction Initiation

1. Connection request

   The IMS Listener is an IMS batch message processing program (BMP). When the Listener starts, it establishes a socket on which it can “listen” for connection requests. It binds itself to the specified port, and then listens for requests from TCP/IP clients. When a client sends a connection request, MVS TCP/IP notifies the Listener of the request.
2. Connection processing

   When the Listener receives a connection request, it issues a socket ACCEPT call, which creates a new socket specifically for that connection.
3. Transaction-Request Message

The client then sends a transaction-request message (TRM) segment, which includes the 8-byte name of the requested IMS server transaction (otherwise known as the TRANCODE).

4. Transaction verification

   The Listener performs several tests to ensure that the requested transaction should be accepted:

   - The TRANCODE is tested against IMS Listener configuration file TRANSACTION statements to ensure that the requested transaction is eligible to be executed from a TCP/IP client.
   - If security data is included in the transaction-request message (TRM), that data is passed to a user-written security exit. The purpose of this exit is to validate the credentials of the client prior to allowing the transaction to be scheduled.
   - The Listener issues an IMS CHNG call to a modifiable alternate PCB, specifying the TRANCODE of the desired transaction. It then issues an IMS INQY call to ensure that the transaction is not stopped (due to previous abend or Master Terminal Operator action).

   The following actions depend on the results of the verification:

   - If the transaction request is *rejected*, the IMS Listener returns a request-status message (RSM) segment to the client with an indication of the reason for rejecting the request; it then closes the connection.
   - If the transaction request is *accepted* the requested transaction is scheduled (the Listener *does not* return a status message to the client).

5. Transaction Initiation Message (TIM)

   The Listener then inserts (ISRT) a transaction initiation message (TIM) segment to the IMS message queue. This message contains information needed by the server program when it takes responsibility for the connection. (Note that the client sends the transaction *request* message (TRM) to the Listener; the Listener sends the transaction *initiation* message (TIM) to the server.)

6. Client-to-server input data transfer (implicit mode only)

   If the transaction is in implicit mode, the Listener reads the client-to-server input data and places it on the message queue.

7. Pass the socket to the server

   Next, the Listener issues a GIVESOCKET call, which makes the socket available to the server program.

8. Schedule the transaction

   Finally, the Listener issues an IMS SYNC call to schedule the requested IMS transaction and waits for the server program to take responsibility for the connection.

When the server issues a TAKESOCKET call, the Listener has completed its responsibility for the socket and dissociates itself from the connection.

Note: The Listener is a never-ending IMS Batch Message Program, which processes multiple concurrent transactions.

How the Server Exchanges Data with the Client

Once the server begins execution, the protocol to pass input data to the server is a function of whether the transaction mode is explicit or implicit.

Explicit-Mode Transactions

The following section describes an explicit-mode server program which exchanges application data with a client.

Step numbers in Figure 2 correspond to the paragraph numbers below.

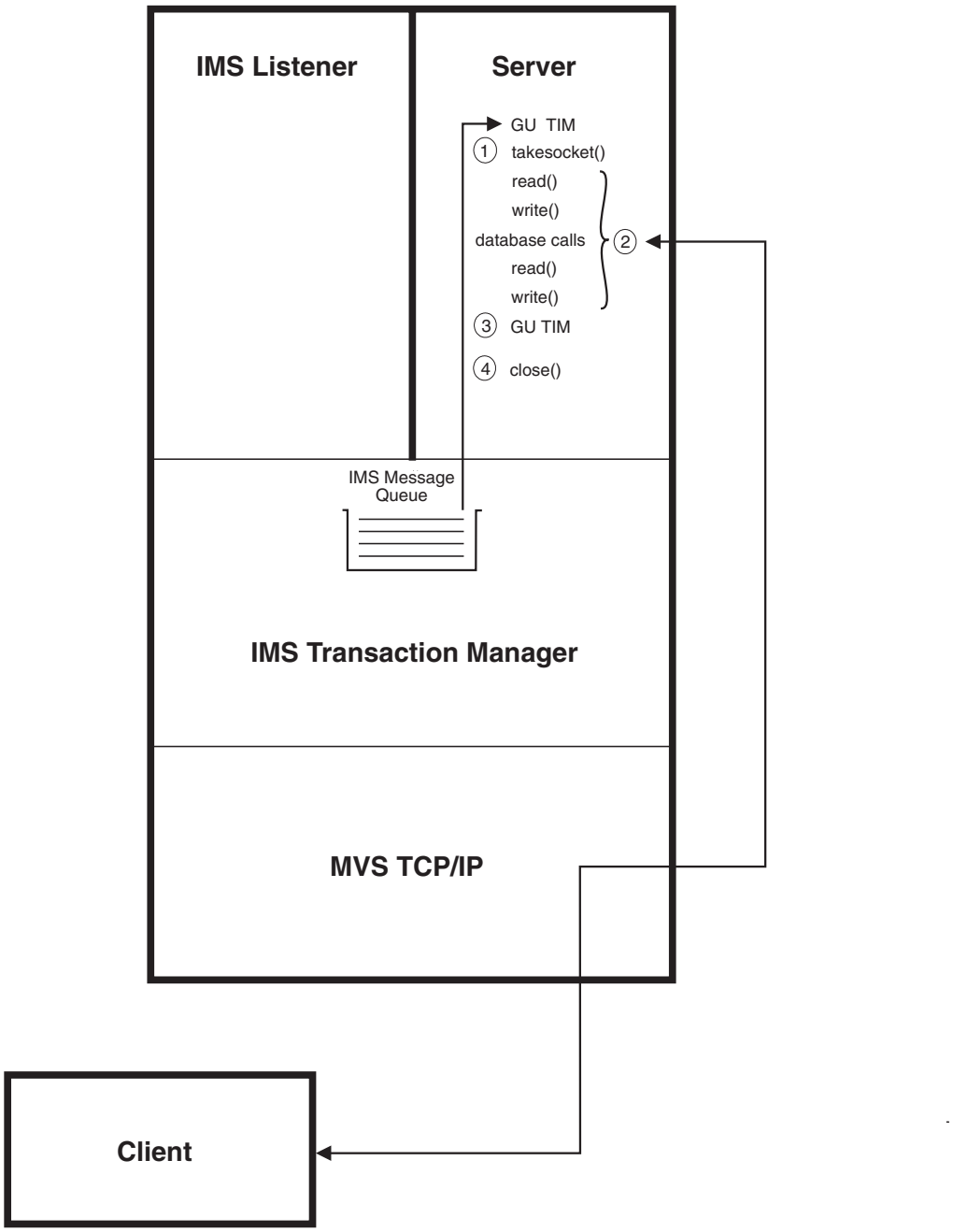


Figure 9. IMS TCP/IP Message Flow for Explicit-Mode Input/Output

1. Once an explicit-mode server begins execution, it issues an IMS GU call to obtain the transaction initiation message (TIM) segment, an INITAPI to establish connection with MVS TCP/IP, and a TAKESOCKET call to establish direct connection between client and server.

2. Subsequently, socket READ and WRITE commands are used to exchange data between client and server. The conversation can consist of any number of database calls and socket READ/WRITE exchanges. [10] Client data is not passed through the IMS message queue and is not subject to any predefined protocols.
3. The transaction indicates completion by issuing another GU to the I/O PCB. This notifies the Transaction Manager that the database changes should be committed. At this point, the server program might send a message to the client indicating that the database changes have been successfully completed.

   If another message awaits this transaction, the GU will cause the first segment of that message to be retrieved and the program should issue a new TAKESOCKET call to start the process again.
4. When the GU call returns with a QC status code, the server ends the conversation by closing the socket.

Implicit-Mode Transactions

The following section describes how the Assist module and the server program interact to exchange application data with the client. The paragraph numbers correspond to the step numbers in Figure 3.

10. Because of the potential for long running conversations, MPPs with multiple conversational iterations should be carefully designed to avoid the possibility of extended message processing region occupancy.

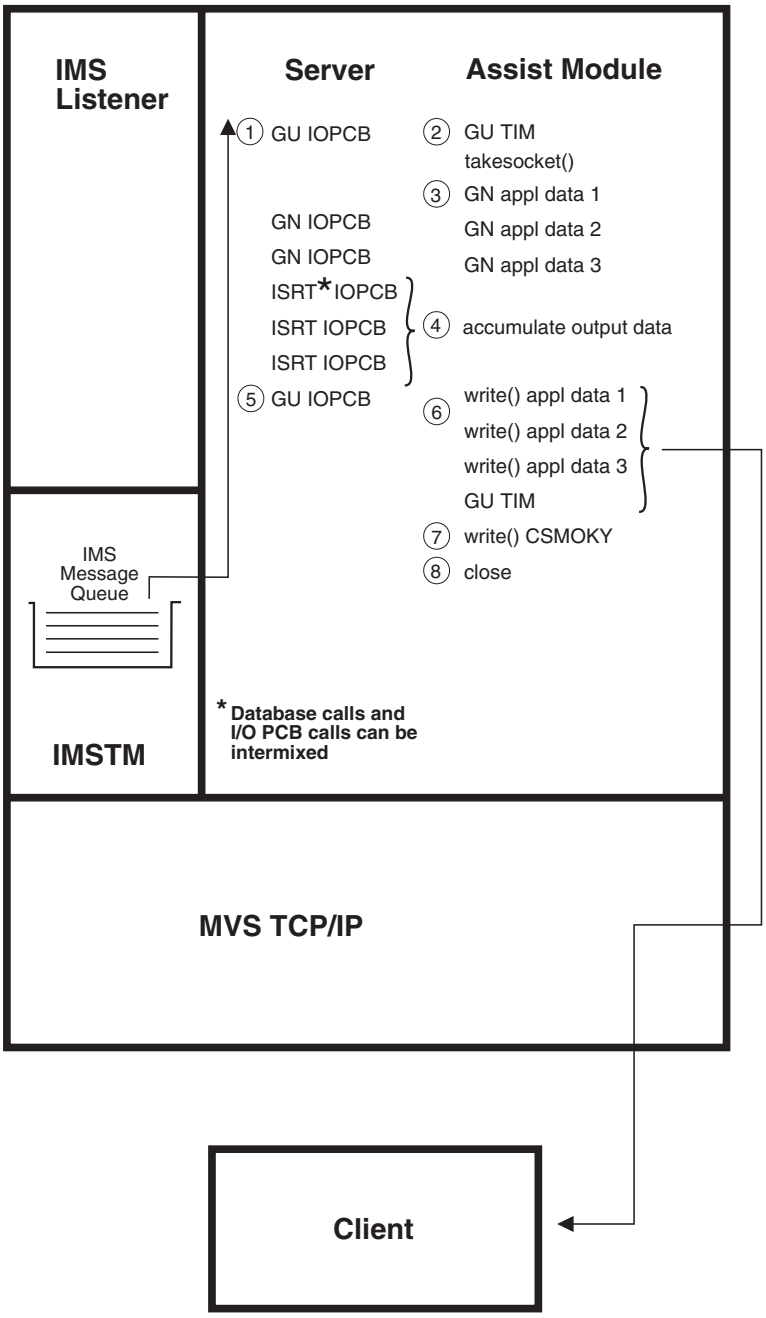


Figure 10. IMS TCP/IP Message Flow for Implicit Mode Input/Output

1. Server GU

   GU must be the first IMS call issued by the server to the I/O PCB. The Assist module retrieves the first segment from the message queue and examines it (for *LISTNR* in the first field) to determine whether it is a transaction initiation message. (If the message was not sent by the Listener, the Assist module assumes the transaction was started by an SNA terminal and immediately passes the input segment to the server. In this case, subsequent I/O PCB calls (as well as database calls) are passed directly through to IMS without further consideration.)
2. Transaction Initiation Message (TIM)

If the message was sent by the Listener, the initial message segment is the transaction initiation message (TIM); the Assist module ***does not*** return it to the server. Instead, the Assist module uses the TIM contents to issue the TAKESOCKET to establish connection between the client and the server program.

3. Server input data

   Once the server owns the socket, the Assist module issues a GN to retrieve the first segment of the client input message and returns it to the server program. Thus, the server program never sees the TIM; it receives the first data segment in response to its GU. Subsequent GN calls from the server cause the Assist module to retrieve the remaining segments of the message. When the Assist module reads the last input segment for that transaction from the message queue, it receives a QD status code from IMS, which it returns to the server program.

   After the initial GU to the I/O PCB, server GN calls, ISRT calls, and database calls can be intermixed.

4. Server output data

   When the server program issues ISRT calls to send output message segments to the client, the IMS Assist module accumulates the output segments, up to maximum of 32KB, into a buffer.

5. Commit

   The server signals completion by issuing a GU to the I/O PCB.

6. TCP/IP writes application data to the client.

   When the server issues the GU, the Assist module issues WRITE calls to send the data to the client and passes the GU to the IMS Transaction Manager to commit the database changes.

7. Confirmation

   If the GU is successful, (that is, QC status or spaces) the Assist module sends a complete-status message segment (CSM) to the client to confirm the successful commit and passes the status code back to the server.

8. Close the socket

   Once the complete-status message has been sent to the client, the Assist module closes the socket, ending the connection.

   If the GU in the previous step resulted in a 'bb' status code (indicating successful return of another message) the program logic returns to step 2 to process the new message.

How the IMS Listener Manages Multiple Connection Requests

The IMS Listener uses 2 queues for the management of connection requests:

1. The ***backlog*** queue (managed by MVS TCP/IP) contains client connection requests that have not yet been accepted by the Listener. If a client requests a connection while the backlog queue is full, TCP/IP rejects the connection request. The number of requests allowed in the backlog queue is specified in the LISTENER startup configuration statement (BACKLOG parameter), see “LISTENER Statement” on page 57.
2. The ***active sockets*** queue contains the sockets that are held by the Listener while they wait for assignment to a server program. Once the Listener has accepted the connection, the connection belongs to the Listener until it is accepted by the server. If the Listener uses all of its sockets and cannot accept any more connections, subsequent requests go into the backlog queue. The

maximum number of sockets available is specified in the LISTENER startup configuration statement, (MAXACTSKT parameter), see “LISTENER Statement” on page 57.

Use of the IMS Message Queue

In conventional 3270 applications, the IMS message queue is a mechanism for passing communications between an MPP and another entity, such as a 3270-type terminal, or another message processing program (MPP). The IMS TCP/IP feature uses the message queue for communication between the Listener and the MPP. Messages from and to TCP/IP hosts bypass IMS message format services (MFS). The following section describes how IMS TCP/IP uses the IMS message queue:

Input Messages

(Messages that are *input* to the MPP)

- Explicit-mode transactions only use the message queue to pass the transaction initiation message (TIM) from the Listener to the server. All application data sent by the client is received by the server using sockets READ calls, thus bypassing the IMS message queue.
- Implicit-mode transactions use the message queue both for the TIM (which is trapped by the Assist module and not passed on to the server) and for all client-to-server application data (which is passed to the server in response to IMS GU, GN calls).

Output Messages

All messages that are *output* from the server go directly via TCP/IP to the client; they do not pass through the message queue.

- Explicit-mode servers use socket WRITE calls to send application data directly to the client.
- Implicit-mode servers use the IMS ISRT call for output, but the inserted data is trapped by the Assist module which, in turn, issues socket WRITE calls to send the data to the client.

Call Sequence for the IMS Listener

Although you will probably not be writing a Listener program, it is important that you match the sequence of calls issued by the Listener when you write your client program. The Listener call sequence is:

Call **Explanation of Function**

INITIALIZE LISTENER

INITAPI
Connect the Listener to MVS TCP/IP at Listener startup. (This call is only used in programs written to the Sockets Extended interface.

SOCKET
Create a socket descriptor.

BIND Allocate the local port for the socket. This port is used by clients when requesting connection to IMS.

LISTEN
Create a queue for incoming connections.

WAIT FOR CONNECTION REQUEST

SELECT
Wait for an incoming connection request.

ACCEPT
Accept the incoming connection request; create a new socket descriptor to be used by the server for this specific connection.

READ Read TRM; determine the IMS TRANCODE.

CHNG Change the modifiable alternate PCB to reflect the desired IMS TRANCODE.

INQY Ensure the desired IMS TRANCODE is available for scheduling.

ISRT Use the alternate PCB to insert the transaction initiation message (TIM) and pass control information and user input data to the server.

GIVESOCKET
Pass the newly created socket to the server.

SYNC Schedule the requested transaction.

SELECT
Wait for the server to take the socket.

CLOSE
Release the socket.

END OF CONNECTION REQUEST

Return to ″WAIT FOR CONNECTION REQUEST″

SHUTDOWN LISTENER

CLOSE
Close the socket through which the Listener receives connection requests from MVS TCP/IP.

TERMAPI
Disconnect the Listener from MVS TCP/IP before shutting down

Application Design Considerations

The following is a set of guidelines and limitations that should be considered when designing IMP TCP/IP applications.

Programs That Are Not Started by the IMS Listener

It is expected that, in most cases, IMS server applications will be started by the IMS Listener. Such programs are known as *dependent* programs because the Listener establishes the TCP/IP connection.

Under some circumstances, application design considerations require that an application establish its own connection between TCP/IP and IMS. For example, an IMS MPP might require the services of a TCP/IP-connected UNIX processor.

An IMS application of this type is known as an *independent* program because it is not started by the Listener. Because independent programs don't use Listener services, they must define their own protocol.

When the Client is an IMS MPP

In this manual, the underlying assumption is that the TCP/IP host acts as client and the IMS MPP acts as server. However, this is not always the case.

For example, consider an IMS MPP that requires the services of a TCP/IP-connected AIX* host. Such an MPP (acting as a client) initiates a TCP/IP

conversation by issuing the ***client*** TCP/IP call sequence. The TCP/IP host would respond with the ***server*** TCP/IP call sequence. This application design is supported because the MPP communicates directly with MVS TCP/IP. The IMS TCP/IP feature does not impose any unique restrictions on the type and usage of socket calls executed by such a program; however, because of the unique and unstructured communication requirements of this application design, you must use explicit mode for this type of program.

Abend Processing

When a task that owns a socket fails, MVS TCP/IP closes the socket. Therefore, when an IMS MPP abends, regardless of the reason, the socket is no longer available and communication between server and client is no longer possible.

True Abends

If an IMS TCP/IP server program abends (for example, because of an S0Cx condition), database changes in progress are backed out and the transaction task is terminated. This breaks the TCP/IP connection. When the connection is broken, the client receives a negative status code and an error number that indicates that the connection has been broken. Upon receipt of this indication, the client should assume that the transaction did not complete and that the database changes have not been made. The client could reschedule the transaction, but the IMS TM will have probably “stopped” it from further execution as a result of the abend.

The solution is to correct the reason for the abend and restart the transaction.

Pseudo Abends

Under certain situations IMS applications cannot complete. Upon such a condition, IMS abends the MPR with a status code (usually U0777, U02478, U02479, or U03303) and reschedules it. This action is not apparent to the conventional 3270-type user.

However, when an IMS TCP/IP transaction is pseudo-abended, the action is apparent to the client because the connection between client and server is lost when the server MPR is abended. In this case, IMS TM reschedules the transaction and places the input message (including the TIM) back on the message queue. When the transaction is rescheduled and issues a GU for the TIM, the socket described in the TIM no longer represents a valid connection. and the associated TAKESOCKET call will fail. At this time, the Assist module will detect the failure of the socket call and return a ZZ status code to the server program. Upon receipt of this status code, the server program should end normally.

Note: At the time of the pseudo-abend, the IMS TM backs out database changes, so the client should restart the transaction.

An Alternative: As an alternative, for deadlock situations it is suggested that you define the transaction as INIT STATUS GROUP B, which allows the application program to regain control after a deadlock with a BC status code (instead of terminating with a U0777 abend). This allows the server program to regain control after the deadlock and notify the client while the connection is still available. :elblbox

Implicit-Mode Support for ROLB Processing

If a server program issues the IMS ROLB call, all database changes are reversed, and all output messages are erased from the IMS message queue. However, the

client is not automatically notified of this action and will (when the transaction completes normally) receive a CSMOKY message, indicating normal completion.

As a result, for transactions that conditionally issue the ROLB call, it is recommended that the server send a message to the client indicating whether the ROLB command was executed. Otherwise, the client might incorrectly interpret the CSMOKY message to mean that database changes have been made (when in fact, the message simply denotes successful termination of the transaction).

Restrictions

- Transactions must be defined as MODE=SNGL in the IMS TRANSACT macro; this will ensure that the database buffers are emptied (flushed) to direct access storage when the second and subsequent GU calls are issued.
- Transactions must not reference other systems (MSC is not supported).
- Transactions must not be conversational (that is, they must not use the IMS Scratch Pad Area (SPA)).

Chapter 4. How to Write an IMS TCP/IP Client Program

When writing an IMS TCP/IP client program, the programmer must follow conventions established by the IMS Listener and by the IMS Assist module (if used). This chapter describes the call sequences and input/output data formats to be used by the client program. For server programming, see “Chapter 5. How to Write an IMS TCP/IP Server Program” on page 47.

Note that, in the context of this chapter, a “client” is typically a TCP/IP host that is requesting the services of an IMS message processing program (MPP). This is considered to be the normal case. However, in some situations, an MPP can start as a server and then (because it needs the services of another program) switch roles from server to client.

In this chapter, the client will be assumed to be the TCP/IP host and the server, the IMS MPP.

Client Program Logic Flow — General

For both explicit- and implicit-mode clients the logic flow is essentially the same:

The client initiates the request for a specific IMS MPP server by communicating with MVS TCP/IP, which passes the request on to the IMS Listener. The Listener schedules the transaction and the client then exchanges application data with the server. When the transaction is complete, the connection is closed; each client request for an IMS transaction requires a new TCP/IP connection.

The following two sections provide more details about the programming requirements for explicit-mode and implicit-mode clients, respectively.

Explicit-Mode Client Program Logic Flow

When the client requests the services of an explicit-mode server, the only protocol imposed by IMS TCP/IP is that the client must begin by establishing TCP/IP connectivity and sending a transaction-request message (TRM).

The Listener uses contents of the transaction-request message (TRM) to determine which transaction to schedule. If the request is not accepted (for example, because of failure to pass the security exit, or because the transaction was stopped by the IMS master terminal operator), the Listener returns a request-status message (RSM) to the client with an indication of the cause of failure. (See “Request-Status Message Segment” on page 44 for the format of the request-status message).

Once an explicit-mode client and server are in communication, there is no predefined input/output protocol. Rules of the conversation are established by agreement between the two programs. Any number of READ/WRITE calls can be issued. Upon termination, the server program should commit any database changes, notify the server of successful completion, and close the socket.

It is suggested that, when all database updates have been committed, the server notify the client by sending a “success” message to the client. This notifies the client that the transaction has completed properly and that all database updates have been committed. Unless such a message is sent, the client has no way of knowing that the transaction completed properly.

Explicit-Mode Client Call Sequence

The call sequence to be used by an explicit-mode client program is:

Call	Explanation of Function
INITAPI	Open the interface. (Only required for client programs that use MVS TCP/IP socket calls).
SOCKET	Obtain a socket descriptor.
CONNECT	Request connection to the IMS Listener port.
WRITE	Send a transaction-request message (TRM)
READ	Test for successful transaction initiation [11]
WRITE/READ	Explicit-mode transactions can issue any number of READ or WRITE socket call sequences.
READ	Ensure that the server ended normally and that the database changes are committed.
CLOSE	Terminate the connection and release socket resources.

Explicit-Mode Application Data

Format

Explicit-mode clients must initiate the connection with the server by sending the transaction-request message (TRM) to the IMS host. The format of this message is defined later in this chapter. Explicit-mode application data is formatted according to agreement between client and server. Explicit-mode imposes no application data format requirements.

Data Translation.

In explicit-mode, application data translation from ASCII to EBCDIC (if necessary) is the responsibility of the client and server programs. Data is not translated by the IMS TCP/IP feature.

Network Byte Order

Fixed-point binary integers (used for segment lengths in TRM and RSM) are specified using the TCP/IP network byte ordering convention (big-endian notation). This means that if the high-order byte is stored at address n, the low-order byte is stored at address n+1. (Little-endian notation stores the other way around).

MVS also uses the big-endian convention. Because this is the same as the network convention, IMS TCP/IP MPP's should not need to convert data from little-endian to big-endian notation. If the client uses little-endian notation, it is responsible for the conversion.

End-of-Message Indicator

IMS TCP/IP does not define an End-of-message indicator for explicit-mode messages.

---

11. If the Listener is unable to initiate the transaction, it sends a request-status message (RSM) to the client indicating the reason for failure. Therefore, the client must be prepared to receive that message. It is suggested that a convention be established that the server initiate the conversation by sending an opening message. By following this convention, the client will receive either positive or negative notification of transaction status before initiating application data exchange.

Implicit-Mode Client Logic Flow

When the client requests the services of an implicit-mode client, the protocol is predefined by IMS TCP/IP.

The client requests an IMS MPP by sending the transaction-request message (TRM). (See “Transaction-Request Message Segment (Client to Listener)” on page 43 for the format of the TRM.) The TRM includes the name of the transaction the Listener is to schedule.

If the transaction cannot be scheduled (for example, because of failure to pass the security exit, or because the transaction was stopped by the IMS master terminal operator), the Listener returns the request-status message with an indication of the cause of failure. (See “Request-Status Message Segment” on page 44 for the format of the request-status message).

For implicit-mode applications, the input data stream consists of the TRM, immediately followed by all segments of application data and an end-of message-segment. The Listener uses the TRM contents to schedule the server and then places the TIM and all of the application data on the IMS message queue for retrieval by the Assist module.

Implicit-mode transactions are limited to one multisegment input message and one multisegment output message. In other words, implicit-mode applications cannot enter into conversations.

When the transaction is complete, the IMS Assist module sends a complete-status message (CSMOKY) segment to the client. If the client receives this message, the client can safely assume that the database changes have been committed. If the client doesn't receive this message, the client cannot determine what has happened. The transaction may have completed normally and database changes committed, or the transaction may have failed with database changes backed out. For this reason, clients that work with implicit mode servers should include application logic that, upon failure to receive the CSMOKY message segment, re-establishes contact with IMS and confirms the success of the previously submitted update.

Implicit-Mode Client Call Sequence

The call sequence to be used by an implicit-mode client program is:

Call	Explanation of Function
INITAPI	Open the interface. (Only required for client programs that use MVS TCP/IP Sockets calls).
SOCKET	Obtain a socket descriptor.
CONNECT	Request connection to the IMS Listener port.
WRITE	Send a transaction-request message (TRM).
WRITE	Send server input data formatted as IMS segments
READ	Receive response. • If the request was rejected, a request-status message (RSM) will be received. • If the transaction was scheduled and executed properly, application data will be received.

Thus, logic in the client must test the output message for the characters *REQSTS* to distinguish between application data and a request-status message (RSM).

READ Upon successful completion of the database updates, the Assist module sends a complete-status message (*CSMOKY*) to the client, indicating that the transaction has completed successfully.

If this message is not received, the client must assume that the application failed to complete properly; in this case, a return code of –1 and ERRNO (typically set to 54) will indicate that application failed. The client must take whatever action is appropriate (for example, reschedule the transaction, resynchronize data).

CLOSE Terminate the connection and release the socket resources

Implicit Mode Application Data Stream

Client-to-Server Data Stream

In implicit mode, the client sends the following data stream:

llzz transaction-request message (TRM) *llzz* application data segment 1 *llzz* application data segment 2 (optional) *llzz* ... *llzz* application data segment n (optional) *04zz* end-of-message segment

WHERE:

ll is the length in bytes of this data segment in binary.

Server-to-Client Data Stream

Data received by the client is formatted (by the Assist module) as above. It consists of n segments of application data including the CSM segment, followed by an end-of-message segment.

Implicit-Mode Application Data

Format.

Data exchanged between implicit-mode client and server is transmitted in a format that resembles an IMS message segment. These segments have the following format: [12]

Field	Format	Description
Length	H	Length of the data segment (including this field)
Reserved *(zz)*	CL2	Reserved field
Data	CLn	Client-supplied data

The length field contains the total length of the message in binary. The length (*ll*) includes the length of the *ll* and *zz* fields.

Data Translation.

The IMS Listener tests the initial input data string (the TRM) to determine whether the terminal is transmitting in ASCII. If the terminal is transmitting in ASCII, and the transaction is defined as ***implicit***-mode in the TRANSACTION configuration statement, the Listener translates the ASCII application data into EBCDIC. Note that

12. This example uses Assembler language notation. See Chapter 7 for COBOL and PL/I equivalents.

when data translation takes place, the entire application data portion of the segment is translated from ASCII to EBCDIC, and vice versa; therefore, the segment should contain only printable characters that are common to both character sets. (For example, the EBCDIC cent sign and the ASCII left square bracket are both printable in their respective native environments, but they are not translated because they do not have an equivalent in the other character set.)

End-of-Message Segment.

The last segment in a message (either sent by the client, or received from the server) is indicated by an end-of-message (EOM) segment. (See "End-of-Message Segment (EOM)" on page 45).

- Implicit-mode messages sent by the client are received by the Listener. When the client program sends an EOM segment, the Listener interprets the EOM as an indication that no more message segments are to be received and inserts the segments onto the IMS message queue.
- Implicit-mode messages received by the client are actually written by the Assist module on behalf of the server program. When the server program sends application data to the client (using the ISRT call), the Assist module intercepts the output data and accumulates it in an output buffer. When the server program issues a subsequent GU to the I/O PCB, the Assist module interprets the GU as an indication that the server has inserted the last segment for that message. The Assist module then adds an end-of-message segment to the output data and issues WRITE commands, which transmit the data to the client. (The client program should test for the EOM segment to determine when the last segment of the message has been sent by the server program.)

IMS TCP/IP Message Segment Formats

The client sends or receives several types of message segments whose formats are defined by the Listener and the Assist module.

- Transaction-request message segment (TRM)
- Request-status message segment (RSM)
- Complete-status message segment (CSMOKY)
- End-of-message segment (EOM)

The following paragraphs describe the formats for each of these segments:

Transaction-Request Message Segment (Client to Listener)

To initiate a connection with an IMS server, the client first issues a transaction-request message segment (TRM), which tells the Listener which transaction to schedule.

The format of the transaction-request message segment (TRM) is:

Field	Format	Meaning
TRMLen	H	Length of the segment (in binary) including this field. This field is sent in network byte order.
TRMRsv	CL2	Reserved
TRMId	CL8	Identifying string. Always *TRNREQ*. If the client data stream will be sent in ASCII, the TRMId field should also be transmitted in ASCII because the Listener uses this field to determine whether ASCII to EBCDIC translation is required.

Field	Format	Meaning
TRMTrnCod	CL8	The transaction code (TRANCODE) of the IMS transaction to be started. It must not begin with a / character; it must follow the naming rules for IMS transactions. If the Listener has determined that data will be transmitted in ASCII, it translates the transaction code to EBCDIC before any further processing is done.
TRMUsrDat	XLn	This variable-length field contains client data that is passed directly to the security exit without translation.

Request-Status Message Segment

If a transaction request is accepted, the IMS Listener does not send the request-status message segment; if the transaction request is rejected, the IMS Listener sends a request-status message segment (RSM) to the client. This segment has the following format:

Field	Format	Description
RSMLen	H	Length of message (in binary), including this field.
RSMRsv	CL2	Reserved
RSMId	CL8	Identifying string. Always *REQSTS*. This field is translated to ASCII if the Listener has determined that the client is transmitting in ASCII.
F	Return code, sent in network byte order. Set to nonzero (for example, 4, 8, 12) to indicate an error. The nonzero value is further explained by the reason code (RSMRsnCod).	
RSMRsnCod	F	Reason Code, sent in network byte order. Reason codes 0 — 100 are reserved for use by the IMS Listener. Codes greater than 100 can be assigned by the user-written security exit.

Request-Status Message Reason Codes

If the IMS Listener sends a request-status message (RSM) segment to the client (indicating that it is unable to complete the processing of the client's transaction-request message (TRM), it sets the return and reason code in the RSM.

- If the security exit rejects a transaction request, it sets the return code and reason code, and returns control to the Listener, which sends the request-status message segment to the client.
- If the Listener detects other errors that cause a request to be rejected, it sets a return code of 8 and a reason code from the following list.

  1 The transaction was not defined to the IMS Listener.

2 An IMS error occurred and the transaction was unable to be started.

3 The transaction failed to perform the TAKESOCKET call within the 3 minute timeframe.

4 The input buffer is full as the client has sent more than 32KB of data for an implicit transaction.

5 An AIB error occurred when the IMS Listener tried to confirm if the transaction was available to be started.

6 The transaction is not defined to IMS or is unavailable to be started.

7 The transaction-request message (TRM) segment was not in the correct format.

9 The application data buffer for the Client-to-Server Data Stream contains an invalid value for the data segment length.

100 up
Reason codes of 100 or higher are defined by the user-supplied security exit.

Complete-Status Message Segment

The complete-status message segment is sent by the Assist module to indicate the successful completion of an implicit-mode transaction, including the fact that database updates have been committed. The format of the complete-status message segment is:

Field	Format	Description
Length	H	Length of the data segment (in binary) including this field
CSMRsv	H	Reserved field; must be set to zero
CSMId	CL8	*CSMOKY* This field is translated to ASCII if the client is transmitting in ASCII.

End-of-Message Segment (EOM)

The end-of-message segment is defined as an IMS-type segment (with *llzz* fields) but no application data. Thus, the EOM segment has an *llzz* field of '0400'; 04 is the length of the *llzz* field.

PL/I Coding

PL/I programmers should note that (although the segments exchanged between the Listener and implicit-mode servers resemble IMS segments) the segments are actually sent by TCP/IP socket calls and do not necessarily follow the standard IMS convention for the PL/I language interface. Specifically, the length field in a segment (TRM or RSM), which is passed via a TCP/IP socket call, ***must*** be a halfword (FIXED BIN(15)) and not a fullword.

Chapter 5. How to Write an IMS TCP/IP Server Program

When writing an IMS TCP/IP server program, the programmer must follow conventions established by the IMS Listener; by the IMS Assist module (if the server program uses it); and by the TCP/IP client. This chapter describes the call sequences and input/output formats necessary for communication between a TCP/IP client program and an IMS server program. (See “Chapter 4. How to Write an IMS TCP/IP Client Program” on page 39 for a discussion of client programming).

Server Program Logic Flow —General

An IMS TCP/IP server program is executed in response to a transaction request from a TCP/IP host. The server program can either explicitly issue TCP/IP socket calls, or implicitly issue them through the IMS Assist module. However, the same TCP/IP functions are completed in either case.

The following sections describe the server logic flow for each mode.

Explicit-Mode Server Program Logic Flow

When an explicit-mode server begins execution, the Listener has received the transaction-request message (TRM) from the client and has inserted the transaction-initiation message (TIM) to the IMS message queue. The Listener has also issued a GIVESOCKET call to pass the connection to the server.

The server’s first action is to obtain the TIM from the IMS message queue. This message contains the information needed to issue the INITAPI and TAKESOCKET calls.

Once the server has issued the TAKESOCKET call, the connection is between client and server; the two can now communicate directly using socket READ/WRITE calls. The number of reads/writes, and the format of the data exchanged, is determined by agreement between the two programs.

At the end of processing a client’s request, the application program should follow the IMS DC programming standard of issuing another GU to the IO/PCB. This informs IMS that the database changes should be committed, and that the database buffers should be emptied (flushed).

Note: For this reason, a transaction invoked by a TCP/IP client should be defined (by the IMS-gen TRANSACT macro) as MODE=SNGL.

Explicit-Mode Call Sequence

The suggested call sequence for an explicit-mode server follows. See “Chapter 7. Using the CALL Instruction Application Programming Interface (API)” on page 61 for the call syntax of the socket calls.

Server call	Explanation of Function
CALL CBLTDLI (GU) I/O PCB	Obtain transaction-initiation message (TIM) from IMS message queue.
INITAPI	Initialize the connection with TCP/IP.
	Parameter **Meaning**

	ADSNAME	Server address space (TIMSrvAddrSpc from the TIM)
	SUBTASK	Server task ID (TIMSrvTaskID from the TIM)
	TCPNAME	TCP address space (TIMTCPAddrSpc from the TIM)
TAKESOCKET	Accept the socket from the Listener.	
	Parameter	**Meaning**
	CLIENT.name	Listener address space (TIMLstAddrSpc from the TIM)
	CLIENT.task	Listener task ID (TIMLstTaskID from the TIM)
	SOCRECV	Socket descriptor (TIMSktDesc from the TIM)
	Note that the TAKESOCKET call returns a new socket descriptor which must be used for the rest of the process. (Do not continue to use the descriptor passed by the Listener in TIMSktDesc.)	
READ/WRITE	Exchange application data with the client.	
Database calls	Read/write database records.	
	Note: TCP/IP and database calls can be intermixed.	
GU	Force IMS synchronization point; update the database from the buffers.	
WRITE	Send complete-status message to the client.	
CLOSE	Shut down the socket and release resources associated with it.	
TERMAPI	End processing on the call interface.	

Explicit-Mode Application Data

Format

Other than the initial transaction-initiation message, explicit-mode imposes no restrictions on the format of application data exchanged between client and server.

EBCDIC/ASCII Data Translation

If the TCP/IP host is transmitting ASCII data, explicit-mode servers are responsible for data translation from EBCDIC to ASCII, and vice versa. Data translation is not performed by IMS TCP/IP. (The data translation subroutines (EZACIC04 and EZACIC05), described in “Chapter 7. Using the CALL Instruction Application Programming Interface (API)” on page 61 can be used for this purpose.)

When the conversation is complete, the server should force an IMS commit and close the connection. This causes IMS to complete the database updates. Explicit-mode server logic is responsible for notifying the client of the success or failure of the commit process.

Transaction-Initiation Message Segment

Once the server has been started, the first segment it receives from the message queue is the transaction-initiation message (TIM) segment, which was created by the IMS Listener.

Field	Format	Explanation
TIMLen [13]	H	The length of the transaction-initiation message segment (in binary), including the length of this field. (X'0038')
TIMRsv	H	Reserved field set to zero. (X'0000').
TIMId	CL8	Identifies the message as having been created by the IMS Listener. Always contains the characters *LISTNR*.
TIMLstAddrSpc	CL8	Listener address space name. Used in server TAKESOCKET.
TIMLstTaskId	CL8	Listener task ID. Used in server TAKESOCKET.
TIMSrvAddrSpc	CL8	Server address space name. Used in server INITAPI. Server address space IDs are generated by the Listener and consist of the 2-character prefix specified in the Listener configuration file (Listener statement) followed by a unique 6-character hexadecimal number.
TIMSrvTaskID	CL8	Server task ID. Used in server INITAPI.
TIMSktDesc	H	Contains the descriptor of the socket given by Listener. Used in server TAKESOCKET.
TIMTCPAddrSpc	CL8	The TCP/IP address space name of TCP/IP. Used in INITAPI.
TIMDataType	H	Indicates the data type of the client messages: ASCII(0) or EBCDIC(1).

Program Design Considerations

- Because MVS TCP/IP ends the connection when a server MPP completes, the client has no way of knowing that the database changes have been committed. Therefore, it is suggested that explicit-mode servers send a message to the client confirming the COMMIT before terminating. (Implicit-mode servers send the CSMOKY segment when the database changes have been committed.)

13. If you use PL/I, you must define the LLLL field as a binary fullword.

- When an explicit-mode server issues a ROLB command, the client has no automatic way of knowing that the database updates have been rolled back. It is suggested, therefore, that the server send a message to the client when a rollback call completes.

I/O PCB — Explicit-Mode Server

When an IMS MPP issues a call for IMS TM services (like a GU or an ISRT), IMS returns information about the results of the call in a control block called the I/O program control block (I/O PCB). The contents of the I/O PCB are:

LTERM NAME	Blanks (8 bytes)
RESERVED	X'00' (2 bytes)
STATUS CODE	See below (2 bytes)
DATE/TIME	Undefined (8 bytes)
INPUT MSG. SEQ. #	Undefined (4 bytes)
MESSAGE OUTPUT DESC. NAME	Blanks (8 bytes)
USERID	PSBname of Listener (8 bytes)

Status Codes

The I/O PCB status code is set by IMS in response to the server GU for the TIM. A status code of `bb` indicates successful completion of the GU call. Since the only data explicit-mode servers receive from the message queue is the TIM, the only call issued by the server is a GU, requesting a new TIM. Thus, the only status codes an explicit-mode server should receive are `bb`, which indicates successful completion of the GU; and `QC`, which indicates that there are no more messages on the message queue for that transaction. In response to the `QC` status code, the server program should end normally.

Explicit-Mode Server — PL/I Programming Considerations

PL/I programmers should note that I/O areas used to retrieve IMS segments must follow standard IMS conventions. That is, the length field for the TIM segment must be defined as a fullword (FIXED BIN(31)).

Implicit-Mode Server Program Logic Flow

An implicit-mode server must perform all of the functions previously described for an explicit-mode server (see “Explicit-Mode Server Program Logic Flow” on page 47). However, the IMS Assist module issues the TCP/IP calls on behalf of the server program; consequently, the implicit-mode application programmer need only issue standard IMS Input/Output calls.

Implicit-Mode Server Call Sequence

When writing an implicit-mode program, you must call the IMS Assist module (CBLADLI, PLIADLI, ASMADLI, CADLI, as appropriate for the language you are using) instead of the conventional IMS equivalent (CBLTDLI, PLITDLI, ASMTDLI, CTDLI). This will cause the I/O PCB calls to be intercepted and processed (if necessary) by the Assist module. The Assist module will pass database calls directly to IMS for processing; it will intercept I/O PCB calls and issue the appropriate sockets calls. A sample call sequence (using COBOL syntax) for an implicit-mode server follows:

IMS Server Call	**Resulting Assist Module Function**

CALL CBLADLI (GU) I/O PCB

Issue CALL CBLTDLI (GU) to obtain the (TIM).

CALL CBLADLI (GN) I/O PCB

(optional) Issue CALL CBLTDLI (GN), which returns a subsequent segment of client input data for each call.

CALL CBLADLI [14] Read/write database records. [15]

CALL CBLADLI (ISRT) I/O PCB

Store segments in the sockets output buffer.

CALL CBLADLI (GU) I/O PCB

Issue WRITE to empty output buffers.

Implicit-Mode Application Data

Format.

All data exchanged between the client and an implicit-mode server is formatted into IMS segments. Each data segment has the following format:

Field	Format	Description
Length	H	Length of the data segment (in binary) including this field.
Reserved	H	Reserved field; must be set to zero.
Data	CLn	Application data.

Data Translation.

Translation of input data (when necessary) is done by the Listener. As a result, all data on the IMS message queue is in EBCDIC; output data is translated (when necessary) by the Assist module.

Note that when data translation takes place, the entire application data portion of the segment is translated from ASCII to EBCDIC, and vice versa; therefore, the segment should contain only printable characters common to both character sets. (For example, the EBCDIC cent sign and the ASCII left bracket are both printable in their respective environments but are not translated because they do not have an equivalent in the other character set.)

End-of-Message Segment.

The last segment in a message (either sent by the client, or received from the server) is indicated by an end-of-message (EOM) segment. (See “End-of-Message Segment (EOM)” on page 45).

- Implicit-mode messages sent by the client are received by the Listener and inserted onto the IMS message queue. The end-of-message segment (defined above) indicates to the Listener that there are no more segments to be inserted for this message. (Note that the server program will ***not*** receive the EOM segment; it will receive a QD status code, indicating that there are no more segments for this message.)

14. For database I/O, you can use either CBLTDLI or CBLADLI. The Assist module simply converts database calls from CBLADLI to CBLTDLI.

15. Database PCB and I/O PCB calls can be intermixed.

- Implicit-mode messages to be sent by the server are actually written by the Assist module on behalf of the server program. When the server program sends application data to the client (using the ISRT call), the Assist module intercepts the output data and accumulates it in an output buffer. When the server program issues a subsequent GU to the I/O PCB, the Assist module interprets the GU as an indication that the server has inserted the last segment for that message. The Assist module then adds an end-of-message segment to the output data and issues WRITE commands, which transmit the data to the client. (Note that the server program should ***not*** attempt to insert an EOM segment to the I/O PCB.)

Programming to the Assist Module Interface

Programs written to the Assist module interface are very similar (in terms of I/O calls) to conventional IMS Transaction Manager (TM) MPPs.

- To communicate with IMS TM, use the following calls (depending upon programming language) — CBLADLI, PLIADLI, ASMADLI, or CADLI — instead of CBLTDLI, PLITDLI, ASMTDLI, and CADLI, respectively.
- Use the same parameters as with the IMS TM counterparts.
- The first IMS call to the I/O PCB must be GU. Subsequent IMS calls to the I/O PCB can be GN and/or ISRT (with intervening database calls, as appropriate).
- When the transaction is complete, the server program should issue another GU to the I/O PCB to finalize processing of the present message. If the server program receives a bb status code, (indicating another message has been received for that program), it should loop back and process that message. Note that the Assist module will have closed the previous connection and opened a new connection associated with the new message. When the GU returns a QC status code, no more messages have been received for that program and the program should end.

  A set of one GU, one or more GN calls, and one or more ISRT calls to the I/O PCB (with intervening database calls, as required) constitute a transaction. The Assist module interprets each GU as the start of a new transaction.
- The PURG call cannot be used to indicate end-of-message; the server should not issue PURG calls to the I/O PCB.
- The Assist module GU reads the TIM into the I/O area defined in the server program; consequently, the I/O area you define in the server must be at least 56 bytes in length (the length of the TIM).
- If the server program attempts to insert more than 32KB, the Assist module flags this as an error by terminating processing and returning a status code of `ZZ`.

Implicit-Mode Server PL/I Programming Considerations

PL/I programmers should note that I/O areas passed to the Assist module must follow standard IMS conventions. That is, the length field for a segment must be defined as a fullword (FIXED BIN(31)). This applies to both input and output data segments; however, the actual segment that is received from and sent to the client uses a halfword (FIXED BIN(15)) length field. Thus, the messages exchanged between the client and server are programming-language independent.

Implicit-Mode Server C Language Programming Considerations

The following statements are required in IMS implicit-mode servers written in C language:

```
#pragma runopts(env(IMS),plist(IMS))
#pragma linkage(cadli, OS)
```

This is in addition to the standard requirements for using C language programs in IMS.

I/O PCB Implicit-Mode Server

When an IMS MPP issues a call for IMS TM services (like a GU or an ISRT), IMS returns information about the results of the call in a control block called the I/O program control block (I/O PCB). When using the Assist module, the contents of the I/O PCB are:

LTERM NAME	Blanks (8 bytes)
RESERVED	See below (2 bytes)
STATUS CODE	See below (2 bytes)
DATE/TIME	Undefined (8 bytes)
INPUT MSG. SEQ. #	Undefined (4 bytes)
MESSAGE OUTPUT DESC. NAME	Blanks (8 bytes)
USERID	PSBname of Listener (8 bytes)

Status Codes

The I/O PCB status code is set by IMS in response to the IMS calls that the Assist module makes on behalf of the server. For example, GU and GN calls usually result in `bb`, `QC`, or `QD` status codes. However, when the Assist module detects a TCP/IP error, it sets the status code field of the I/O PCB to `ZZ` with further information about the error in the **reserved** field of thE I/O PCB. This field should be initially tested as a signed, fixed binary halfword:

- If the halfword is positive, then a socket error has occurred, and the field should continue to be treated as a signed fixed binary halfword. The field contains the 2 low-order bytes from the ERRNO resulting from the socket call. (See “Appendix A. Return Codes” on page 193).
- If the halfword is negative, then an IMS or other type of error has occurred, and the field should be treated as a fixed-length, 2-byte character string containing one of the following:

Code	Meaning
EA	A call that used the AIB interface to determine the I/O PCB address failed.
EB	The output buffer is full. An attempt was made to insert (ISRT) more than 32KB (including the segment length and reserved bytes) to be sent to the client.
EC	A `QD` status code was received in response to a GU or ROLB call when attempting to retrieve the first segment of data after the transaction-initiation message (TIM) segment. This implies that the client sent only the TIM segment followed by an end-of-message segment with no actual data segments.

Chapter 6. How to Customize and Operate the IMS Listener

The IMS Listener is an IMS batch message program (BMP) whose main purpose is to validate connection requests from TCP/IP clients and to schedule IMS message processing programs (MPP) servers.

This chapter describes the IMS Listener and the user-written security exit that can be used to validate incoming transaction requests.

How to Start the IMS Listener

The IMS Listener is executed as an MVS 'started task' using job control language (JCL) statements. Copy the sample job in the *hlq*.SEZAINST(EZAIMSJL) to your system or recognized PROCLIB and modify it to suit your conditions. Below is a sample of the JCL needed for the Listener BMP. Note the STEPLIB statements pointing to MVS TCP/IP. Also note the `EZAIMSJL G.LSTNCFG DD` statement points to the Listener configuration file. For more information on configuring the IMS Listener, see "The IMS Listener Configuration File" on page 56.

```
.*
//EZAIMSJL   PROC MBR=EZAIMSLN,PSB=EZAIMSLN,IMSID=IMS,CFG=TCPIMS,SOUT=A
//*
//LISTENER   EXEC PROC=IMSBATCH,MBR=&MBR,SOUT=&SOUT,IMSID=&IMSID,
//           PSB=&PSB,CPUTIME=1440
//G.STEPLIB  DD DSN=IMSVS31.&SYS2.RESLIB,DISP=SHR
//           DD DSN=IMSVS31.&SYS2.PGMLIB,DISP=SHR
//           DD DSN=TCPIP.SEZALINK,DISP=SHR
//           DD DSN=TCPIP.SEZATCP,DISP=SHR
//G.LSTNCFG  DD DSN=TCPIP.LSTNCFG(&CFG.),DISP=SHR
//G.SYSPRINT DD SYSOUT=&SOUT,DCB=(LRECL=137,RECFM=VBA,BLKSIZE=1374),
//             SPACE=(141,(2500,100),RLSE,,ROUND)
```

Figure 11. Sample JCL for starting the IMS Listener

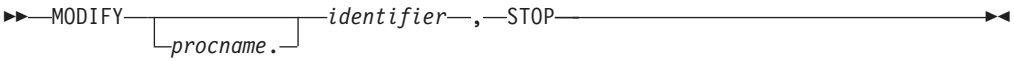
Once you have configured your JCL, you can start the Listener using the MVS START command. The basic syntax and parameters of this command are given below.

procname
The name of the cataloged procedure that defines the IMS Listener job to be started.

identifier
A user-determined name which, with the procedure name, (*procname*) uniquely identifies the started job. This name can be up to 8 characters long with the first character being alphabetic. If the identifier is omitted, MVS automatically uses the procedure name as the identifier.

How to Stop the IMS Listener

The Listener is normally ended by issuing an MVS MODIFY command. The syntax of this command and a description of the parameters is given below.



procname
The name of the cataloged procedure that was used to start the Listener. This is only required if an identifier that was different from *procname* was specified with the START command when the Listener was started.

identifier
The user-determined identifier used on the START command when the Listener was started. If an explicit identifier was not specified (on the START command), MVS automatically uses the procedure name (*procname*) on the START command as the default identifier.

stop
Stops the Listener.

On receipt of a MODIFY command, the Listener closes the socket bound to the listening port so that no new requests can be accepted. It ends once all other sockets have been closed following acceptance of each socket by the corresponding server.

As a BMP, the Listener can be forcibly ended by issuing the IMS STOP REGION command with the ABDUMP option.

The IMS Listener Configuration File

The IMS Listener obtains startup parameters from a configuration file. In Figure 11 on page 55 the `EZAIMSJL G.LSTNCFG DD` statement points to the Listener configuration file. This statement will be in the JCL sample you customize.

The configuration file contains three types of statements which must appear in the following order:
1. TCPIP statement
2. LISTENER statement
3. TRANSACTION statements

The following describes each of the configuration statements and their respective parameters.

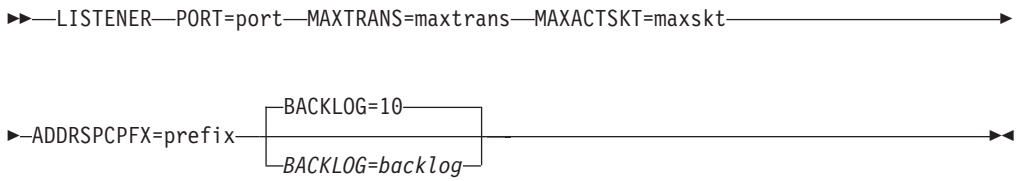
TCPIP Statement

Description: This statement is required and is used to specify the name of the TCP/IP address space.

ADDRSPC= *name*
Specifies the name of the TCP/IP address space. The name can be 1 to 8 characters long, consisting of the numbers 0–9, the letters A–Z, and the characters $, @, and #.

LISTENER Statement

Description: This statement is required. It is used to specify configuration information used by the IMS Listener.



PORT= *port*
Port number that the Listener binds to for connection requests. Use an integer between 0 and 65 535, inclusive.

MAXTRANS= *maxtrans*
The maximum number of TRANSACTION statements to be processed in the configuration file. Use an integer between 1 and 32 767, inclusive.

MAXACTSKT= *maxskt*
The maximum number of sockets the Listener can have open awaiting an MPP TAKESOCKET at one time. This value is an integer from 1 to 2000, inclusive. The number includes the socket bound to the port through which it accepts incoming requests.

ADDRSPCPFX= *prefix*
One or two characters (consisting of the numbers 0–9, the letters A–Z, and the characters $, @, and #) used in generating unique identifiers for started IMS transactions.

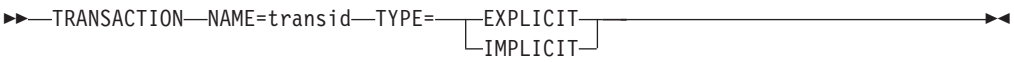
BACKLOG= *backlog*
This parameter is optional and is used to specify the length of the backlog queue maintained in TCP/IP for connection requests that have not yet been assigned sockets by the Listener. Use an unsigned number from 1 to 32 767 inclusive. The default value is 10.

TRANSACTION Statement

Description: This statement specifies which transactions can be started by the Listener. One statement is required for each transaction that can be initiated by a TCP/IP-connected client.

Note that the transactions named here are subject to limitations:

- They must be defined to IMS as MODE=SNGL in the IMS TRANSACT macro; this will ensure that the database buffers are emptied (flushed) to direct access storage when the second and subsequent GU calls are issued.
- They must not be IMS conversational transactions.
- They cannot name transactions that are executed in a remote Multiple Systems Coupling (MSC) environment.
- They must not use Message Format Services for messages to the client.



NAME= *transid*
The name of an IMS transaction that is designed to interact with a TCP/IP-connected program. This parameter must be 1 to 8 characters long, containing alphanumeric characters, or the characters @, $, and #.

TYPE=
This parameter specifies whether the transaction uses the IMS Assist module. It must specify either EXPLICIT or IMPLICIT.

The IMS Listener Security Exit

The IMS Listener includes an exit (IMSLSECX), which can be programmed by the user to perform a security check on the incoming transaction-request. This Listener exit can be designed to validate the contents of the UserData field in the transaction request message.

To use the user-supplied security exit, you must define an entry point named IMSLSECX. If a module with this name is link-edited with the Listener (EZAIMSLN) load module, the security exit is called as part of transaction verification. The security exit is called using standard MVS linkage with register 1 (R1) pointing to the parameter list (described below). Note that the security exit must have the attribute AMODE(31).

The exit returns 2 indicators: a return code and a reason code. The Listener uses the return code to determine whether to honor the request. Both the return code and the reason code are passed back to the client. Data passed in the UserData field is not translated from ASCII to EBCDIC; this translation is the responsibility of the security exit. (EZACIC05 and EZACIC04 can be used to accomplish translation between ASCII and EBCDIC. See the chapter on CALL instructions in *z/OS Communications Server: IP Application Programming Interface Guide* for a description of these utilities.)

The format of the data passed to the security exit is:

Field	Format	Description
IpAddr	F	The address of a fullword containing the client's IP address.
Port	H	The address of a halfword containing the client's port number.
TransNam	CL8	The address of an 8-character string defining the name of the requested transaction.
DataType	H	The address of a halfword containing the data type (0 if ASCII or 1 if EBCDIC).
DataLen	F	The address of a fullword containing the length of the user data.
Userdata	XLn	The address of the user-supplied data.
RetnCode	F	The address of a fullword set by the security exit to indicate the return status. Set to nonzero (4, 8, 12, ...) to indicate an error.

Field	Format	Description
ReasnCode	F	The address of a fullword set by the security exit as a reason code associated with the value of the return code. Reason codes 0–100 are reserved for use by the Listener. The security exit can use reason codes greater than 100.

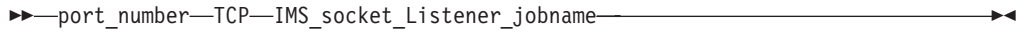
TCP/IP Services Definitions

To run IMS, you need to modify the *tcpip*.PROFILE.TCPIP data set and the *hlq*.TCPIP.DATA [16] data set that are part of the TCP/IP Services configuration file.

The hlq.PROFILE.TCPIP Data Set

You define the IMS socket Listener to TCP/IP on MVS in the *hlq*.PROFILE.TCPIP data set. In it, you must provide entries for the IMS socket Listener started task name in the PORT statement, as shown in Figure 12 on page 60.

The format for the PORT statement is:



As an example, assume you want to define two different IMS control regions. Create a different line for each port that you want to reserve. Figure 12 on page 60 shows 2 entries, allocating port number 4000 for SERVA, and port number 4001 for SERVB. SERVA and SERVB are the names of the IMS socket Listener started task names.

These 2 entries reserve port 4000 for exclusive use by SERVA and port 4001 for exclusive use by SERVB. The Listener transactions for SERVA and SERVB should be bound to ports 4000 and 4001 respectively. Other applications that want to access TCP/IP on MVS are prevented from using these ports.

Ports that are not defined in the PORT statement can be used by any application, including SERVA and SERVB if they need other ports.

16. Note: in this book, the abbreviation *hlq* stands for an installation-dependent *high level qualifier* which you must supply.

```
;
; hlq.PROFILE.TCPIP
; ==================
;
; This is a sample configuration file for the TCPIP address space.
; For more information about this file, see "Configuring the TCPIP
; Address Space" and "Configuring the Telnet Server" in the Planning and
; Customization Manual.
       ..........
       ..........
; ---------------------------------------------------------------------
; Reserve PORTs for the following servers.
;
; NOTE:  A port that is not reserved in this list can be used by
;        any user.  If you have TCP/IP hosts in your network that
;        reserve ports in the range 1-1023 for privileged
;        applications, you should reserve them here to prevent users
;        from using them.
PORT
       ..........
       ..........
  4000 TCP SERVA            ; IMS Port for SERVA
  4001 TCP SERVB            ; IMS Port for SERVB
```

Figure 12. Definition of the TCP/IP Profile

The hlq.TCPIP.DATA Data Set

For IMS, you do not have to make any extra entries in *hlq*.TCPIP.DATA. However, you need to check the TCPIPJOBNAME parameter that was entered during TCP/IP Services setup. This parameter is the name of the started procedure used to start the TCP/IP MVS address space. This must match the job name in the Listener configuration file TCPIP statement, as described in “TCPIP Statement” on page 56. In the example below, TCPIPJOBNAME is set to TCPV3. The default name is TCPIP.

```
;*********************************************************************
;                                                                    *
;   Name of Data Set:      hlq.TCPIP.DATA                            *
;                                                                    *
;   This data, TCPIP.DATA, is used to specify configuration          *
;   information required by TCP/IP client programs.                  *
;                                                                    *
;*********************************************************************
; TCPIPJOBNAME specifies the name of the started procedure which was
; used to start the TCP/IP address space.    TCPIP is the default.
;
TCPIPJOBNAME TCPV3
       ..........
       ..........
       ..........
```

Figure 13. The TCPIPJOBNAME Parameter in the DATA Data Set

Chapter 7. Using the CALL Instruction Application Programming Interface (API)

This chapter describes the CALL Instruction API and includes the following topics:

Environmental Restrictions and Programming Requirements

The following restrictions apply to both the Macro Socket API and the Callable Socket API:

- SRB mode

  These APIs may only be invoked in TCB mode (task mode).
- Cross-memory mode

  These APIs may only be invoked in a non-cross-memory environment (PASN=SASN=HASN).
- Functional Recovery Routine (FRR)

  Do not invoke these APIs with an FRR set. This will cause system recovery routines to be bypassed and severely damage the system.
- Locks

  No locks should be held when issuing these calls.
- INITAPI/TERMAPI macros

  The INITAPI/TERMAPI macros must be issued under the same task.
- Storage

  Storage acquired for the purpose of containing data returned from a socket call must be obtained in the same key as the application program status word (PSW) at the time of the socket call.
- Nested socket API calls

  You cannot issue nested API calls within the same task. That is, if a request block (RB) issues a socket API call and is interrupted by an interrupt request block (IRB) in an STIMER exit, any additional socket API calls that the IRB attempts to issue are detected and flagged as an error.
- Addressability mode (Amode) considerations

  The EZASMI macro API may be invoked while the caller is in either 31-bit or 24-bit Amode. However, if the application is running in 24-bit addressability mode at the time of the call, all addresses of parameters passed by the application must be addressable in 31-bit Amode. This implies that even if the addresses being passed reside in storage below the 16 MB line (and therefore addressable by 24-bit Amode programs) the high-order byte of these addresses needs to be 0.
- Use of UNIX® System Services

Address spaces using the EZASMI API should not use any UNIX System Services facilities. Doing so can yield unpredictable results.

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 System/370 Assembler language. The format and parameters are described for each socket call.

For more information about sockets, refer to the *UNIX Programmer's Reference Manual*.

Notes:

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

   ```
   DCL EZASOKET ENTRY OPTIONS(RETCODE,ASM,INTER) EXT;
   ```

4. A C run-time library is required when you use the GETHOSTBYADDR or GETHOSTBYNAME call.
5. The entry point for the CICS Sockets Extended module (EZASOKET) is within the EZACICAL module. Therefore EZACICAL should be included explicitly in your link-edit JCL. If not included, you could experience problems, such as the CICS region waiting for the socket calls to complete.

Understanding COBOL, Assembler, and PL/1 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. It corresponds to the value returned by the tcperror() function in C.

RETCODE
A fullword binary variable containing a code returned by the EZASOKET call. This value corresponds to the normal return value of a C function.

Assembler Language Call Format

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

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

PL/1 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. It corresponds to the value returned by the tcperror() function in C.

RETCODE
A fullword binary variable containing a code returned by the EZASOKET call. This value corresponds to the normal return value of a C function.

Converting Parameter Descriptions

The parameter descriptions in this chapter are written using the VS COBOL II PIC language syntax and conventions, but you should use the syntax and conventions that are appropriate for the language you want to use.

Figure 14 on page 64 shows examples of storage definition statements for COBOL, PL/1, and assembler language programs.

```
VS COBOL II PIC

  PIC S9(4) BINARY                  HALFWORD BINARY VALUE
  PIC S9(8) BINARY                  FULLWORD BINARY VALUE
  PIC   X(n)                        CHARACTER FIELD OF N BYTES

COBOL PIC

  PIC S9(4) COMP                    HALFWORD BINARY VALUE
  PIC S9(8) COMP                    FULLWORD BINARY VALUE
  PIC   X(n)                        CHARACTER FIELD OF N BYTES

PL/1 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 14. Storage Definition Statement Examples

Diagnosing Problems in Applications Using the CALL Instruction API

TCP/IP provides a trace facility that can be helpful in diagnosing problems in applications using the CALL instruction API. The trace is implemented using the TCP/IP Component Trace (CTRACE) SOCKAPI trace option. The SOCKAPI trace option allows all Call instruction socket API calls issued by an application to be traced in the TCP/IP CTRACE. The SOCKAPI trace records include information such as the type of socket call, input, and output parameters and return codes. This trace can be helpful in isolating failing socket API calls and in determining the nature of the error or the history of socket API calls that may be the cause of an error. For more information on the SOCKAPI trace option, refer to *z/OS Communications Server: IP Diagnosis*.

Error Messages and Return Codes

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

For information about error codes that are returned by TCP/IP, see “Appendix A. Return Codes” on page 193.

Code CALL Instructions

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

ACCEPT

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

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

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

Notes:

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 15 on page 66 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 15. ACCEPT Call Instructions Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

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 A socket address structure that contains the client’s socket address.

FAMILY
A halfword binary field specifying the addressing family. The call returns the value 2 for AF_INET.

PORT A halfword binary field that is set to the client’s port number.

IP-ADDRESS
A fullword binary field that is set to the 32-bit internet address, in network-byte-order, of the client’s host machine.

RESERVED
Specifies 8 bytes of binary zeros. This field is required, but not used.

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix A. Return Codes” on page 193 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.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 16 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 16. BIND Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

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 Specifies the socket address structure for the socket that is to be bound.

FAMILY
A halfword binary field specifying the addressing family. The value is always set to 2, indicating AF_INET.

PORT A halfword binary field that is set to the port number to which you want the socket to be bound.

Note: If PORT is set to 0 when the call is issued, the system assigns the port number for the socket. The application can call the GETSOCKNAME macro after the BIND macro to discover the assigned port number.

IP-ADDRESS
A fullword binary field that is set to the 32-bit internet address (network byte order) of the socket to be bound.

RESERVED
Specifies an 8-byte character field that is required but not used.

Parameter Values Returned to the Application

ERRNO
A fullword binary field. If RETCODE is negative, this field contains an error number. See “Appendix A. Return Codes” on page 193 for information about ERRNO return codes.

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

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

CLOSE

The CLOSE call performs the following functions:

- The CLOSE call shuts down a socket and frees all resources allocated to it. If the socket refers to an open TCP connection, the connection is closed.
- The CLOSE call is also issued by a concurrent server after it gives a socket to a child server program. After issuing the GIVESOCKET and receiving notification that the client child has successfully issued a TAKESOCKET, the concurrent server issues the close command to complete the passing of ownership. In high-performance, transaction-based systems the timeout associated with the CLOSE call can cause performance problems. In such systems you should consider the use of a SHUTDOWN call before you issue the CLOSE call. See “SHUTDOWN” on page 129 for more information.

  Notes:

  1. If a stream socket is closed while input or output data is queued, the TCP connection is reset and data transmission may be incomplete. The SETSOCKET 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 125.
  2. A concurrent server differs from an iterative server. An iterative server provides services for one client at a time; a concurrent server receives connection requests from multiple clients and creates child servers that actually serve the clients. When a child server is created, the concurrent

server obtains a new socket, passes the new socket to the child server, and then dissociates itself from the connection. The CICS Listener is an example of a concurrent server.

3. After an unsuccessful socket call, a close should be issued and a new socket should be opened. An attempt to use the same socket with another call results in a nonzero return code.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 17 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 17. CLOSE Call Instruction Example

For equivalent PL/I and assembler language declarations, see "Converting Parameter Descriptions" on page 63.

Parameter Values Set by the Application

SOC-FUNCTION
A 16-byte field containing CLOSE. Left-justify the field and pad it on the right with blanks.

S A halfword binary field containing the descriptor of the socket to be closed.

Parameter Values Returned to the Application

ERRNO
A fullword binary field. If RETCODE is negative, this field contains an error number. See "Appendix A. Return Codes" on page 193 for information about ERRNO return codes.

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

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

CONNECT

The CONNECT call is issued by a client to establish a connection between a local socket and a remote socket.

Stream Sockets

For stream sockets, the CONNECT call is issued by a client to establish connection with a server. The call performs two tasks:

- It completes the binding process for a stream socket if a BIND call has not been previously issued.
- It attempts to make a connection to a remote socket. This connection is necessary before data can be transferred.

UDP Sockets

For UDP sockets, a CONNECT call need not precede an I/O call, but if issued, it allows you to send messages without specifying the destination.

The call sequence issued by the client and server for stream sockets is:

1. The *server* issues BIND and LISTEN to create a passive open socket.
2. The *client* issues CONNECT to request the connection.
3. The *server* accepts the connection on the passive open socket, creating a new connected socket.

The blocking mode of the CONNECT call conditions its operation.

- If the socket is in blocking mode, the CONNECT call blocks the calling program until the connection is established, or until an error is received.
- If the socket is in nonblocking mode, the return code indicates whether the connection request was successful.
  - A 0 RETCODE indicates that the connection was completed.
  - A nonzero RETCODE with an ERRNO of 36 (EINPROGRESS) indicates that the connection is not completed, but since the socket is nonblocking, the CONNECT call returns normally.

  The caller must test the completion of the connection setup by calling SELECT and testing for the ability to write to the socket.

The completion cannot be checked by issuing a second CONNECT. For more information, see “SELECT” on page 112.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 61.
ASC mode:	Primary address space control (ASC) mode.

Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 18 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 18. CONNECT Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

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 A structure that contains the socket address of the target to which the local, client socket is to be connected.

FAMILY
A halfword binary field specifying the addressing family. The value must be 2 for AF_INET.

PORT A halfword binary field that is set to the server’s port number in network byte order. For example, if the port number is 5000 in decimal, it is stored as X'1388' in hex.

IP-ADDRESS
A fullword binary field that is set to the 32-bit internet address of the server’s host machine in network byte order. For example, if the internet address is 129.4.5.12 in dotted decimal notation, it would be represented as ’8104050C’ in hex.

RESERVED
Specifies an 8-byte reserved field. This field is required, but is not used.

Parameter Values Returned to the Application

ERRNO

A fullword binary field. If RETCODE is negative, this field contains an error number. See “Appendix A. Return Codes” on page 193 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 95 for another way to control a socket’s blocking mode.

Values for command that are supported by the UNIX Systems Services fcntl callable service will also be accepted. Refer to the *OpenEdition MVS Programming: Assembler Callable Services Reference* publication for more information.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 19 on page 73 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 19. FCNTL Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

Parameter Values Set by the Application

SOC-FUNCTION
A 16-byte character field containing FCNTL. The field is left-justified and padded on the right with blanks.

S A halfword binary number specifying the socket descriptor for the socket that you want to unblock or query.

COMMAND
A fullword binary number with the following values:

Value	Description
3	Query the blocking mode of the socket.
4	Set the mode to blocking or nonblocking for the socket.

REQARG
A fullword binary field containing a mask that TCP/IP uses to set the FNDELAY flag.

- If COMMAND is set to 3 ('query') the REQARG field should be set to 0.
- If COMMAND is set to 4 ('set')
  - Set REQARG to 4 to turn the FNDELAY flag on. This places the socket in nonblocking mode.
  - Set REQARG to 0 to turn the FNDELAY flag off. This places the socket in blocking mode.

Parameter Values Returned to the Application

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix A. Return Codes” on page 193 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).

GETCLIENTID

GETCLIENTID call returns the identifier by which the calling application is known to the TCP/IP address space in the calling program. The CLIENT parameter is used in the GIVESOCKET and TAKESOCKET calls. See “GIVESOCKET” on page 91 for a discussion of the use of GIVESOCKET and TAKESOCKET calls.

Do not be confused by the terminology; when GETCLIENTID is called by a server, the identifier of the *caller* (not necessarily the *client*) is returned.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 20 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 20. GETCLIENTID Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

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 caller's domain. For TCP/IP the value is set to 2 for AF_INET.

NAME An 8-byte character field set to the caller's address space name.

TASK An 8-byte character field set to the task identifier of the caller.

RESERVED
Specifies 20-byte character reserved field. This field is required, but not used.

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See "Appendix A. Return Codes" on page 193 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.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 21 on page 76 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 21. GETHOSTBYADDR Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

Parameter Values Set by the Application

SOC-FUNCTION

A 16-byte character field containing 'GETHOSTBYADDR'. The field is left-justified and padded on the right with blanks.

HOSTADDR

A fullword binary field set to the internet address (specified in network byte order) of the host whose name is being sought. See “Appendix A. Return Codes” on page 193 for information about ERRNO return codes.

Parameter Values Returned to the Application

HOSTENT

A fullword containing the address of the HOSTENT structure.

RETCODE

A fullword binary field that returns one of the following:

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

GETHOSTBYADDR returns the HOSTENT structure shown in Figure 22 on page 77.

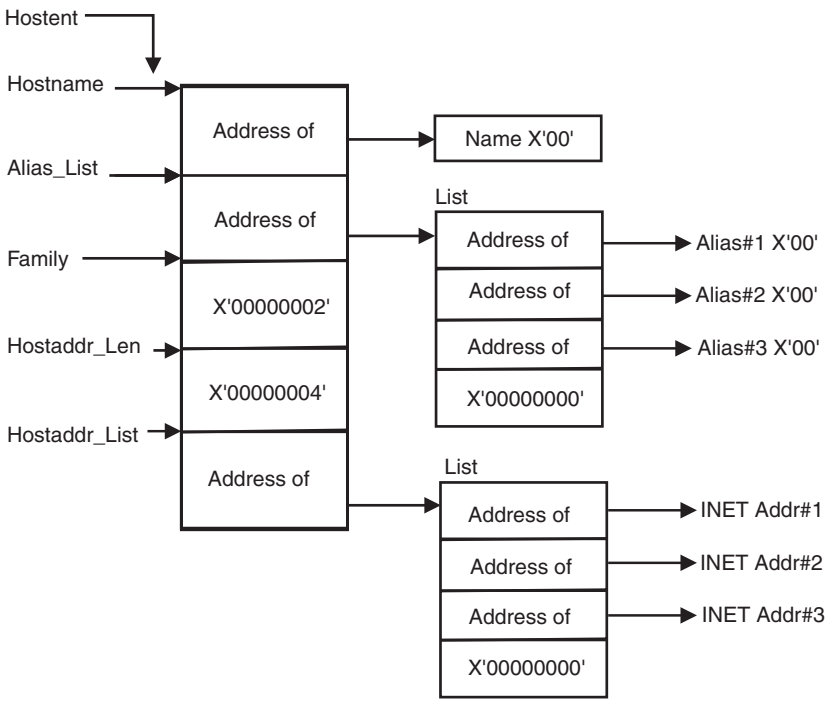


Figure 22. HOSTENT Structure Returned by the GETHOSTBYADDR Call

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'.
- The value returned in the FAMILY field is always 2 for AF_INET.
- The length of the host internet address returned in the HOSTADDR_LEN field is always 4 for AF_INET.
- The address of a list of addresses that point to the host internet addresses returned by the call. The list is ended by the pointer X'00000000'. If the call cannot be resolved, the HOSTENT structure contains the ERRNO 10214.

The HOSTENT structure uses indirect addressing to return a variable number of alias names and internet addresses. If you are coding in PL/1 or assembler language, this structure can be processed in a relatively straight-forward manner. If you are coding in COBOL, this structure may be difficult to interpret. You can use the subroutine EZACIC08 to simplify interpretation of the information returned by the GETHOSTBYADDR and GETHOSTBYNAME calls. For more information about EZACIC08, see “EZACIC08” on page 142.

GETHOSTBYNAME

The GETHOSTBYNAME call returns the alias name and the internet address of a host whose domain name is specified in the call. A given TCP/IP host can have multiple alias names and multiple host internet addresses.

TCP/IP tries to resolve the host name through a name server, if one is present. If a name server is not present, the system searches the HOSTS.SITEINFO data set until a matching host name is found or until an EOF marker is reached.

Notes:

1. HOSTS.LOCAL, HOSTS.ADDRINFO, and HOSTS.SITEINFO are described in *z/OS Communications Server: IP Configuration Reference*.
2. The C run-time libraries are required when GETHOSTBYNAME is issued by your program.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 23 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 23. GETHOSTBYNAME Call Instruction Example

For equivalent PL/I and assembler language declarations, see "Converting Parameter Descriptions" on page 63.

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. Any trailing

blanks will be removed from the specified name prior to trying to resolve it to an IP address. This call returns the address of the HOSTENT structure for this name.

Parameter Values Returned to the Application

HOSTENT

A fullword binary field that contains the address of the HOSTENT structure.

RETCODE

A fullword binary field that returns one of the following:

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

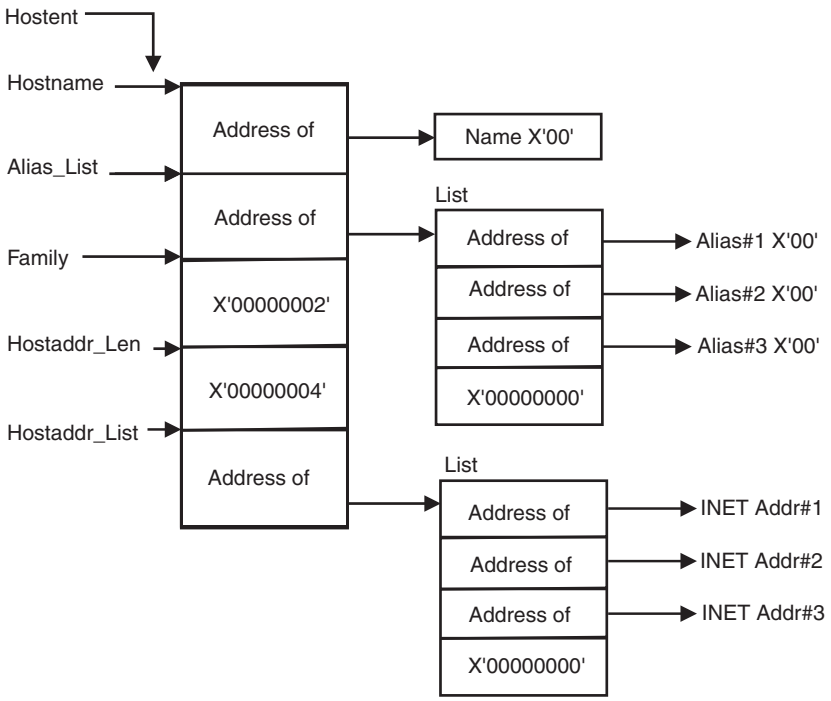


Figure 24. HOSTENT Structure Returned by the GETHOSTYBYNAME Call

GETHOSTBYNAME returns the HOSTENT structure shown in Figure 24. This structure contains:

- The address of the host name that is returned by the call. The name length is variable and is ended by X'00'.
- The address of a list of addresses that point to the alias names returned by the call. This list is ended by the pointer X'00000000'. Each alias name is a variable length field ended by X'00'.
- The value returned in the FAMILY field is always 2 for AF_INET.
- The length of the host internet address returned in the HOSTADDR_LEN field is always 4 for AF_INET.
- The address of a list of addresses that point to the host internet addresses returned by the call. The list is ended by the pointer X'00000000'. If the call cannot be resolved, the HOSTENT structure contains the ERRNO 10214.

The HOSTENT structure uses indirect addressing to return a variable number of alias names and internet addresses. If you are coding in PL/1 or assembler language, this structure can be processed in a relatively straight-forward manner. If you are coding in COBOL, this structure may be difficult to interpret. You can use the subroutine EZACIC08 to simplify interpretation of the information returned by the GETHOSTBYADDR and GETHOSTBYNAME calls. For more information about EZACIC08, see “EZACIC08” on page 142.

GETHOSTID

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 25 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 25. GETHOSTID Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

Parameter Values Set by the Application

SOC-FUNCTION

A 16-byte character field containing 'GETHOSTID'. The field is left-justified and padded on the right with blanks.

RETCODE

Returns a fullword binary field containing the 32-bit internet address of the host. There is no ERRNO parameter for this call.

GETHOSTNAME

The GETHOSTNAME call returns the domain name of the local host.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 26 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 26. GETHOSTNAME Call Instruction Example

For equivalent PL/I and assembler language declarations, see "Converting Parameter Descriptions" on page 63.

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 Services allows a maximum length of 24 characters. The internet standard is a maximum name length of 255 characters. The actual length of the NAME field is found in NAMELEN.

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See "Appendix A. Return Codes" on page 193 for information about ERRNO return codes.

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

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

GETIBMOPT

The GETIBMOPT call returns the number of TCP/IP images installed on a given MVS system and their status, versions, and names.

Note: Images from pre-V3R2 releases of TCP/IP Services are excluded. The GETIBMOPT call is not meaningful for pre-V3R2 releases. With this information, the caller can dynamically choose the TCP/IP image with which to connect by using the INITAPI call. The GETIBMOPT call is optional. If it is not used, follow the standard method to determine the connecting TCP/IP image:

- Connect to the TCP/IP specified by TCPIPJOBNAME in the active TCPIP.DATA file.
- Locate TCPIP.DATA using the search order described in *z/OS Communications Server: IP Configuration Reference*.

For detailed information about the standard method, refer to *z/OS Communications Server: IP Migration*.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 27 on page 83 shows an example of GETIBMOPT call instructions.

```
 WORKING STORAGE
     01  SOC-FUNCTION    PIC X(16)  VALUE IS 'GETIBMOPT'.
     01  COMMAND         PIC 9(8)   BINARY VALUE IS 1.
     01  BUF.
         03  NUM-IMAGES  PIC 9(8) COMP.
         03  TCP-IMAGE   OCCURS 8 TIMES.
             05  TCP-IMAGE-STATUS  PIC 9(4) BINARY.
             05  TCP-IMAGE-VERSION PIC 9(4) BINARY.
             05  TCP-IMAGE-NAME    PIC X(8)
     01  ERRNO           PIC 9(8)   BINARY.
     01  RETCODE         PIC S9(8)  BINARY.

PROCEDURE

    CALL 'EZASOKET' USING SOC-FUNCTION COMMAND BUF ERRNO RETCODE.
```

Figure 27. GETIBMOPT Call Instruction Example

Parameter Values Set by the Application

SOC-FUNCTION
A 16-byte character field containing GETIBMOPT. The field is left-justified and padded on the right with blanks.

COMMAND A value or the address of a fullword binary number specifying the command to be processed. The only valid value is 1.

Parameter Values Returned to the Application

BUF A 100-byte buffer into which each active TCP/IP image status, version, and name are placed.

On successful return, these buffer entries contain the status, names, and versions of up to eight active TCP/IP images. The following layout shows the BUF field upon completion of the call.

The NUM_IMAGES field indicates how many entries of TCP_IMAGE are included in the total BUF field. If the NUM_IMAGES returned is 0, there are no TCP/IP images present.

The status field can have a combination of the following information:

Status Field	Meaning
X'8xxx'	Active
X'4xxx'	Terminating
X'2xxx'	Down
X'1xxx'	Stopped or stopping

Note: In the above status fields, *xxx* is reserved for IBM use and can contain any value.

When the status field is returned with a combination of Down and Stopped, TCP/IP abended. Stopped, when returned alone, indicates that TCP/IP was stopped.

The version field is:

Version	Field
TCP/IP V3R2 for MVS	X'0302'
TCP/IP eNetwork CS V2R5	X'0304'
TCP/IP SecureWay® CS V2R8	X'0308'
TCP/IP OS/390® CS V2R10	X'0510'
TCP/IP z/OS CS V1R2	X'0612'

The name field is the PROC name, left-justified, and padded with blanks.

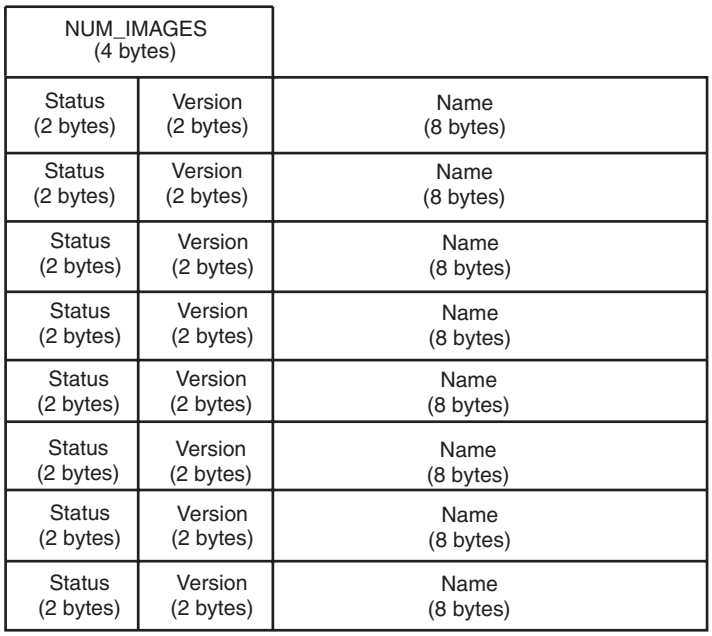


Figure 28. Example of Name Field

ERRNO

A fullword binary field. If RETCODE is negative, this contains an error number. See “Appendix A. Return Codes” on page 193 for information about ERRNO return codes.

RETCODE

A fullword binary field with the following values:

Value	Description
–1	Call returned error. See ERRNO field.
0	Successful call.

GETPEERNAME

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 29 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 29. GETPEERNAME Call Instruction Example

For equivalent PL/I and assembler language declarations, see "Converting Parameter Descriptions" on page 63.

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 A structure to contain the peer name. The structure that is returned is the socket address structure for the remote socket that is connected to the local socket specified in field S.

FAMILY
A halfword binary field containing the connection peer's addressing family. The call always returns the value 2, indicating AF_INET.

PORT A halfword binary field set to the connection peer's port number.

IP-ADDRESS
A fullword binary field set to the 32-bit internet address of the connection peer's host machine.

RESERVED
Specifies an 8-byte reserved field. This field is required, but not used.

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See "Appendix A. Return Codes" on page 193 for information about ERRNO return codes.

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

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

GETSOCKNAME

The GETSOCKNAME call returns the address currently bound to a specified socket. If the socket is not currently bound to an address, the call returns with the FAMILY field set, and the rest of the structure set to 0.

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 30 on page 87 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 30. GETSOCKNAME Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

Parameter Values Set by the Application

SOC-FUNCTION
A 16-byte character field containing GETSOCKNAME. The field is left-justified and padded on the right with blanks.

S A halfword binary number set to the descriptor of a local socket whose address is required.

Parameter Values Returned to the Application

NAME Specifies the socket address structure returned by the call.

FAMILY
A halfword binary field containing the addressing family. The call always returns the value 2, indicating AF_INET.

PORT A halfword binary field set to the port number bound to this socket. If the socket is not bound, 0 is returned.

IP-ADDRESS
A fullword binary field set to the 32-bit internet address of the local host machine.

RESERVED
Specifies 8 bytes of binary zeros. This field is required but not used.

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix A. Return Codes” on page 193 for information about ERRNO return codes.

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

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

GETSOCKOPT

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

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 31 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.
    01 TCP-NODELAY-VAL PIC 9(10) COMP VALUE 2147483649.
    01 TCP-NODELAY-REDEF REDEFINES TCP-NODELAY-VAL.
       05 FILLER PIC 9(6) COMP.
       05 TCP-NODELAY-BITSTREAM PIC 9(8) COMP.

    01 OPTNAME PIC 9(8) COMP.
        88  SO-REUSEADDR  VALUE  4.
        88  SO-KEEPALIVE  VALUE  8.
        88  SO-BROADCAST  VALUE  32.
        88  SO-LINGER     VALUE  128.
        88  SO-OOBINLINE  VALUE  256.
        88  SO-SNDBUF     VALUE  4097.
        88  SO-RCVBUF     VALUE  4098.
        88  SO-ERROR      VALUE  4103.
        88  SO-TYPE       VALUE  4104.
    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 31. GETSOCKOPT Call Instruction Example

For equivalent PL/I and assembler language declarations, see "Converting Parameter Descriptions" on page 63.

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 options before you issue GETSOCKOPT. The options are as follows:

- The following may be specified for TCP level options.

  Note: If not using the literal when specifing a TCP level option, turn on the high order bit in the option value.

  TCP_NODELAY

  Returns the status of Nagle algorithm (RFC 896).

  When *optval* is 0, Nagle algorithm is enabled and TCP will wait to send small packets of data until the acknowledgment for the previous data is received.

  When *optval* is nonzero, Nagle algorithm is disabled and TCP will send small packets of data even before the acknowledgment for previous data sent is received.

- The following may be specified for socket level options.

  SO-REUSEADDR

  Returns the status of local address reuse. When enabled, this option allows local addresses that are already in use to be bound. Instead of checking at BIND time (the normal algorithm) the system checks at CONNECT time to ensure that the local address and port do not have the same remote address and port. If the association already exists, Error 48 (EADDRINUSE) is returned when the CONNECT is issued.

  SO-KEEPALIVE

  Requests the status of the TCP keepalive mechanism for a stream socket. When activated, the keepalive mechanism periodically sends a packet on an otherwise idle connection. If the remote TCP does not respond to the packet or to retransmissions of the packet, the connection is terminated with the error ETIMEDOUT.

  SO-BROADCAST

  Requests the status of the broadcast option, which is the ability to send broadcast messages. This option has no meaning for stream sockets.

  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.

SO-OOBINLINE
Requests the status of how out-of-band data is to be received. This option has meaning only for stream sockets.

- When this option is enabled, out-of-band data is placed in the normal data input queue as it is received, making it available to RECV and RECVFROM without having to specify the MSG-OOB flag in those calls.
- When this option is disabled, out-of-band data is placed in the priority data input queue as it is received, making it available to RECV and RECVFROM only when the MSG-OOB flag is set.

SO_SNDBUF Returns the size of the data portion of the TCP/IP send buffer in OPTVAL. The size of the data portion of the send buffer is protocol-specific, based on the following value prior to any SETSOCKOPT call:

- The TCPSENDBufrsize keyword on the TCPCONFIG statement in the PROFILE.TCPIP data set for a TCP socket.
- The UDPSENDBufrsize keyword on the UDPCONFIG statement in the PROFILE.TCPIP data set for a UDP socket.
- The default of 65535 for a raw socket.

SO_RCVBUF Returns the size of the data portion of the TCP/IP receive buffer in OPTVAL. The size of the data portion of the receive buffer is protocol-specific, based on the following value prior to any SETSOCKOPT call:

- The TCPRCVBufrsize keyword on the TCPCONFIG statement in the PROFILE.TCPIP data set for a TCP socket.
- The UDPRCVBufrsize keyword on the UDPCONFIG statement in the PROFILE.TCPIP data set for a UDP socket.
- The default of 65535 for a raw socket.

SO-ERROR Requests any pending error on the socket and clears the error status. It can be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors (errors that are not returned explicitly by one of the socket calls).

SO-TYPE Returns socket type: stream, datagram, or raw.

Parameter Values Returned to the Application

OPTVAL

- For all values of OPTNAME other than SO-LINGER, OPTVAL is a 32-bit fullword, containing the status of the specified option.
  - If the requested option is enabled, the fullword contains a positive value; if the requested option is disabled, the fullword contains 0.

- If OPTNAME is set to SO-ERROR, OPTVAL contains the most recent ERRNO for the socket. This error variable is then cleared.
- If OPTNAME is set to SO-TYPE, OPTVAL returns X'1' for SOCK-STREAM, or X'2' for SOCK-DGRAM, or X'3' for SOCK-RAW.

• 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 0 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 125.

OPTLEN

A fullword binary field containing the length of the data returned in OPTVAL.

- For all values of OPTNAME except SO-LINGER, OPTLEN will be set to 4 (one fullword).
- 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 "Appendix A. Return Codes" on page 193 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.

UNIX-based platforms use a command called FORK to create a new child process that has the same descriptors as the parent process. You can use this new child process in the same way that you used the parent process.

TCP/IP normally uses GETCLIENTID, GIVESOCKET, and TAKESOCKET calls in the following sequence:

1. A process issues a GETCLIENTID call to get the job name of its region and its MVS subtask identifier. This information is used in a GIVESOCKET call.
2. The process issues a GIVESOCKET call to prepare a socket for use by a child process.
3. The child process issues a TAKESOCKET call to get the socket. The socket now belongs to the child process, and can be used by TCP/IP to communicate with another process.

   Note: The TAKESOCKET call returns a new socket descriptor in RETCODE. The child process must use this new socket descriptor for all calls that use this socket. The socket descriptor that was passed to the TAKESOCKET call must not be used.

4. After issuing the GIVESOCKET command, the parent process issues a SELECT command that waits for the child to get the socket.
5. When the child gets the socket, the parent receives an exception condition that releases the SELECT command.
6. The parent process closes the socket.

The original socket descriptor can now be reused by the parent.

Sockets that have been given, but not taken for a period of four days, will be closed and will no longer be available for taking. If a select for the socket is outstanding, it will be posted.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 32 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 32. GIVESOCKET Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

Parameter Values Set by the Application

SOC-FUNCTION

A 16-byte character field containing 'GIVESOCKET'. The field is left-justified and padded on the right with blanks.

S A halfword binary number set to the socket descriptor of the socket to be given.

CLIENT
A structure containing the identifier of the application to which the socket should be given.

DOMAIN
A fullword binary number that must be set to 2, indicating AF_INET.

NAME Specifies an eight-character field, left-justified, padded to the right with blanks, that can be set to the name of the MVS address space that will contain the application that is going to take the socket.

- If the socket-taking application is in the *same* address space as the socket-giving application (as in CICS), NAME can be specified. The socket-giving application can determine its own address space name by issuing the GETCLIENTID call.
- If the socket-taking application is in a *different* MVS address space (as in IMS), this field should be set to blanks. When this is done, any MVS address space that requests the socket can have it.

TASK Specifies an eight-character field that can be set to blanks, or to the identifier of the socket-taking MVS subtask. If this field is set to blanks, any subtask in the address space specified in the NAME field can take the socket.

- As used by IMS and CICS, the field should be set to blanks.
- If TASK identifier is non-blank, the socket-receiving task should already be in execution when the GIVESOCKET is issued.

RESERVED
A 20-byte reserved field. This field is required, but not used.

Parameter Values Returned to the Application

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix A. Return Codes” on page 193 for information about ERRNO return codes.

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

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

INITAPI

The INITAPI call connects an application to the TCP/IP interface. Almost all sockets programs that are written in COBOL, PL/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
- GETIBMOPT
- SELECT

- SELECTEX
- SOCKET
- TAKESOCKET

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 33 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 33. INITAPI Call Instruction Example

For equivalent PL/I and assembler language declarations, see "Converting Parameter Descriptions" on page 63.

Parameter Values Set by the Application

SOC-FUNCTION

A 16-byte character field containing INITAPI. The field is left-justified and padded on the right with blanks.

MAXSOC

A halfword binary field set to the maximum number of sockets this application will ever have open at one time. The maximum number is 2000 and the minimum number is 50. This value is used to determine the amount of memory that will be allocated for socket control blocks and buffers. If less than 50 are requested, MAXSOC defaults to 50.

IDENT A structure containing the identities of the TCP/IP address space and the calling program's address space. Specify IDENT on the INITAPI call from an address space.

TCPNAME
An 8-byte character field that should be set to the MVS job name of the TCP/IP address space with which you are connecting.

ADSNAME
An 8-byte character field set to the identity of the calling program's address space. For explicit-mode IMS server programs, use the TIMSrvAddrSpc field passed in the TIM. If ADSNAME is not specified, the system derives a value from the MVS control block structure.

SUBTASK
Indicates an 8-byte field, containing a unique subtask identifier which is used to distinguish between multiple subtasks within a single address space. Use your own job name as part of your subtask name. This will ensure that, if you issue more than one INITAPI command from the same address space, each SUBTASK parameter will be unique.

Parameter Values Returned to the Application

MAXSNO
A fullword binary field that contains the highest socket number assigned to this application. The lowest socket number is 0. If you have 50 sockets, they are numbered from 0 to 49. If MAXSNO is not specified, the value for MAXSNO is 49.

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See "Appendix A. Return Codes" on page 193 for information about ERRNO return codes.

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

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

IOCTL

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

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

The variable length parameters REQARG and RETARG are arguments that are passed to and returned from IOCTL. The length of REQARG and RETARG is determined by the value that you specify in COMMAND. See Table 3 on page 98 for information about REQARG and RETARG.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.

Amode:	31-bit or 24-bit. **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 34 shows an example of IOCTL call instructions.

```
       WORKING-STORAGE SECTION.
       01  SOC-FUNCTION           PIC X(16) VALUE 'IOCTL           '.
       01  S                      PIC 9(4)  BINARY.
       01  COMMAND                PIC 9(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  REQARG                 POINTER ;
       01  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 34. IOCTL Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

Parameter Values Set by the Application

SOC-FUNCTION

A 16-byte character field containing IOCTL. The field is left-justified and padded to the right with blanks.

S A halfword binary number set to the descriptor of the socket to be controlled.

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

FIONBIO
Sets or clears blocking status.

FIONREAD
Returns the number of immediately readable bytes for the socket.

SIOCADDRT
Adds a specified routing table entry.

SIOCATMARK
Determines whether the current location in the data input is pointing to out-of-band data.

SIOCDELRT
Deletes a specified routing table entry.

SIOCGIFADDR
Requests the network interface address for a given interface name. See the NAME field in Figure 35 for the address format.

SIOCGIFBRDADDR
Requests the network interface broadcast address for a given interface name. See the NAME field in Figure 35 for the address format.

SIOCGIFCONF
Requests the network interface configuration. The configuration is a variable number of 32-byte structures formatted as shown in Figure 35.

- When IOCTL is issued, REQARG must contain the length of the array to be returned. To determine the length of REQARG, multiply the structure length (array element) by the number of interfaces requested. The maximum number of array elements that TCP/IP can return is 100.
- When IOCTL is issued, RETARG must be set to the beginning of the storage area that you have defined in your program for the array to be returned.

```
03  NAME         PIC X(16).
03  FAMILY       PIC 9(4) BINARY.
03  PORT         PIC 9(4) BINARY.
03  ADDRESS      PIC 9(8) BINARY.
03  RESERVED     PIC X(8).
```

Figure 35. Interface Request Structure (IFREQ) for the IOCTL Call

SIOCGIFDSTADDR
Requests the network interface destination address for a given interface name. (See IFREQ NAME field, Figure 35 for format.)

SIOCGIFFLAGS
Requests the network interface flags.

SIOCGIFMETRIC
Requests the network interface routing metric.

SIOCGIFNETMASK
Requests the network interface network mask.

SIOCSIFMETRIC
Sets the network interface routing metric.

SIOCSIFDSTADDR
Sets the network interface destination address.

SIOCSIFFLAGS
Sets the network interface flags.

REQARG and RETARG
REQARG is used to pass arguments to IOCTL, and RETARG receives arguments from IOCTL. The REQARG and RETARG parameters are described in Table 3.

Table 3. IOCTL Call Arguments

COMMAND/CODE	SIZE	REQARG	SIZE	RETARG
FIONBIO X'8004A77E'	4	Set socket mode to: X'00'=blocking, X'01'=nonblocking.	0	Not used
FIONREAD X'4004A77F'	0	Not used.	4	Number of characters available for read.
SIOCADDRT X'8030A70A'	48	For IBM use only.	0	For IBM use only.
SIOCATMARK X'4004A707'	0	Not used.	4	X'00'= at OOB data X'01'= not at OOB data.
SIOCDELRT X'8030A70B'	48	For IBM use only.	0	For IBM use only.
SIOCGIFADDR X'C020A70D'	32	First 16 bytes - interface name. Last 16 bytes - not used.	32	Network interface address, see Figure 35 on page 97 for format.
SIOCGIFBRDADDR X'C020A712'	32	First 16 bytes - interface name. Last 16 bytes - not used.	32	Network interface address, see Figure 35 on page 97 for format.
SIOCGIFCONF X'C008A714'	8	Size of RETARG.	See note.	
Note: When you call IOCTL with the SIOCGIFCONF command set, REQARG should contain the length in bytes of RETARG. Each interface is assigned a 32-byte array element and REQARG should be set to the number of interfaces times 32. TCP/IP Services can return up to 100 array elements.				
SIOCGIFDSTADDR X'C020A70F'	32	First 16 bytes - interface name. Last 16 bytes - not used.	32	Destination interface address, see Figure 35 on page 97 for format.
SIOCGIFFLAGS X'C020A711'	32	For IBM use only.	32	For IBM use only.
SIOCGIFMETRIC X'C020A717'	32	For IBM use only.	32	For IBM use only.

Table 3. IOCTL Call Arguments (continued)

COMMAND/CODE	SIZE	REQARG	SIZE	RETARG
SIOCGIFNETMASK X'C020A715'	32	For IBM use only.	32	For IBM use only.
SIOCSIFMETRIC X'8020A718'	32	For IBM use only.	0	For IBM use only.
SIOCSIFDSTADDR X'8020A70E'	32	For IBM use only.	0	For IBM use only.
SIOCSIFFLAGS X'8020A710'	32	For IBM use only.	0	For IBM use only.

Parameter Values Returned to the Application

RETARG

Returns an array whose size is based on the value in COMMAND. See Table 3 for information about REQARG and RETARG.

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix A. Return Codes” on page 193 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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

The COMMAND SIOGIFCONF returns a variable number of network interface configurations. Figure 36 contains an example of a COBOL II routine that can be used to work with such a structure.

Note: This call can only be programmed in languages that support address pointers. Figure 36 shows a COBOL II example for SIOCGIFCONF.

```
WORKING STORAGE SECTION.
  77   REQARG        PIC 9(8) COMP.
  77   COUNT         PIC 9(8) COMP VALUE max number of interfaces.
LINKAGE SECTION.
  01   RETARG.
       05   IOCTL-TABLE OCCURS 1 TO max TIMES DEPENDING ON COUNT.
            10    NAME     PIC X(16).
            10    FAMILY   PIC 9(4) BINARY.
            10    PORT     PIC 9(4) BINARY.
            10    ADDR     PIC 9(8) BINARY.
            10    NULLS    PIC X(8).
PROCEDURE DIVISION.
  MULTIPLY COUNT BY 32 GIVING REQARQ.
  CALL 'EZASOKET' USING SOC-FUNCTION S COMMAND
      REQARG RETARG ERRNO RETCODE.
```

Figure 36. COBOL II Example for SIOCGIFCONF

LISTEN

The LISTEN call:

- Completes the bind, if BIND has not already been called for the socket.

- Creates a connection-request queue of a specified length for incoming connection requests.

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

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 37 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 37. LISTEN Call Instruction Example

For equivalent PL/I and assembler language declarations, see "Converting Parameter Descriptions" on page 63.

Parameter Values Set by the Application

SOC-FUNCTION
A 16-byte character field containing LISTEN. The field is left-justified and padded to the right with blanks.

S A halfword binary number set to the socket descriptor.

BACKLOG
A fullword binary number set to the number of communication requests to be queued.

Parameter Values Returned to the Application

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix A. Return Codes” on page 193 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 socket s. This is the conventional TCP/IP read data operation. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard extra bytes.

For stream sockets, data is processed as streams of information with no boundaries separating the data. For example, if programs A and B are connected with a stream socket and program A sends 1000 bytes, each call to this function can return any number of bytes, up to the entire 1000 bytes. The number of bytes returned will be contained in RETCODE. Therefore, programs using stream sockets should place this call in a loop that repeats until all data has been received.

Note: See “EZACIC05” on page 139 for a subroutine that will translate ASCII input data to EBCDIC.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 38 on page 102 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 38. READ Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

Parameter Values Set by the Application

SOC-FUNCTION
A 16-byte character field containing READ. The field is left-justified and padded to the right with blanks.

S A halfword binary number set to the socket descriptor of the socket that is going to read the data.

NBYTE
A fullword binary number set to the size of BUF. READ does not return more than the number of bytes of data in NBYTE even if more data is available.

Parameter Values Returned to the Application

BUF On input, a buffer to be filled by completion of the call. The length of BUF must be at least as long as the value of NBYTE.

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix A. Return Codes” on page 193 for information about ERRNO return codes.

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

Value	Description
0	A 0 return code indicates that the connection is closed and no data is available.
>0	A positive value indicates the number of bytes copied into the buffer.
–1	Check **ERRNO** for an error code.

READV

The READV function reads data on a socket and stores it in a set of buffers. If a datagram packet is too long to fit in the supplied buffers, datagram sockets discard extra bytes.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.

Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 39 shows an example of READV call instructions.

```
WORKING-STORAGE SECTION.
01  SOKET-FUNCTION         PIC X(16) VALUE 'READV           '.
01  S                       PIC 9(4)  BINARY.
01  IOVCNT                 PIC 9(4)  BINARY.

01  IOV.
    03 BUFFER-ENTRY OCCURS N TIMES.
      05 BUFFER_ADDR       POINTER.
      05 RESERVED          PIC X(4).
      05 BUFFER_LENGTH     PIC 9(4).

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

Figure 39. READV Call Instruction Example

Parameter Values Set by the Application

SOC-FUNCTION
A 16-byte character field containing READV. The field is left-justified and padded to the right with blanks.

S A value or the address of a halfword binary number specifying the descriptor of the socket into which the data is to be read.

IOV An array of tripleword structures with the number of structures equal to the value in IOVCNT and the format of the structures as follows:

Fullword 1
Pointer to the address of a data buffer, which is filled in on completion of the call

Fullword 2
Reserved

Fullword 3
The length of the data buffer referenced in fullword one

IOVCNT
A fullword binary field specifying the number of data buffers provided for this call.

Parameter Values Returned to the Application

ERRNO

A fullword binary field. If RETCODE is negative, this contains an error number. See “Appendix A. Return Codes” on page 193 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

Value	Description
0	A 0 return code indicates that the connection is closed and no data is available.
>0	A positive value indicates the number of bytes copied into the buffer.
–1	Check **ERRNO** for an error code.

RECV

The RECV call, like READ, receives data on a socket with descriptor S. RECV applies only to connected sockets. If a datagram packet is too long to fit in the supplied buffers, datagram sockets discard extra bytes.

For additional control of the incoming data, RECV can:
- Peek at the incoming message without having it removed from the buffer
- Read out-of-band data

For stream sockets, data is processed as streams of information with no boundaries separating the data. For example, if programs A and B are connected with a stream socket and program A sends 1000 bytes, each call to this function can return any number of bytes, up to the entire 1000 bytes. The number of bytes returned will be contained in RETCODE. Therefore, programs using stream sockets should place RECV in a loop that repeats until all data has been received.

If data is not available for the socket, and the socket is in blocking mode, RECV blocks the caller until data arrives. If data is not available and the socket is in nonblocking mode, RECV returns a –1 and sets ERRNO to 35 (EWOULDBLOCK). See “FCNTL” on page 72 or “IOCTL” on page 95 for a description of how to set nonblocking mode.

For raw sockets, RECV adds a 20-byte header.

Note: See “EZACIC05” on page 139 for a subroutine that will translate ASCII input data to EBCDIC.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 61.

ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 40 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.
        88  OOB                    VALUE IS 1.
        88  PEEK                   VALUE IS 2.
    01  NBYTE           PIC 9(8) BINARY.
    01  BUF             PIC X(length of buffer).
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

PROCEDURE
     CALL 'EZASOKET' USING SOC-FUNCTION S FLAGS NBYTE BUF
                     ERRNO RETCODE.
```

Figure 40. RECV Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

Parameter Values Set by the Application

SOC-FUNCTION

A 16-byte character field containing RECV. The field is left-justified and padded to the right with blanks.

S A halfword binary number set to the socket descriptor of the socket to receive the data.

FLAGS

A fullword binary field with values as follows:

Literal Value	Binary Value	Description
NO-FLAG	0	Read data.
OOB	1	Receive out-of-band data (stream sockets only). Even if the OOB flag is not set, out-of-band data can be read if the SO-OOBINLINE option is set for the socket.
PEEK	2	Peek at the data, but do not destroy data. If the peek flag is set, the next RECV call will read the same data.

NBYTE

A value or the address of a fullword binary number set to the size of BUF. RECV does not receive more than the number of bytes of data in NBYTE even if more data is available.

Parameter Values Returned to the Application

BUF The input buffer to receive the data.

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix A. Return Codes” on page 193 for information about ERRNO return codes.

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

Value	Description
0	The socket is closed.
>0	A positive return code indicates the number of bytes copied into the buffer.
–1	Check **ERRNO** for an error code.

RECVFROM

The RECVFROM call receives data on a socket with descriptor S and stores it in a buffer. The RECVFROM call applies to both connected and unconnected sockets. The socket address is returned in the NAME structure. If a datagram packet is too long to fit in the supplied buffers, datagram sockets discard extra bytes.

For datagram protocols, recvfrom() returns the source address associated with each incoming datagram. For connection-oriented protocols like TCP, getpeername() returns the address associated with the other end of the connection.

If NAME is nonzero, the call returns the address of the sender. The NBYTE parameter should be set to the size of the buffer.

On return, NBYTE contains the number of data bytes received.

For stream sockets, data is processed as streams of information with no boundaries separating the data. For example, if programs A and B are connected with a stream socket and program A sends 1000 bytes, each call to this function can return any number of bytes, up to the entire 1000 bytes. The number of bytes returned will be contained in RETCODE. Therefore, programs using stream sockets should place RECVFROM in a loop that repeats until all data has been received.

For raw sockets, RECVFROM adds a 20-byte header.

If data is not available for the socket, and the socket is in blocking mode, RECVFROM blocks the caller until data arrives. If data is not available and the socket is in nonblocking mode, RECVFROM returns a –1 and sets ERRNO to 35 (EWOULDBLOCK). See “FCNTL” on page 72 or “IOCTL” on page 95 for a description of how to set nonblocking mode.

Note: See “EZACIC05” on page 139 for a subroutine that will translate ASCII input data to EBCDIC.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.

Amode:	31-bit or 24-bit. **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 41 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.
        88  OOB                    VALUE IS 1.
        88  PEEK                   VALUE IS 2.
    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 41. RECVFROM Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

Parameter Values Set by the Application

SOC-FUNCTION

A 16-byte character field containing RECVFROM. The field is left-justified and padded to the right with blanks.

S A halfword binary number set to the socket descriptor of the socket to receive the data.

FLAGS

A fullword binary field containing flag values as follows:

Literal Value	Binary Value	Description
NO-FLAG	0	Read data.
OOB	1	Receive out-of-band data (stream sockets only). Even if the OOB flag is not set, out-of-band data can be read if the SO-OOBINLINE option is set for the socket.
PEEK	2	Peek at the data, but do not destroy data. If the peek flag is set, the next RECVFROM call will read the same data.

NBYTE
A fullword binary number specifying the length of the input buffer.

Parameter Values Returned to the Application

BUF Defines an input buffer to receive the input data.

NAME A structure containing the address of the socket that sent the data. The structure is:

FAMILY
A halfword binary number specifying the addressing family. The value is always 2, indicating AF_INET.

PORT A halfword binary number specifying the port number of the sending socket.

IP-ADDRESS
A fullword binary number specifying the 32-bit internet address of the sending socket.

RESERVED
An 8-byte reserved field. This field is required, but is not used.

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix A. Return Codes” on page 193 for information about ERRNO return codes.

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

Value	Description
0	The socket is closed.
>0	A positive return code indicates the number of bytes of data transferred by the read call.
–1	Check **ERRNO** for an error code.

RECVMSG

The RECVMSG call receives messages on a socket with descriptor S and stores them in an array of message headers. If a datagram packet is too long to fit in the supplied buffers, datagram sockets discard extra bytes.

For datagram protocols, recvmsg() returns the source address associated with each incoming datagram. For connection-oriented protocols like TCP, getpeername() returns the address associated with the other end of the connection.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 42 on page 110 shows an example of RECVMSG call instructions.

```
 WORKING STORAGE
     01  SOC-FUNCTION    PIC X(16)  VALUE IS 'RECVMSG'.
     01  S               PIC 9(4)   BINARY.
     01  MSG-HDR.
         03  MSG-NAME        USAGE IS POINTER.
         03  MSG-NAME-LEN    USAGE IS POINTER.
         03  IOV             USAGE IS POINTER.
         03  IOVCNT          USAGE IS POINTER.
         03  MSG-ACCRIGHTS   USAGE IS POINTER.
         03  MSG-ACCRIGHTS-LEN USAGE IS POINTER.

     01  FLAGS           PIC 9(8)   BINARY.
         88  NO-FLAG                  VALUE IS 0.
         88  OOB                      VALUE IS 1.
         88  PEEK                     VALUE IS 2.
     01  ERRNO           PIC 9(8)   BINARY.
     01  RETCODE         PIC S9(8)  BINARY.

LINKAGE SECTION.

     01 RECVMSG-IOVECTOR.
        03 IOV1A               USAGE IS POINTER.
           05 IOV1AL              PIC 9(8) COMP.
           05 IOV1L               PIC 9(8) COMP.
        03 IOV2A               USAGE IS POINTER.
           05 IOV2AL              PIC 9(8) COMP.
           05 IOV2L               PIC 9(8) COMP.
        03 IOV3A               USAGE IS POINTER.
           05 IOV3AL              PIC 9(8) COMP.
           05 IOV3L               PIC 9(8) COMP.

     01 RECVMSG-BUFFER1     PIC X(16).
     01 RECVMSG-BUFFER2     PIC X(16).
     01 RECVMSG-BUFFER3     PIC X(16).
     01 RECVMSG-BUFNO       PIC 9(8) COMP.

PROCEDURE

           SET MSG-NAME TO NULLS.
           SET MSG-NAME-LEN TO NULLS.
           SET IOV TO ADDRESS OF RECVMSG-IOVECTOR.
           MOVE 3 TO RECVMSG-BUFNO.
           SET MSG-IOVCNT TO ADDRESS OF RECVMSG-BUFNO.
           SET IOV1A TO ADDRESS OF RECVMSG-BUFFER1.
           MOVE 0 TO MSG-IOV1AL.
           MOVE LENGTH OF RECVMSG-BUFFER1 TO MSG-IOV1L.
           SET IOV2A TO ADDRESS OF RECVMSG-BUFFER2.
           MOVE 0 TO IOV2AL.
           MOVE LENGTH OF RECVMSG-BUFFER2 TO IOV2L.
           SET IOV3A TO ADDRESS OF RECVMSG-BUFFER3.
           MOVE 0 TO IOV3AL.
           MOVE LENGTH OF RECVMSG-BUFFER3 TO IOV3L.
           SET MSG-ACCRIGHTS TO NULLS.
           SET MSG-ACCRIGHTS-LEN TO NULLS.
           MOVE X'00000000' TO FLAGS.
           MOVE SPACES TO RECVMSG-BUFFER1.
           MOVE SPACES TO RECVMSG-BUFFER2.
           MOVE SPACES TO RECVMSG-BUFFER3.

       CALL 'EZASOKET' USING SOC-FUNCTION S MSGHDR FLAGS ERRNO RETCODE.
```

Figure 42. RECVMSG Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

Parameter Values Set by the Application

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

MSG On input, a pointer to a message header into which the message is received upon completion of the call.

Field Description

NAME On input, a pointer to a buffer where the sender address is stored upon completion of the call.

NAME-LEN
On input, a pointer to the size of the address buffer that is filled in on completion of the call.

IOV On input, a pointer to an array of tripleword structures with the number of structures equal to the value in IOVCNT and the format of the structures as follows:

Fullword 1
A pointer to the address of a data buffer.

Fullword 2
Reserved.

Fullword 3
A pointer to the length of the data buffer referenced in fullword 1.

In COBOL, the IOV structure must be defined separately in the Linkage section, as shown in the example.

IOVCNT
On input, a pointer to a fullword binary field specifying the number of data buffers provided for this call.

ACCRIGHTS
On input, a pointer to the access rights received. This field is ignored.

ACCRLEN
On input, a pointer to the length of the access rights received. This field is ignored.

FLAGS
A fullword binary field with values as follows:

Literal Value	Binary Value	Description
NO-FLAG	0	Read data.
OOB	1	Receive out-of-band data (stream sockets only). Even if the OOB flag is not set, out-of-band data can be read if the SO-OOBINLINE option is set for the socket.
PEEK	2	Peek at the data, but do not destroy data. If the peek flag is set, the next RECVMSG call will read the same data.

Parameter Values Returned to the Application

ERRNO

A fullword binary field. If RETCODE is negative, this contains an error number. See “Appendix A. Return Codes” on page 193 for information about ERRNO return codes.

RETCODE

A fullword binary field with the following values:

Value	Description
<0	Call returned error. See ERRNO field.
0	Connection partner has closed connection.
>0	Number of bytes read.

SELECT

In a process where multiple I/O operations can occur it is necessary for the program to be able to wait on one or several of the operations to complete.

For example, consider a program that issues a READ to multiple sockets whose blocking mode is set. Because the socket would block on a READ call, only one socket could be read at a time. Setting the sockets nonblocking would solve this problem, but would require polling each socket repeatedly until data became available. The SELECT call allows you to test several sockets and to execute a subsequent I/O call only when one of the tested sockets is ready, thereby ensuring that the I/O call will not block.

To use the SELECT call as a timer in your program, do one of the following:
- Set the read, write, and except arrays to zeros.
- Specify MAXSOC <= 0.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Defining Which Sockets to Test

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

- When a socket is ready to read, one of the following has occurred:
  - A buffer for the specified sockets contains input data. If input data is available for a given socket, a read operation on that socket will not block.

  - A connection has been requested on that socket.
- When a socket is ready to write, TCP/IP can accommodate additional output data. If TCP/IP can accept additional output for a given socket, a write operation on that socket will not block.
- When an exception condition has occurred on a specified socket it is an indication that a TAKESOCKET has occurred for that socket.

Each socket descriptor is represented by a bit in a bit string. The bit strings are contained in 32-bit fullwords, numbered from right to left. The rightmost bit represents socket descriptor 0, the leftmost bit represents socket descriptor 31, and so on. If your process uses 32 or fewer sockets, the bit string is 1 fullword. If your process uses 33 sockets, the bit string is 2 fullwords. You define the sockets that you want to test by turning on bits in the string.

Note: To simplify string processing in COBOL, you can use the program EZACIC06 to convert each bit in the string to a character. For more information, see "EZACIC06" on page 140.

Read Operations

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

To test whether any of several sockets is ready for reading, set the appropriate bits in RSNDMSK to one before issuing the SELECT call. When the SELECT call returns, the corresponding bits in the RRETMSK indicate sockets are ready for reading.

Write Operations

A socket is selected for writing (ready to be written) when:

- TCP/IP can accept additional outgoing data.
- The socket is marked nonblocking and a previous CONNECT did not complete immediately. In this case, CONNECT returned an ERRNO with a value of 36 (EINPROGRESS). This socket will be selected for write when the CONNECT completes.

A call to WRITE, SEND, or SENDTO blocks when the amount of data to be sent exceeds the amount of data TCP/IP can accept. To avoid this, you can precede the write operation with a SELECT call to ensure that the socket is ready for writing. Once a socket is selected for WRITE, the program can determine the amount of TCP/IP buffer space available by issuing the GETSOCKOPT call with the SO-SNDBUF option.

To test whether any of several sockets is ready for writing, set the WSNDMSK bits representing those sockets to 1 before issuing the SELECT call. When the SELECT call returns, the corresponding bits in the WRETMSK indicate sockets are ready for writing.

Exception Operations

For each socket to be tested, the SELECT call can check for an existing exception condition. Two exception conditions are supported:

- The calling program (concurrent server) has issued a GIVESOCKET command and the target child server has successfully issued the TAKESOCKET call. When this condition is selected, the calling program (concurrent server) should issue CLOSE to dissociate itself from the socket.

- A socket has received out-of-band data. On this condition, a READ will return the out-of-band data ahead of program data.

To test whether any of several sockets have an exception condition, set the ESNDMSK bits representing those sockets to 1. When the SELECT call returns, the corresponding bits in the ERETMSK indicate sockets with exception conditions.

MAXSOC Parameter

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

TIMEOUT Parameter

If the time specified in the TIMEOUT parameter elapses before any event is detected, the SELECT call returns, and the RETCODE is set to 0.

Figure 43 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 43. 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 63.

Parameter Values Set by the Application

SOC-FUNCTION

A 16-byte character field containing SELECT. The field is left-justified and padded on the right with blanks.

MAXSOC

A fullword binary field set to the largest socket descriptor number that is to be checked plus 1. (Remember to start counting at 0).

TIMEOUT

If TIMEOUT is a positive value, it specifies the maximum interval to wait for the selection to complete. If TIMEOUT-SECONDS is a negative value, the SELECT call blocks until a socket becomes ready. To poll the sockets and return immediately, specify the TIMEOUT value to be 0.

TIMEOUT is specified in the two-word TIMEOUT as follows:

- TIMEOUT-SECONDS, word one of the TIMEOUT field, is the seconds component of the timeout value.
- TIMEOUT-MICROSEC, word two of the TIMEOUT field, is the microseconds component of the timeout value (0—999999).

For example, if you want SELECT to time out after 3.5 seconds, set TIMEOUT-SECONDS to 3 and TIMEOUT-MICROSEC to 500000.

RSNDMSK

A bit string sent to request read event status.

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

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

WSNDMSK

A bit string sent to request write event status.

- For each socket to be checked for pending write events, the corresponding bit in the string should be set to *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 "Appendix A. Return Codes" on page 193 for information about ERRNO return codes.

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

Value	Description
>0	Indicates the sum of all ready sockets in the three masks.
0	Indicates that the SELECT time limit has expired.
–1	Check **ERRNO** for an error code.

SELECTEX

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

To use the SELECTEX call as a timer in your program, do either of the following:
- Set the read, write, and except arrays to zeros.
- Specify MAXSOC <= 0.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 44 on page 117 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 44. SELECTEX Call Instruction Example

Parameter Values Set by the Application

SOC-FUNCTION

A 16-byte character field containing SELECT. The field is left-justified and padded on the right with blanks.

MAXSOC

A fullword binary field specifying the largest socket descriptor number being checked.

TIMEOUT

If TIMEOUT is a positive value, it specifies a maximum interval to wait for the selection to complete. If TIMEOUT-SECONDS is a negative value, the SELECT call blocks until a socket becomes ready. To poll the sockets and return immediately, set TIMEOUT to be zeros.

TIMEOUT is specified in the two-word TIMEOUT as follows:

- TIMEOUT-SECONDS, word one of the TIMEOUT field, is the seconds component of the timeout value.
- TIMEOUT-MICROSEC, word two of the TIMEOUT field, is the microseconds component of the timeout value (0—999999).

For example, if you want SELECTEX to time out after 3.5 seconds, set TIMEOUT-SECONDS to 3 and TIMEOUT-MICROSEC to 500000.

RSNDMSK

The bit-mask array to control checking for read interrupts. If this parameter is not specified or the specified bit-mask is zeros, the SELECT will not check for read interrupts. The length of this bit-mask array is dependent on the value in MAXSOC.

WSNDMSK
The bit-mask array to control checking for write interrupts. If this parameter is not specified or the specified bit-mask is zeros, the SELECT will not check for write interrupts. The length of this bit-mask array is dependent on the value in MAXSOC.

ESNDMSK
The bit-mask array to control checking for exception interrupts. If this parameter is not specified or the specified bit-mask is zeros, the SELECT will not check for exception interrupts. The length of this bit-mask array is dependent on the value in MAXSOC.

SELECB
An ECB which, if posted, causes completion of the SELECTEX.

Parameter Values Returned to the Application

ERRNO
A fullword binary field; if RETCODE is negative, this contains an error number. See “Appendix A. Return Codes” on page 193 for information about ERRNO return codes.

RETCODE
A fullword binary field

Value	Meaning
>0	The number of ready sockets.
0	Either the SELECTEX time limit has expired (ECB value will be 0) or one of the caller's ECBs has been posted (ECB value will be nonzero and the caller's descriptor sets will be set to 0). The caller must initialize the ECB values to 0 before issuing the SELECTEX macro.
-1	Check **ERRNO** for an error code.

RRETMSK
The bit-mask array returned by the SELECT if RSNDMSK is specified. The length of this bit-mask array is dependent on the value in MAXSOC.

WRETMSK
The bit-mask array returned by the SELECT if WSNDMSK is specified. The length of this bit-mask array is dependent on the value in MAXSOC.

ERETMSK
The bit-mask array returned by the SELECT if ESNDMSK is specified. The length of this bit-mask array is dependent on the value in MAXSOC.

SEND

The SEND call sends data on a specified connected socket.

The FLAGS field allows you to:

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

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

For stream sockets, data is processed as streams of information with no boundaries separating the data. For example, if a program is required to send 1000 bytes, each call to this function can send any number of bytes, up to the entire 1000 bytes, with the number of bytes sent returned in RETCODE. Therefore, programs using stream sockets should place this call in a loop, reissuing the call until all data has been sent.

Note: See "EZACIC04" on page 138 for a subroutine that will translate EBCDIC input data to ASCII.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 45 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 45. SEND Call Instruction Example

For equivalent PL/I and assembler language declarations, see "Converting Parameter Descriptions" on page 63.

Parameter Values Set by the Application

SOC-FUNCTION
A 16-byte character field containing SEND. The field is left-justified and padded on the right with blanks.

S A halfword binary number specifying the socket descriptor of the socket that is sending data.

FLAGS
A fullword binary field with values as follows:

Literal Value	Binary Value	Description
NO-FLAG	0	No flag is set. The command behaves like a WRITE call.
OOB	1	Send out-of-band data. (Stream sockets only.) Even if the OOB flag is not set, out-of-band data can be read if the SO-OOBINLINE option is set for the socket.
DONT-ROUTE	4	Do not route. Routing is provided by the calling program.

NBYTE
A fullword binary number set to the number of bytes of data to be transferred.

BUF The buffer containing the data to be transmitted. BUF should be the size specified in NBYTE.

Parameter Values Returned to the Application

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix A. Return Codes” on page 193 for information about ERRNO return codes.

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

Value	Description
≥0	A successful call. The value is set to the number of bytes transmitted.
–1	Check **ERRNO** for an error code.

SENDMSG

The SENDMSG call sends messages on a socket with descriptor S passed in an array of messages.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 46 shows an example of SENDMSG call instructions.

```
 WORKING STORAGE
     01  SOC-FUNCTION    PIC X(16)  VALUE IS 'SENDMSG'.
     01  S               PIC 9(4)   BINARY.
     01  MSGHDR.
         03  MSG-NAME        USAGE IS POINTER.
         03  MSG-NAME-LEN    USAGE IS POINTER.
         03  IOV             USAGE IS POINTER.
         03  IOVCNT          USAGE IS POINTER.
         03  MSG-ACCRIGHTS   USAGE IS POINTER.
         03  MSG-ACCRIGHTS-LEN USAGE IS POINTER.

     01  FLAGS           PIC 9(8)   BINARY.
         88  NO-FLAG                  VALUE IS 0.
         88  OOB                      VALUE IS 1.
         88  DONTROUTE                VALUE IS 4.
     01  ERRNO           PIC 9(8)   BINARY.
     01  RETCODE         PIC S9(8)  BINARY.

LINKAGE SECTION.

     01 SENDMSG-IOVECTOR.
        03 IOV1A               USAGE IS POINTER.
           05 IOV1AL              PIC 9(8) COMP.
           05 IOV1L               PIC 9(8) COMP.
        03 IOV2A               USAGE IS POINTER.
           05 IOV2AL              PIC 9(8) COMP.
           05 IOV2L               PIC 9(8) COMP.
        03 IOV3A               USAGE IS POINTER.
           05 IOV3AL              PIC 9(8) COMP.
           05 IOV3L               PIC 9(8) COMP.

     01 SENDMSG-BUFFER1     PIC X(16).
     01 SENDMSG-BUFFER2     PIC X(16).
     01 SENDMSG-BUFFER3     PIC X(16).
     01 SENDMSG-BUFNO       PIC 9(8) COMP.

PROCEDURE

           SET MSG-NAME TO NULLS.
           SET MSG-NAME-LEN TO NULLS.
           SET IOV TO ADDRESS OF SENDMSG-IOVECTOR.
           MOVE 3 TO SENDMSG-BUFNO.
           SET MSG-IOVCNT TO ADDRESS OF SENDMSG-BUFNO.
           SET IOV1A TO ADDRESS OF SENDMSG-BUFFER1.
           MOVE 0 TO MSG-IOV1AL.
           MOVE LENGTH OF SENDMSG-BUFFER1 TO MSG-IOV1L.
           SET IOV2A TO ADDRESS OF SENDMSG-BUFFER2.
           MOVE 0 TO IOV2AL.
           MOVE LENGTH OF SENDMSG-BUFFER2 TO IOV2L.
           SET IOV3A TO ADDRESS OF SENDMSG-BUFFER3.
           MOVE 0 TO IOV3AL.
           MOVE LENGTH OF SENDMSG-BUFFER3 TO IOV3L.
           SET MSG-ACCRIGHTS TO NULLS.
           SET MSG-ACCRIGHTS-LEN TO NULLS.
           MOVE X'00000000' TO FLAGS.
           MOVE SPACES TO SENDMSG-BUFFER1.
           MOVE SPACES TO SENDMSG-BUFFER2.
           MOVE SPACES TO SENDMSG-BUFFER3.

       CALL 'EZASOKET' USING SOC-FUNCTION S MSGHDR FLAGS ERRNO RETCODE.
```

Figure 46. SENDMSG Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

Parameter Values Set by the Application

SOC-FUNCTION
A 16-byte character field containing SENDMSG. The field is left-justified and padded on the right with blanks.

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

MSG A pointer to an array of message headers from which messages are sent.

Field Description

NAME On input, a pointer to a buffer where the sender's address is stored upon completion of the call.

NAME-LEN
On input, a pointer to the size of the address buffer that is filled in on completion of the call.

IOV On input, a pointer to an array of three fullword structures with the number of structures equal to the value in IOVCNT and the format of the structures as follows:

Fullword 1
A pointer to the address of a data buffer.

Fullword 2
Reserved.

Fullword 3
A pointer to the length of the data buffer referenced in Fullword 1.

In COBOL, the IOV structure must be defined separately in the Linkage section, as shown in the example.

IOVCNT
On input, a pointer to a fullword binary field specifying the number of data buffers provided for this call.

ACCRIGHTS
On input, a pointer to the access rights received. This field is ignored.

ACCRIGHTS-LEN
On input, a pointer to the length of the access rights received. This field is ignored.

FLAGS
A fullword field containing the following:

Literal Value	Binary Value	Description
NO-FLAG	0	No flag is set. The command behaves like a WRITE call.
OOB	1	Send out-of-band data. (Stream sockets only.) Even if the OOB flag is not set, out-of-band data can be read if the SO-OOBINLINE option is set for the socket.
DONTROUTE	4	Do not route. Routing is provided by the calling program.

Parameter Values Returned to the Application

ERRNO

A fullword binary field. If RETCODE is negative, this contains an error number. See “Appendix A. Return Codes” on page 193 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	An error occurred.

SENDTO

SENDTO is similar to SEND, except that it includes the destination address parameter. The destination address allows you to use the SENDTO call to send datagrams on a UDP socket, regardless of whether the socket is connected.

The FLAGS parameter allows you to:

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

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

For stream sockets, data is processed as streams of information with no boundaries separating the data. For example, if a program is required to send 1000 bytes, each call to this function can send any number of bytes, up to the entire 1000 bytes, with the number of bytes sent returned in RETCODE. Therefore, programs using stream sockets should place SENDTO in a loop that repeats the call until all data has been sent.

Note: See “EZACIC04” on page 138 for a subroutine that will translate EBCDIC input data to ASCII.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.

Amode:	31-bit or 24-bit. **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 47 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.
        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  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 47. SENDTO Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

Parameter Values Set by the Application

SOC-FUNCTION
A 16-byte character field containing SENDTO. The field is left-justified and padded on the right with blanks.

S A halfword binary number set to the socket descriptor of the socket sending the data.

FLAGS
A fullword field that returns one of the following:

Literal Value	Binary Value	Description
NO-FLAG	0	No flag is set. The command behaves like a WRITE call.
OOB	1	Send out-of-band data. (Stream sockets only.) Even if the OOB flag is not set, out-of-band data can be read if the SO-OOBINLINE option is set for the socket.
DONT-ROUTE	4	Do not route. Routing is provided by the calling program.

NBYTE
A fullword binary number set to the number of bytes to transmit.

BUF Specifies the buffer containing the data to be transmitted. BUF should be the size specified in NBYTE.

NAME Specifies the socket name structure as follows:

FAMILY
A halfword binary field containing the addressing family. For TCP/IP the value must be 2, indicating AF_INET.

PORT A halfword binary field containing the port number bound to the socket.

IP-ADDRESS
A fullword binary field containing the socket's 32-bit internet address.

RESERVED
Specifies 8-byte reserved field. This field is required, but not used.

Parameter Values Returned to the Application

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See "Appendix A. Return Codes" on page 193 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.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 48 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  TCP_NODELAY   VALUE  -2147483649 ('x80000001').
        88  SO-REUSEADDR  VALUE  4.
        88  SO-KEEPALIVE  VALUE  8.
        88  SO-BROADCAST  VALUE  32.
        88  SO-LINGER     VALUE  128.
        88  SO-OOBINLINE  VALUE  256.
        88  SO-SNDBUF     VALUE  4097.
        88  SO-RCVBUF     VALUE  4098.
    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 48. SETSOCKOPT Call Instruction Example

For equivalent PL/I and assembler language declarations, see "Converting Parameter Descriptions" on page 63.

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.

- The following may be specified for TCP level options.

  Note: If not using the literal when specifing a TCP level option, turn on the high order bit in the option value.

TCP_NODELAY

Toggles the use of Nagle algorithm (RFC 896) for all data sent over the socket. Under most circumstances, TCP sends data when it is presented.

When outstanding data has not yet been acknowledged, it gathers small amounts of output to be sent in a single packet once an acknowledgment is received. For interactive applications, such as ones that send a stream of mouse events that receive no replies, this gathering of output can cause significant delays. For these types of applications, disabling Nagle algorithm would improve response time.

When Nagle algorithm is enabled, TCP will wait to send small amounts of data until the acknowledgment for the previous data is received.

When Nagle algorithm is disabled, TCP will send small amounts of data even before the acknowledgment for previous data sent is received.

- The following may be specified for socket level options:

SO-REUSEADDR

Toggles local address reuse. The default is disabled. This alters the normal algorithm used in the bind() call.

The normal bind() call algorithm allows each internet address and port combination to be bound only once. If the address and port have been bound already, a subsequent bind() will fail and result error EADDRINUSE.

After the 'SO_REUSEADDR' option is active, the following situations are supported:

– A server can bind() the same port multiple times as long as every invocation uses a different local IP address, and the wildcard address INADDR_ANY is used only one time per port.

– A server with active client connections can be restarted and can bind to its port without having to close all of the client connections.

– For datagram sockets, multicasting is supported so multiple bind() calls can be made to the same class D address and port number.

SO-KEEPALIVE

Toggles the TCP keepalive mechanism for a stream socket. The default is disabled. When activated, the keepalive mechanism periodically sends a packet on an otherwise idle connection. If the remote TCP does not respond to the packet or to retransmissions of the packet, the connection is terminated with the error ETIMEDOUT.

SO-BROADCAST

Toggles the ability to broadcast messages. This option has no meaning for stream sockets.

If SO-BROADCAST is enabled, the program can send broadcast messages over the socket to destinations that support the receipt of packets.

The default is DISABLED.

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.

SO-OOBINLINE
Toggles the ability to receive out-of-band data. This option has meaning only for stream sockets.

- When this option is enabled, out-of-band data is placed in the normal data input queue as it is received, and is available to a RECVFROM or a RECV call whether or not the OOB flag is set in the call.
- When this option is disabled, out-of-band data is placed in the priority data input queue as it is received and is available to a RECV or a RECVFROM call only when the OOB flag is set.

The default is DISABLED.

SO_SNDBUF
Sets the size of the data portion of the TCP/IP send buffer in OPTVAL. The size of the data portion of the send buffer is protocol-specific.

SO_RCVBUF
Sets the size of the data portion of the TCP/IP receive buffer in OPTVAL. The size of the data portion of the receive buffer is protocol-specific.

OPTVAL
Contains data that further defines the option specified in OPTNAME.

- For OPTNAME of SO-BROADCAST, SO-OOBINLINE, and SO-REUSEADDR, OPTVAL is a one-word binary integer. Set OPTVAL to a nonzero positive value to enable the option; set OPTVAL to 0 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 0 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 “Appendix A. Return Codes” on page 193 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 SHUTDOWN call can be used to close one-way traffic while completing data transfer in the other direction. The HOW parameter determines the direction of traffic to shutdown.

When the CLOSE call is used, the SETSOCKOPT OPTVAL LINGER parameter determines the amount of time the system will wait before releasing the connection. For example, with a LINGER value of 30 seconds, system resources (including the IMS or CICS transaction) will remain in the system for up to 30 seconds after the CLOSE call is issued. In high volume, transaction-based systems like CICS and IMS, this can impact performance severely.

If the SHUTDOWN call is issued when the CLOSE call is received, the connection can be closed immediately, rather than waiting for the 30 second delay.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 49 on page 130 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-FROM      VALUE  0.
        88  END-TO        VALUE  1.
        88  END-BOTH      VALUE  2.
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

PROCEDURE
     CALL 'EZASOKET' USING SOC-FUNCTION S HOW ERRNO RETCODE.
```

Figure 49. SHUTDOWN Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

Parameter Values Set by the Application

SOC-FUNCTION
A 16-byte character field containing SHUTDOWN. The field is left-justified and padded on the right with blanks.

S A halfword binary number set to the socket descriptor of the socket to be shutdown.

HOW A fullword binary field. Set to specify whether all or part of a connection is to be shut down. The following values can be set:

Value	Description
0 (END-FROM)	Ends further receive operations.
1 (END-TO)	Ends further send operations.
2 (END-BOTH)	Ends further send and receive operations.

Parameter Values Returned to the Application

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix A. Return Codes” on page 193 for information about ERRNO return codes.

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

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

SOCKET

The SOCKET call creates an endpoint for communication and returns a socket descriptor representing the endpoint.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.

Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 50 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.
        88  RAW           VALUE  3.
    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 50. SOCKET Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

Parameter Values Set by the Application

SOC-FUNCTION

A 16-byte character field containing 'SOCKET'. The field is left-justified and padded on the right with blanks.

AF A fullword binary field set to the addressing family. For TCP/IP the value is set to 2 for AF_INET.

SOCTYPE

A fullword binary field set to the type of socket required. The types are:

Value	Description
1	Stream sockets provide sequenced, two-way byte streams that are reliable and connection-oriented. They support a mechanism for out-of-band data.
2	Datagram sockets provide datagrams, which are connectionless messages of a fixed maximum length whose reliability is not guaranteed. Datagrams can be corrupted, received out of order, lost, or delivered multiple times.
3	Raw sockets provide the interface to internal protocols (such as IP and ICMP).

PROTO

A fullword binary field set to the protocol to be used for the socket. If this field is set to 0, the default protocol is used. For streams, the default is TCP; for datagrams, the default is UDP.

PROTO numbers are found in the *hlq*.etc.proto data set.

Parameter Values Returned to the Application

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix A. Return Codes” on page 193 for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

Value	Description
> or = 0	Contains the new socket descriptor.
–1	Check **ERRNO** for an error code.

TAKESOCKET

The TAKESOCKET call acquires a socket from another program and creates a new socket. Typically, a child server issues this call using client ID and socket descriptor data that it obtained from the concurrent server. See “GIVESOCKET” on page 91 for a discussion of the use of GETSOCKET and TAKESOCKET calls.

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 51 on page 133 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 51. TAKESOCKET Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

Parameter Values Set by the Application

SOC-FUNCTION
A 16-byte character field containing TAKESOCKET. The field is left-justified and padded to the right with blanks.

SOCRECV
A halfword binary field set to the descriptor of the socket to be taken. The socket to be taken is passed by the concurrent server.

CLIENT
Specifies the client ID of the program that is giving the socket. In CICS and IMS, these parameters are passed by the Listener program to the program that issues the TAKESOCKET call.

- In CICS, the information is obtained using EXEC CICS RETRIEVE.
- In IMS, the information is obtained by issuing GU TIM.

DOMAIN
A fullword binary field set to domain of the program giving the socket. It is always 2, indicating AF_INET.

NAME Specifies an 8-byte character field set to the MVS™ address space identifier of the program that gave the socket.

TASK Specifies an 8-byte character field set to the task identifier of the task that gave the socket.

RESERVED
A 20-byte reserved field. This field is required, but not used.

Parameter Values Returned to the Application

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix A. Return Codes” on page 193 for information about ERRNO return codes.

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

Value	Description
≥ 0	Contains the new socket descriptor.
–1	Check **ERRNO** for an error code.

TERMAPI

This call terminates the session created by INITAPI.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 52 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 52. TERMAPI Call Instruction Example

For equivalent PL/I and assembler language declarations, see "Converting Parameter Descriptions" on page 63.

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 53 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 53. WRITE Call Instruction Example

For equivalent PL/I and assembler language declarations, see "Converting Parameter Descriptions" on page 63.

Parameter Values Set by the Application

SOC-FUNCTION
A 16-byte character field containing WRITE. The field is left-justified and padded on the right with blanks.

S A halfword binary field set to the socket descriptor.

NBYTE
A fullword binary field set to the number of bytes of data to be transmitted.

BUF Specifies the buffer containing the data to be transmitted.

Parameter Values Returned to the Application

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See "Appendix A. Return Codes" on page 193 for information about ERRNO return codes.

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

Value Description

≥0 A successful call. A return code greater than 0 indicates the number of bytes of data written.

–1 Check **ERRNO** for an error code.

WRITEV

The WRITEV function writes data on a socket from a set of buffers.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key.
Dispatchable unit mode:	Task.
Cross memory mode:	PASN = HASN.
Amode:	31-bit or 24-bit. **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 61.
ASC mode:	Primary address space control (ASC) mode.
Interrupt status:	Enabled for interrupts.
Locks:	Unlocked.
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Figure 54 shows an example of WRITEV call instructions.

```
WORKING-STORAGE SECTION.
01  SOKET-FUNCTION         PIC X(16) VALUE 'WRITEV'.
01  S                      PIC 9(4)  BINARY.
01  IOVCNT                 PIC 9(4)  BINARY.

01  IOV.
    03 BUFFER-ENTRY OCCURS N TIMES.
      05 BUFFER_ADDR       POINTER.
      05 RESERVED          PIC X(4).
      05 BUFFER_LENGTH     PIC 9(4).

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

PROCEDURE

    SET BUFFER-POINTER(1) TO ADDRESS-OF BUFFER1.
    SET BUFFER-LENGTH(1)  TO LENGTH-OF  BUFFER1.
    SET BUFFER-POINTER(2) TO ADDRESS-OF BUFFER2.
    SET BUFFER-LENGTH(2)  TO LENGTH-OF  BUFFER2.
    "   "                 "  "          "
    "   "                 "  "          "
    SET BUFFER-POINTER(n) TO ADDRESS-OF BUFFERn.
    SET BUFFER-LENGTH(n)  TO LENGTH-OF  BUFFERn.

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

Figure 54. WRITEV Call Instruction Example

For equivalent PL/I and assembler language declarations, see "Converting Parameter Descriptions" on page 63.

Parameter Values Set by the Application

S A value or the address of a halfword binary number specifying the descriptor of the socket from which the data is to be written.

IOV An array of tripleword structures with the number of structures equal to the value in IOVCNT and the format of the structures as follows:

Fullword 1
The address of a data buffer.

Fullword 2
Reserved.

Fullword 3
The length of the data buffer referenced in Fullword 1.

IOVCNT
A fullword binary field specifying the number of data buffers provided for this call.

Parameters Returned by the Application

ERRNO
A fullword binary field. If RETCODE is negative, this contains an error number. See “Appendix A. Return Codes” on page 193 for information about ERRNO return codes.

RETCODE
A fullword binary field.

Value	Meaning
<0	Check **ERRNO** for an error code.
0	Connection partner has closed connection.
>0	Number of bytes sent.

Using Data Translation Programs for Socket Call Interface

In addition to the socket calls, you can use the following utility programs to translate data:

Data Translation

TCP/IP hosts and networks use ASCII data notation; MVS TCP/IP and its subsystems use EBCDIC data notation. In situations where data must be translated from one notation to the other, you can use the following utility programs:

- EZACIC04 translates EBCDIC data to ASCII data.
- 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.

EZACIC04

The EZACIC04 program is used to translate EBCDIC data to ASCII data.

Figure 55 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 55. EZACIC04 Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

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/1, and assembler language programs.

Figure 56 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 56. EZACIC05 Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

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 57 shows an example of EZACIC06 call instructions.

```
WORKING STORAGE
    01  TOKEN PIC X(16) VALUE 'TCPIPBITMASKCOBL
    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 .



PROCEDURE CALL (to convert from character to binary)
     CALL 'EZACIC06' USING TOKEN
                           BIT-MASK
                           CHAR-MASK
                           BIT-MASK-LENGTH
                           RETCODE.


PROCEDURE CALL (to convert from binary to character)
     CALL 'EZACIC06' USING TOKEN
                           BIT-MASK
                           CHAR-MASK
                           BIT-MASK-LENGTH
                           RETCODE.
```

Figure 57. EZACIC06 Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

TOKEN
Specifies a 16-character identifier. This identifier is required and it must be the first parameter in the list.

CH-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 rightmost bit representing socket 0. The socket positions in the character array are indexed starting with one, which makes

socket 0 index number one in the character array. You should keep this in mind when turning character positions on and off.

Commands

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.

RETCODE

A binary field that returns one of the following:

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

Examples: If you want to use the SELECT call to test sockets 0, 5, and 9, and you are using a character array to represent the sockets, you must set the appropriate characters in the character array to 1. In this example, index positions 1, 6 and 10 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 0, 5, and 9 (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 TOKEN CTOB BIT-MASK CH-MASK
      BIT-LENGTH RETCODE.
MOVE BIT-MASK TO ....
```

When the select call returns and you want to check the bit-mask string for socket activity, enter the following instructions.

```
MOVE ..... TO BIT-MASK.
CALL 'EZACIC06' USING TOKEN BTOC BIT-MASK CH-MASK
         BIT-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/1 or assembler language, the HOSTENT structure can be processed in a relatively straight-forward manner. However, if you are coding in COBOL, HOSTENT can be more difficult to process and you should use the EZACIC08 subroutine to process it for you.

It works as follows:

1. GETHOSTBYADDR or GETHOSTBYNAME returns a HOSTENT structure that indirectly addresses the lists of alias names and internet addresses.
2. Upon return from GETHOSTBYADDR or GETHOSTBYNAME, your program calls EZACIC08 and passes it the address of the HOSTENT structure. EZACIC08 processes the structure and returns the following:
   - The length of host name, if present
   - The host name
   - The number of alias names for the host
   - The alias name sequence number
   - The length of the alias name
   - The alias name
   - The host internet address type, always 2 for AF_INET
   - The host internet address length, always 4 for AF_INET
   - The number of host internet addresses for this host
   - The host internet address sequence number
   - The host internet address
3. If the GETHOSTBYADDR or GETHOSTBYNAME call returns more than one alias name or host internet address (or above), the application program should repeat the call to EZACIC08 until all alias names and host internet addresses have been retrieved.

Figure 58 on page 143 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 58. EZAZIC08 Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 63.

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 1 to the current value of HOSTALIAS-SEQ and uses the resulting value to index into the table of alias names. Therefore, for a given instance of GETHOSTBYxxxx, this field should be set to 0 for the initial call to EZACIC08. For all subsequent calls to EZACIC08, this field should contain the HOSTALIAS-SEQ number returned by the previous invocation.

HOSTADDR-SEQ

This halfword field is used by EZACIC08 to index the list of IP addresses. When EZACIC08 is called, it adds 1 to the current value of HOSTADDR-SEQ and uses the resulting value to index into the table of IP addresses. Therefore, for a given instance of GETHOSTBYxxxx, this field should be set to 0 for the initial call to EZACIC08. For all subsequent calls to EZACIC08, this field should contain the HOSTADDR-SEQ number returned by the previous call.

Parameter Values Returned to the Application

HOSTNAME-LENGTH
This halfword binary field contains the length of the host name (if host name was returned).

HOSTNAME-VALUE
This 255-byte character string contains the host name (if host name was returned).

HOSTALIAS-COUNT
This halfword binary field contains the number of alias names returned.

HOSTALIAS-SEQ
This halfword binary field is the sequence number of the alias name currently found in HOSTALIAS-VALUE.

HOSTALIAS-LENGTH
This halfword binary field contains the length of the alias name currently found in HOSTALIAS-VALUE.

HOSTALIAS-VALUE
This 255-byte character string contains the alias name returned by this instance of the call. The length of the alias name is contained in HOSTALIAS-LENGTH.

HOSTADDR-TYPE
This halfword binary field contains the type of host address. For FAMILY type AF_INET, HOSTADDR-TYPE is always 2.

HOSTADDR-LENGTH
This halfword binary field contains the length of the host internet address currently found in HOSTADDR-VALUE. For FAMILY type AF_INET, HOSTADDR-LENGTH is always set to 4.

HOSTADDR-COUNT
This halfword binary field contains the number of host internet addresses returned by this instance of the call.

HOSTADDR-SEQ
This halfword binary field contains the sequence number of the host internet address currently found in HOSTADDR-VALUE.

HOSTADDR-VALUE
This fullword binary field contains a host internet address.

RETURN-CODE
This fullword binary field contains the EZACIC08 return code:

Value	Description
0	Successful completion.
-1	HOSTENT address is not valid.

Call Interface PL/1 Sample Programs

This section provides sample programs for the call interface that you can use for a PL/1 application program.

The following are the sample programs available in the *hlq*.SEZAINST data set:

Program	Description
EZASOKPS	PL/1 call interface sample server program
EZASOKPC	PL/1 call interface sample client program
CBLOCK	PL/1 common variables

Sample Code for Server Program

The EZASOKPS PL/I sample program is a server program that shows you how to use the following calls:

- INITAPI
- SOCKET
- BIND
- GETSOCKNAME
- LISTEN
- ACCEPT
- READ
- WRITE
- CLOSE
- TERMAPI

```
EZASOKPS: PROC OPTIONS(MAIN);
/* INCLUDE CBLOCK - common variables                                  */
% include CBLOCK;
open file(driver);
/*********************************************************************/
/*                                                                   */
/* Execute INITAPI                                                   */
/*                                                                   */
/*********************************************************************/
call ezasoket(INITAPI, MAXSOC, ID, SUBTASK,
                      MAXSNO, ERRNO, RETCODE);
if retcode < 0 then do;
   msg = 'FAIL: initapi' || errno;
   write file(driver) from (msg);
   goto getout;
end;
/*********************************************************************/
/*                                                                   */
/* Execute SOCKET                                                    */
/*                                                                   */
/*********************************************************************/
call ezasoket(SOCKET, AF_INET, TYPE_STREAM, PROTO,
                     ERRNO, RETCODE);
if retcode < 0 then do;
   msg = blank;                        /* clear field               */
   msg = 'FAIL: socket, stream, internet' || errno;
   write file(driver) from (msg);
   goto getout;
end;
else sock_stream = retcode;
/*********************************************************************/
/*                                                                   */
/* Execute BIND                                                      */
/*                                                                   */
/*********************************************************************/
name_id.port = 8888;
name_id.address = '01234567'BX;       /* internet address            */
call ezasoket(BIND, SOCK_STREAM, NAME_ID,
                    ERRNO, RETCODE);
if retcode < 0 then do;
   msg = blank;                        /* clear field               */
   msg = 'FAIL: bind' || errno;
   write file(driver) from (msg);
   goto getout;
end;
/*********************************************************************/
/*                                                                   */
/* Execute GETSOCKNAME                                               */
```

```
/*                                                                  */
/********************************************************************/
name_id.port = 8888;
name_id.address = '01234567'BX;       /* internet address           */
call ezasoket(GETSOCKNAME, SOCK_STREAM,
                    NAME_ID, ERRNO, RETCODE);
msg = blank;                          /* clear field                */
if retcode < 0 then do;
   msg = 'FAIL: getsockname, stream, internet' || errno;
   write file(driver) from (msg);
end;
else do;
   msg = 'getsockname = ' || name_id.address;
   write file(driver) from (msg);
end;
/********************************************************************/
/*                                                                  */
/* Execute LISTEN                                                   */
/*                                                                  */
/********************************************************************/
backlog = 5;
call ezasoket(LISTEN, SOCK_STREAM, BACKLOG,
                    ERRNO, RETCODE);
if retcode < 0 then do;
   msg = blank;                       /* clear field                */
   msg = 'FAIL: listen w/ backlog = 5' || errno;
   write file(driver) from (msg);
   goto getout;
end;
/********************************************************************/
/*                                                                  */
/* Execute ACCEPT                                                   */
/*                                                                  */
/********************************************************************/
name_id.port = 8888;
name_id.address = '01234567'BX;       /* internet address           */
call ezasoket(ACCEPT, SOCK_STREAM,
                    NAME_ID, ERRNO, RETCODE);
msg = blank;                          /* clear field                */
if retcode < 0 then do;
   msg = 'FAIL: accept' || errno;
   write file(driver) from (msg);
end;
else do;
   accpsock = retcode;
   msg = 'accept socket = ' || accpsock;
   write file(driver) from (msg);
end;
/********************************************************************/
/*                                                                  */
/* Execute READ                                                     */
/*                                                                  */
/********************************************************************/
nbyte = length(bufin);
call ezasoket(READ, ACCPSOCK,
                    NBYTE, BUFIN, ERRNO, RETCODE);
msg = blank;                          /* clear field                */
if retcode < 0 then do;
   msg = 'FAIL: read' || errno;
   write file(driver) from (msg);
end;
else do;
   msg = 'read = ' || bufin;
   write file(driver) from (msg);
   bufout = bufin;
   nbyte = length(bufout);
end;
```

```
/*********************************************************************/
/*                                                                   */
/* Execute WRITE                                                     */
/*                                                                   */
/*********************************************************************/
call ezasoket(WRITE, ACCPSOCK, NBYTE, BUFOUT,
                     ERRNO, RETCODE);
msg = blank;                          /* clear field                 */
if retcode < 0 then do;
   msg = 'FAIL: write' || errno;
   write file(driver) from (msg);
end;
else do;
   msg = 'write = ' || bufout;
   write file(driver) from (msg);
end;
/*********************************************************************/
/*                                                                   */
/* Execute CLOSE accept socket                                       */
/*                                                                   */
/*********************************************************************/
call ezasoket(CLOSE, ACCPSOCK,
                     ERRNO, RETCODE);
if retcode < 0 then do;
   msg = blank;                       /* clear field                 */
   msg = 'FAIL: close, accept sock' || errno;
   write file(driver) from (msg);
end;
/*********************************************************************/
/*                                                                   */
/* Execute TERMAPI                                                   */
/*                                                                   */
/*********************************************************************/
getout:
call ezasoket(TERMAPI);
close file(driver);
end ezasokps;
```

Sample Program for Client Program

The EZASOKPC PL/I sample program is a client program that shows you how to use the following calls provided by the call socket interface:

- INITAPI
- SOCKET
- CONNECT
- GETPEERMANE
- WRITE
- READ
- SHUTDOWN
- TERMAPI

```
EZASOKPC: PROC OPTIONS(MAIN);
/* INCLUDE CBLOCK - common variables                                 */
% include CBLOCK;
open file(driver);
/*********************************************************************/
/*                                                                   */
/* Execute INITAPI                                                   */
/*                                                                   */
/*********************************************************************/
call ezasoket(INITAPI, MAXSOC, ID, SUBTASK,
                       MAXSNO, ERRNO, RETCODE);
if retcode < 0 then do;
   msg = 'FAIL: initapi' || errno;
   write file(driver) from (msg);
   goto getout;
```

```
end;
/*********************************************************************/
/*                                                                   */
/* Execute SOCKET                                                    */
/*                                                                   */
/*********************************************************************/
call ezasoket(SOCKET, AF_INET, TYPE_STREAM, PROTO,
                     ERRNO, RETCODE);
if retcode < 0 then do;
   msg = blank;                        /* clear field               */
   msg = 'FAIL: socket, stream, internet' || errno;
   write file(driver) from (msg);
   goto getout;
end;
sock_stream = retcode;                 /* save socket descriptor    */
/*********************************************************************/
/* Execute CONNECT                                                   */
/*                                                                   */
/*********************************************************************/
name_id.port = 8888;
name_id.address = '01234567'BX;        /* internet address          */
call ezasoket(CONNECT, SOCK_STREAM, NAME_ID,
                     ERRNO, RETCODE);
if retcode < 0 then do;
   msg = blank;                        /* clear field               */
   msg = 'FAIL: connect, stream, internet' || errno;
   write file(driver) from (msg);
   goto getout;
end;
/*********************************************************************/
/*                                                                   */
/*   Execute GETPEERNAME                                             */
/*                                                                   */
/*********************************************************************/
call ezasoket(GETPEERNAME, SOCK_STREAM,
                     NAME_ID, ERRNO, RETCODE);
msg = blank;                           /* clear field               */
if retcode < 0 then do;
   msg = 'FAIL: getpeername' || errno;
   write file(driver) from (msg);
end;
else do;
   msg = 'getpeername =' || name_id.address;
   write file(driver) from (msg);
end;
/*********************************************************************/
/*                                                                   */
/* Execute WRITE                                                     */
/*                                                                   */
/*********************************************************************/
bufout = message;
nbyte = length(message);
call ezasoket(WRITE, SOCK_STREAM, NBYTE, BUFOUT,
                     ERRNO, RETCODE);
msg = blank;                           /* clear field               */
if retcode < 0 then do;
   msg = 'FAIL: write' || errno;
   write file(driver) from (msg);
end;
else do;
   msg = 'write = ' || bufout;
   write file(driver) from (msg);
end;
/*********************************************************************/
/*                                                                   */
/* Execute READ                                                      */
/*                                                                   */
```

```
/*********************************************************************/
nbyte = length(bufin);
call ezasoket(READ, SOCK_STREAM,
                  NBYTE, BUFIN, ERRNO, RETCODE);
msg = blank;                          /* clear field                 */
if retcode < 0 then do;
   msg = 'FAIL: read' || errno;
   write file(driver) from (msg);
end;
else do;
   msg = 'read = ' || bufin;
   write file(driver) from (msg);
end;
/*********************************************************************/
/*                                                                   */
/* Execute SHUTDOWN from/to                                          */
/*                                                                   */
/*********************************************************************/
getout:
how = 2;
call ezasoket(SHUTDOWN, SOCK_STREAM, HOW,
                   ERRNO, RETCODE);
if retcode < 0 then do;
   msg = blank;                       /* clear field                 */
   msg = 'FAIL: shutdown' || errno;
   write file(driver) from (msg);
end;
/*********************************************************************/
/*                                                                   */
/* Execute TERMAPI                                                   */
/*                                                                   */
/*********************************************************************/
call ezasoket(TERMAPI);
close file(driver);
end ezasokpc;
```

Common Variables Used in PL/1 Sample Programs

The CBLOCK common storage area contains the variables that are used in the PL/1 programs in this section.

```
/*********************************************************************/
/*                                                                   */
/* SOKET COMMON VARIABLES                                            */
/*                                                                   */
/*********************************************************************/
DCL ABS     BUILTIN;
DCL ADDR    BUILTIN;
DCL ACCEPT  CHAR(16) INIT('ACCEPT');
DCL ACCPSOCK FIXED BIN(15);           /* temporary ACCEPT socket    */
DCL AF_INET FIXED BIN(31) INIT(2);    /* internet domain            */
DCL AF_IUCV FIXED BIN(31) INIT(17);   /* iucv domain                */
DCL ALIAS   CHAR(255);                /* alternate NAME             */
DCL APITYPE FIXED BIN(15) INIT(2);    /* default API type           */
DCL BACKLOG FIXED BIN(31);            /* max length of pending queue*/
DCL BADNAME CHAR(20);                 /* temporary name             */
DCL BIND    CHAR(16) INIT('BIND');
DCL BIT     BUILTIN;
DCL BITZERO BIT(1);                   /* bit zero value             */
DCL BLANK   CHAR(255) INIT(' ');      /*                            */
DCL BUF     CHAR(80) INIT(' ');       /* macro READ/WRITE buffer    */
DCL BUFF  CHAR(15)         INIT(' '); /* short buffer               */
DCL BUFFER   CHAR(32767) INIT(' ');   /* BUFFER                     */
DCL BUFIN    CHAR(32767) INIT(' ');   /* Read buffer                */
DCL BUFOUT   CHAR(32767) INIT(' ');   /* WRITE buffer               */
DCL 1 CLIENT,                      /* socket addr of connection peer */
      2 DOMAIN FIXED BIN(31) INIT(2), /* domain of client (AF_INET) */
```

```
      2 NAME   CHAR(8) INIT(' '),     /* addr identifier for client */
      2 TASK   CHAR(8) INIT(' '),     /* task identifier for client */
      2 RESERVED CHAR(20) INIT(' ');  /* reserved                   */
DCL CLOSE   CHAR(16) INIT('CLOSE');
DCL COMMAND FIXED BIN(31) INIT(3);    /* Query FNDELAY flag         */
DCL CONNECT CHAR(16) INIT('CONNECT');
DCL COUNT FIXED BIN(31) INIT(100);    /* elements in GRP_IOCTL_TABLE*/
DCL DATA_SOCK FIXED BIN(15);          /* temporary datagram socket  */
DCL DEF     FIXED BIN(31) INIT(0);    /* default protocol           */
DCL DRIVER  FILE OUTPUT UNBUF ENV(FB RECSIZE(100)) RECORD;
DCL ERETMSK CHAR(4);                 /* indicate exception events  */
DCL ERR     FIXED BIN(31);            /* error number variable      */
DCL ERRNO   FIXED BIN(31) INIT(0);    /*  error number              */
DCL ESNDMSK CHAR(4);                  /* check for pending          */
                                      /*   exception events         */
DCL EXIT    LABEL;                    /* common exit point          */
DCL EZACIC05 ENTRY OPTIONS(RETCODE,ASM,INTER) EXT;
DCL EZASOKET ENTRY OPTIONS(RETCODE,ASM,INTER) EXT; /* socket call   */
DCL FCNTL  CHAR(16) INIT('FCNTL');
DCL FIONBIO FIXED BIN(31) INIT(-2147178626);/* flag: nonblocking    */
DCL FIONREAD FIXED BIN(31) INIT(+1074046847);/* flag:#readable bytes*/
DCL FLAGS   FIXED BIN(31) INIT(0);    /* default: no flags          */
                                      /*   1 = OOB, SEND OUT-OF-BAND*/
                                      /*   4 = DON'T ROUTE          */
DCL GETCLIENTID  CHAR(16) INIT('GETCLIENTID');
DCL GETHOSTBYADDR  CHAR(16) INIT('GETHOSTBYADDR');
DCL GETHOSTBYNAME CHAR(16) INIT('GETHOSTBYNAME');
DCL GETHOSTNAME CHAR(16) INIT('GETHOSTNAME');
DCL GETHOSTID CHAR(16) INIT('GETHOSTID');
DCL GETIBMOPT CHAR(16) INIT('GETIBMOPT');
DCL GETPEERNAME CHAR(16) INIT('GETPEERNAME');
DCL GETSOCKNAME CHAR(16) INIT('GETSOCKNAME');
DCL GETSOCKOPT CHAR(16) INIT('GETSOCKOPT');
DCL GIVESOCKET  CHAR(16) INIT('GIVESOCKET');
DCL GLOBAL CHAR(16) INIT('GLOBAL');
DCL HOSTADDR FIXED BIN(31);           /* host internet address      */
DCL HOW     FIXED BIN(31) INIT(2);    /* how shutdown is to be done */
DCL I       FIXED BIN(15);            /* loop index                 */
DCL ICMP    FIXED BIN(31) INIT(2);    /* prototype icmp  ???        */
DCL 1 ID,                             /*                            */
      2 TCPNAME CHAR(8) INIT('TCPIP'), /* remote address space      */
      2 ADSNAME CHAR(8) INIT('USER19J'); /* local address space     */
DCL IDENT   POINTER;                  /* TCP/IP Addr Space          */
DCL IFCONF  CHAR(255);                /* configuration structure    */
DCL IFREQ   CHAR(255);                /* interface structure        */
DCL INDEX   BUILTIN;
DCL IOCTL  CHAR(16) INIT('IOCTL');
DCL IOCTL_CMD FIXED BIN(31);          /* ioctl command              */
DCL IOCTL_REQARG  POINTER ;           /* send   pointer to data area*/
DCL IOCTL_RETARG  POINTER ;           /* return pointer to data area*/
DCL IOCTL_REQ00   FIXED BIN(31);      /* command request argument   */
DCL IOCTL_REQ04   FIXED BIN(31);      /* command request argument   */
DCL IOCTL_REQ08   FIXED BIN(31);      /* command request argument   */
DCL IOCTL_REQ32   CHAR(32) INIT(' '); /* command request argument   */
DCL IOCTL_RET00   FIXED BIN(31);      /* command return  argument   */
DCL IOCTL_RET04   FIXED BIN(31);      /* command return  argument   */
DCL INITAPI CHAR(16) INIT('INITAPI'); /*                            */
DCL 1 INTERNET,                       /* internet address           */
      2 NETID1 FIXED BIN(31) INIT(9), /* network id, part 1         */
      2 NETID2 FIXED BIN(31) INIT(67), /* network id, part 2        */
      2 SUBNETID FIXED BIN(31) INIT(30), /* subnet id               */
      2 HOSTID FIXED BIN(31) INIT(137); /* host id                 */
DCL IP      FIXED BIN(31) INIT(1);    /* prototype ip   ???         */
DCL J       FIXED BIN(15);            /* loop index                 */
DCL K       FIXED BIN(15);            /* loop index                 */
DCL LENGTH  BUILTIN;
DCL LABL    CHAR(9);
```

```
DCL LISTEN  CHAR(16) INIT('LISTEN');
DCL MAXSNO  FIXED BIN(31) INIT(0);    /* max descriptor assigned    */
DCL MAXSOC  FIXED BIN(15) INIT(255);  /* largest socket # checked   */
DCL MESSAGE CHAR(50) INIT('I love my 1 @ Rottweiler!'); /* message  */
DCL MSG     CHAR(100) INIT(' ');      /*   message text             */
DCL 1 NAME_ID,                    /* socket addr of connection peer */
      2 FAMILY FIXED BIN(15) INIT(2), /*addr'g family; TCP/IP def   */
      2 PORT   FIXED BIN(15),         /* system assigned port #     */
      2 ADDRESS FIXED BIN(31),        /* 32-bit internet           */
      2 RESERVED CHAR(8);             /* reserved                  */
DCL NAMEL   CHAR(255)    VARYING;     /* name field, long          */
DCL NAMES   CHAR(24);                 /* name field, short         */
DCL NAMELEN FIXED BIN(31);            /* length of name/alias field */
DCL NBYTE   FIXED BIN(31);            /* Number of bytes in buffer  */
DCL NOTE(3) CHAR(25) INIT('Now is the time for 198 g',
                          'ood people to come to the',
                          ' aid of their parties!');
DCL NS      FIXED BIN(15);            /* socket descriptor, new     */
DCL NULL    BUILTIN;
DCL OPTL    FIXED BIN(31);            /* length of OPTVAL string    */
DCL OPTLEN  FIXED BIN(31);            /* length of OPTVAL string    */
DCL OPTN    CHAR(15);                 /* OPTNAME value (macro)      */
DCL OPTNAME FIXED BIN(31);            /* OPTNAME value (call)       */
DCL OPTVAL  CHAR(255);                /* GETSOCKOPT option data     */
DCL OPTVALD FIXED BIN(31);            /* SETSOCKOPT option data     */
DCL 1 OPT_STRUC,                      /* structure for option       */
      2 ON_OFF FIXED BIN(31) INIT(1), /* enable option              */
      2 TIME   FIXED BIN(31) INIT(5); /* time-out in seconds        */
DCL 1 OPT_STRUCT,                     /* structure for option       */
      2 ON     FIXED BIN(31),         /* used for getsockopt        */
      2 TIMEOUT FIXED BIN(31);        /* time-out in seconds        */
DCL PLITEST BUILTIN;                  /* debug tool                 */
DCL PROTO   FIXED BIN(31) INIT(0);    /* prototype default          */
DCL READ    CHAR(16) INIT('READ');
DCL READV   CHAR(16) INIT('READV');
DCL RECV    CHAR(16) INIT('RECV');
DCL RECVFROM CHAR(16) INIT('RECVFROM');
DCL RECVMSG CHAR(16) INIT('RECVMSG');
DCL REUSE   FIXED BIN(31) INIT('4');  /* toggle, reuse local addr   */
DCL REQARG  FIXED BIN(31);            /* command request argument   */
DCL RETC    FIXED BIN(31);            /* return code variable       */
DCL RETARG  CHAR(255);                /* return argument data area  */
DCL RETCODE FIXED BIN(31) INIT(0);    /* return code                */
DCL RETLEN  FIXED BIN(31);            /* return area data length    */
DCL RRETMSK CHAR(4);                 /* indicate READ EVENTS       */
DCL RSNDMSK CHAR(4):               /* check for pending read events */
DCL RTENTRY CHAR(50) INIT('dummy table'); /* router entry           */
DCL SAVEFAM  FIXED BIN(15);           /* temporary family name      */
DCL SELECB CHAR(4) INIT('1');
DCL SELECT CHAR(16) INIT('SELECT');
DCL SELECTEX CHAR(16) INIT('SELECTEX');
DCL SEND CHAR(16) INIT('SEND');
DCL SENDMSG CHAR(16) INIT('SENDMSG');
DCL SENDTO CHAR(16) INIT('SENDTO');
DCL SETSOCKOPT CHAR(16) INIT('SETSOCKOPT');
DCL SHUTDOWN CHAR(16) INIT('SHUTDOWN');
DCL SIOCADDRT FIXED BIN(31) INIT(-2144295158);
                                      /* flag: add routing entry*/
DCL SIOCATMARK FIXED BIN(31) INIT(+1074046727);
                                       /* flag: out-of-band data*/
DCL SIOCDELRT FIXED BIN(31) INIT(-2144295157);
                                      /* flag: delete routing   */
DCL SIOCGIFADDR FIXED BIN(31) INIT(-1071601907);
                                        /*flag: network int addr*/
DCL SIOCGIFBRDADDR FIXED BIN(31) INIT(-1071601902);
                                          /*flag  net broadcast*/
DCL SIOCGIFCONF FIXED BIN(31) INIT(-1073174764);
```

```
                                          /* flag: netw int config*/
DCL SIOCGIFDSTADDR FIXED BIN(31) INIT(-1071601905);
                                           /* flag: net des addr*/
DCL SIOCGIFFLAGS FIXED BIN(31) INIT(-1071601903);
                                         /* flag: net intf flags*/
DCL SIOCGIFMETRIC FIXED BIN(31) INIT(-1071601897);
                                          /* flag: get rout metr*/
DCL SIOCGIFNETMASK FIXED BIN(31) INIT(-1071601899);
                                           /* flag: network mask*/
DCL SIOCGIFNONSENSE FIXED BIN(31) INIT(-1234567890);
                                            /* flag: nonsense   */
DCL SIOCSIFMETRIC FIXED BIN(31) INIT(-2145343720);
                                           /* flag: set rout metr*/
DCL SOCK     FIXED BIN(15);           /* socket descriptor          */
DCL SOCKET  CHAR(16) INIT('SOCKET');
DCL SOCK_DATAGRAM FIXED BIN(15);      /* socket descriptor datagram */
DCL SOCK_RAW FIXED BIN(15);           /* socket descriptor raw      */
DCL SOCK_STREAM FIXED BIN(15);        /* stream socket descriptor   */
DCL SOCK_STREAM_1 FIXED BIN(15);      /* stream socket descriptor   */
DCL SO_BROADCAST FIXED BIN(31) INIT(32); /* toggle, broadcast msg   */
DCL SO_ERROR FIXED BIN(31) INIT(4103); /* check/clear async error   */
DCL SO_KEEPALIVE FIXED BIN(31) INIT(8); /* request status of stream*/
DCL SO_LINGER  FIXED BIN(31) INIT(128); /* toggle, linger on close  */
DCL SO_OOBINLINE FIXED BIN(31) INIT(256);/*toggle, out-of-bound data*/
DCL SO_REUSEADDR FIXED
              BIN(31) INIT(4);       /* toggle, local address reuse*/
DCL SO_SNDBUF  FIXED BIN(31) INIT(4097);
DCL SO_TYPE FIXED BIN(31) INIT(4104); /* return type of socket      */
DCL STRING  BUILTIN;
DCL SUBSTR  BUILTIN;
DCL SUBTASK CHAR(8) INIT('ANYNAME');  /* task/path identifier       */
DCL SYNC    CHAR(16) INIT('SYNC');
DCL TAKESOCKET CHAR(16) INIT('TAKESOCKET');
DCL TASK    CHAR(16) INIT('TASK');
DCL TERMAPI CHAR(16) INIT('TERMAPI'); /*                           */
DCL TIME    BUILTIN;
DCL 1 TIMEOUT,
      2  TIME_SEC  FIXED BIN(31),     /* value in secs              */
      2  TIME_MSEC FIXED BIN(31);     /* value in millisecs         */
DCL TYPE_DATAGRAM FIXED BIN(31) INIT(2);/*fixed lengthconnectionless*/
DCL TYPE_RAW  FIXED BIN(31) INIT(3); /* internal protocol interface */
DCL TYPE_STREAM FIXED BIN(31) INIT(1); /* two-way byte stream       */
DCL WRETMSK CHAR(4);                 /* indicate WRITE EVENTS      */
DCL WRITE   CHAR(16) INIT('WRITE');
DCL WRITEV   CHAR(16) INIT('WRITEV');
DCL WSNDMSK CHAR(4);               /*check for pending write events */
```

Chapter 8. IMS Listener Samples

This chapter includes sample programs using the IMS Listener. The following samples are included:

IMS TCP/IP Control Statements

This chapter contains examples of the control statements required to define and initiate the various IMS TCP/IP components.

JCL for Starting a Message Processing Region

The following is an example of the JCL that is required to start an IMS message processing region in which TCP/IP servers can operate. Note the STEPLIB statements that point to TCP/IP and the C run-time library. A C run-time library is required when you use the GETHOSTBYADDR or GETHOSTBYNAME call. For more information, see the *z/OS Program Directory* or the section on C compilers and run time libraries in *z/OS Communications Server: IP Application Programming Interface Guide*.

This sample is based on the IMS procedure (DFSMPR). You might have to modify the language run time libraries to match your programming language requirements.

```
//       PROC SOUT=A,RGN=2M,SYS2=,
//            CL1=001,CL2=000,CL3=000,CL4=000,
//            OPT=N,OVLA=0,SPIE=0,VALCK=0,TLIM=00,
//            PCB=000,PRLD=,STIMER=,SOD=,DBLDL=,
//            NBA=,OBA=,IMSID=IMS1,AGN=,VSFX=,VFREE=,
//            SSM=,PREINIT=,ALTID=,PWFI=N,
//            APARM=
//*
//REGION EXEC  PGM=DFSRRC00,REGION=&RGN,;
//             TIME=1440,DPRTY=(12,0),
//             PARM=(MSG,&CL1&CL2&CL3&CL4,;
//             &OPT&OVLA&SPIE&VALCK&TLIM&PCB,;
//             &PRLD,&STIMER,&SOD,&DBLDL,&NBA,;
//             &OBA,&IMSID,&AGN,&VSFX,&VFREE,;
//             &SSM,&PREINIT,&ALTID,&PWFI,;
//             '&APARM')
//&*;
//STEPLIB  DD DSN=IMS31.&SYS2;RESLIB,DISP=SHR
//         DD DSN=IMS31.&SYS2;PGMLIB,DISP=SHR
//         DD DSN=PLI.LL.V2R3M0.SIBMLINK,DISP=SHR
//         DD DSN=PLI.LL.V2R3M0.PLILINK,DISP=SHR
//         DD DSN=C370.LL.V2R2M0.SEDCLINK,DISP=SHR
//*        Use the following for LE/370 C run-time libraries:
//*        DD  DSN=CEE.V1R3M0.SCEERUN,DISP=SHR
//         DD DSN=TCPIP.SEZATCP,DISP=SHR
//PROCLIB  DD DSN=IMS31.&SYS2;PROCLIB,DISP=SHR
//SYSUDUMP DD  SYSOUT=&SOUT,DCB=(LRECL=121,BLKSIZE=3129,RECFM=VBA),;
//             SPACE=(125,(2500,100),RLSE,,ROUND)
//
```

JCL for Linking the IMS Listener

The following examples are JCL that can be used to link the IMS listener.

EZAIMSCZ JCLIN

```
 //EZAIMSCZ JOB (accounting,information),programmer.name,
 //            MSGLEVEL=(1,1),MSGCLASS=A,CLASS=A
 //*************************************************************
 //*NOTE: ANY ZONE UPDATED WITH THE LINK COMMAND OR CROSS-ZONE  *
 //*      INFORMATION CANNOT BE PROCESSED BY SMP/E R6 OR EARLIER*
 //*************************************************************
 //*                                                           *
 //*    Function: Perform SMP/E LINK for IMS module            *
 //*                                                           *
 //*    Instructions:                                          *
 //*         Change all lower case characters to values        *
 //*         suitable for your installation.                   *
 //*                                                           *
 //*      tcptgt  :   TCP/IP Target Zone                       *
 //*      imszone :   IMS Target Zone                          *
 //*                                                           *
 //* This job uses the installation procedure EZAPROC by default*
 //* If you have chosen to use DDDEFs to install TCP/IP, you    *
 //* must perform the following steps:                         *
//*   Delete or comment the 'LNKCZ EXEC PROC=EZAPROC' statement*
//*                                                           *
//*   Uncomment the 'LNKCZ EXEC PROC=EZAPROCD' statement.     *
//*                                                           *
//*   Change the high-level qualifier 'ims' to match the      *
//*   high-level qualifier for your installation's IMS RESLIB  *
//*   data set.                                               *
//*                                                           *
//LNKCZ EXEC  PROC=EZAPROC
//*LNKCZ EXEC  PROC=EZAPROCD
//*************************************************************
//RESLIB   DD DSN=ims.RESLIB,DISP=SHR
//*************************************************************
//*
//SMPCNTL   DD  *
SET BDY(tcptgt).          /* TCP/IP target zone   */
LINK MODULE(DFSLI000)
FROMZONE(imszone)         /* IMS target zone      */
INTOLMOD(EZAIMSLN)
RC(LINK)=00.
```

EZAIMSPL JCLIN

```
//LINKIMS JOB (accounting,information),programmer.name,
//           MSGLEVEL=(1,1),MSGCLASS=A,CLASS=A
//*************************************************************
//*                                                           *
//*   THIS JOB SERVES AS AN ALTERNATIVE TO THE CROSS ZONE LINK *
//*   PERFORMED BY RUNNING EZAIMSCZ.                          *
//*                                                           *
//*   UPDATE THE JOB, SYSLMOD AND RESLIB DD CARDS TO SUIT YOUR *
//*   INSTALLATION .                                          *
//*                                                           *
//*************************************************************
//LNKIMS   EXEC PGM=IEWL,PARM='XREF,LIST,REUS'
//SYSPRINT  DD SYSOUT=*
//SYSUT1    DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSLMOD  DD DSN=tcpip.v3r1.SEZALINK,DISP=SHR
```

```
//RESLIB   DD DSN=ims.RESLIB,DISP=SHR
//SYSLIN    DD *
  ORDER CMCOPYR
  INCLUDE RESLIB(DFSLI000)
  INCLUDE SYSLMOD(EZAIMSLN)
  ENTRY EZAIMSLN
  MODE RMODE(24) AMODE(31)
  NAME EZAIMSLN(R)
/*
```

Listener IMS Definitions

The following statements define the Listener as an IMS BMP application and the PSB that it uses. Note that the name ALTPCB is required.

PSB Definition

```
ALTPCB   PCB    TYPE=TP,MODIFY=YES
         PSBGEN PSBNAME=EZAIMSLN,IOASIZE=1000
         SSASIZE=1000,LANG=ASSEM
```

TRANSACT MODE=SNGL

Application Definition

```
         APPLCTN  PSB=EZAIMSLN,PGMTYPE=BATCH
```

Sample Program Explicit-Mode

The following is an example of an explicit-mode client server program pair. The client program name is EZAIMSC2; you can find it in *hlq*.SEZAINST(EZAIMSC2). The server program name is EZASVAS2; its IMS trancode is DLSI102. You can find the sample in *hlq*.SEZAINST(EZASVAS2).

Program Flow

The client begins execution and obtains the host name and port number from startup parameters. It then issues SOCKET and CONNECT calls to establish connectivity to the specified host and port. Upon successful completion of the connect, the client sends the TRM, which tells the Listener to schedule the specified transaction (DLSI102). The Listener schedules that transaction and places a TIM on the IMS message queue. Finally, it issues a GIVESOCKET call and waits for the server to take the socket.

When the requested server (EZASVAS2) begins execution, it issues a GU call to obtain the TIM. Using addressibility information from the TIM, it issues INITAPI and TAKESOCKET calls. The server then sends SERVER MSG #1 to the client.

When the client receives the message, it displays SERVER MSG #1 on stdout and then sends END CLIENT MSG #2 to the server, and displays a success message on stdout. It then blocks on another receive() until the server responds.

The server, upon receipt of a message with the characters END as the first 3 characters, sends SERVER MSG #2 back to the client and closes the socket.

When the client receives this message, it prints SERVER MSG #2 on stdout, closes the socket, and ends.

Sample Explicit-Mode Client Program (C Language)

```
.* Different than part at level OLDPROD OLDVER/OLDLVL.
/*
 * Include Files.
 */
/*  #define RESOLVE_VIA_LOOKUP */
#pragma runopts(NOSPIE NOSTAE)
#define lim 50
#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>
#include <socket.h>
#include <netdb.h>
#include <stdio.h>
/*
 * Client Main.
 */
main(argc, argv)
int argc;
char **argv;
{
    unsigned short port;       /* port client will connect to         */
    char buf ??(lim??);        /* sned receive buffers 0 -3           */
    char buf1 ??(lim??);
    char buf2 ??(lim??);
    char buf3 ??(lim??);
    struct hostent *hostnm;    /* server host name information        */
    struct sockaddr_in server; /* server address                      */
    int s;                     /* client socket                       */
    /*
     * Check Arguments Passed. Should be hostname and port.
     */
    if (argc != 3)
    {
    /*  fprintf(stderr, "Usage: %s hostname port\n", argvÝ0¨); */
         printf("Usage: %s hostname port\n", argv        Ý0¨);
        exit(1);
    }
         printf("Usage: %s hostname port\n", argv        Ý0¨);
    /*
     * The host name is the first argument. Get the server address.
     */
    hostnm = gethostbyname(argvÝ1¨);
    if (hostnm == (struct hostent *) 0)
    {
    /*  fprintf(stderr, "Gethostbyname failed\n"); */
         printf("Gethostbyname failed\n");
        exit(2);
    }
    /*
     * The port is the second argument.
     */
    port = (unsigned short) atoi(argvÝ2¨);
    /*
     * Put a message into the buffer.
     */
    strcpy(buf,"2000*TRNREQ*DLSI102 ");
    /*
     * Put the server information into the server structure.
     * The port must be put into network byte order.
     */
```

```
server.sin_family      = AF_INET;
server.sin_port        = htons(port);
server.sin_addr.s_addr = *((unsigned long *)hostnm->h_addr);
/*
 * Get a stream socket.
 */
if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
    tcperror("Socket()");
    exit(3);
}
/*
 * Connect to the server.
 */
if (connect(s, (struct sockaddr *)&amp.server, sizeof(server)) < 0)
{
    tcperror("Connect()");
    exit(4);
}
if (send(s, buf, sizeof(buf), 0) < 0)
{
    tcperror("Send()");
    exit(5);
}
printf("send one complete\n");
/*
 * The server sends message #1. Receive it into buffer1
 */
if (recv(s, buf1, sizeof(buf1), 0) < 0)
{
    tcperror("Recv()");
    exit(6);
}
printf("receive one complete\n");
printf(buf1,"\n");
/* fprintf(stdout,buf1,"\n"); */
/*
 * Put end message into the buffer.
 */
 strcpy(buf2, "END CLIENT MESSAGE #2 ");
if (send(s, buf2, sizeof(buf2), 0) < 0)
{
    tcperror("Send()");
    exit(7);
}
printf("send two complete\n");
/*
 * The server sends back message #2. Receive it into buffer 2.
 */
if (recv(s, buf3, sizeof(buf3), 0) < 0)
{
    tcperror("Recv()");
    exit(8);
}
printf("receive two complete\n");
/* fprintf(stdout,buf3,"\n"); */
printf(buf3,"\n");
/*
 * Close the socket.
 */
close(s);
```

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

Sample Explicit-Mode Server Program (Assembler Language)

```
EZASVAS2  CSECT                       ENTRY POINT
          USING EZASVAS2,BASE         ADDRESSABILITY
          SAVE  (14,12)               SAVE DL/I REGS
          LR    BASE,15
          ST    R13,SAVEAREA+4        SAVE AREA CHAINING
          LA    R13,SAVEAREA          NEW SAVE AREA
          MVC   PSBS(L'PSBS*3),0(1)   SAVE PCB LIST
*
* REG 1 CONTAINS PTR TO PCB ADDR LIST
* REG 13 CONTAINS PTR TO DL/I SAVE AREA
* REG 14 CONTAINS PTR DL/I RETURN ADDRESS
* REG 15 CONTAINS PROGRAMS ENTRY POINT
*
          L     R2,0(R0,R1)           LOAD ADDR OF I/O PCB
*
          USING IOPCB,R2              ADDRESSABILITY
*
          L     R3,4(R0,R1)           LOAD ADDR OF ALT PCB
*
          USING ALTPCB1,R3            ADDRESSABILITY
*
          L     R4,8(R0,R1)           LOAD ADDR OF ALT PCB
          LA    R4,0(R0,R4)           REMOVE HIGH ORDER BIT
*
          USING ALTPCB2,R4            ADDRESSABILITY
*
          LA    R5,IOAREAIN
          LA    R7,IOAREAOT           POINT TO OUTPUT AREA FOR TCPIP
*
GUCALL    DS    0H                    GET UNIQUE CALL
*********************************************************************
*  Get Transaction-initiation message containing Sockets data     *
*********************************************************************
          CALL  ASMTDLI,(GUFUNCT,(2),(5)),VL     GET TIM
          CLC   STATUS(L'STATUS),=CL2'QC'    IF NO MESSAGES
          BE    GOBACK                       RETURN TO IMS
*                                            ELSE NEXT INSTR
          CLC   STATUS(L'STATUS),=CL2'  '    IF BLANK OK
          BNE   ERRRTN                       SOME WRONG HERE
*                                            ELSE NEXT INSTR
*
          XR  R6,R6                          CLEAR REG
          BAL R6,INITAPI                     GO INSERT SEGMENT
          B   GUCALL                         SET RETURN ADDRESS
*
*
INITAPI   DS   0H
* Set up for INITAPI
          MVC  TCPNAME(L'TCPNAME),TIMTCPAS     TCP Address space
          MVC  ASDNAME(L'ASDNAME),TIMSAS       Server address space
          MVC  SUBTASK(L'SUBTASK),TIMSTD       Server task id
* Set up for takeSOCKET
          MVC  NAME(L'NAME),TIMLAS             Listener address space
```

```
         MVC  TASK(L'TASK),TIMLTD            Listener task id
         MVC  S(L'S),TIMSD                   Socket descriptor
*
         XC   ERRNO(L'ERRNO),ERRNO
         XC   RETCODE(L'RETCODE),RETCODE
*        EX   0,*
*********************************************************************
*    Issue INITAPI                                                  *
*********************************************************************
         CALL EZASOKET,(INITFUNC,MAXSOC,APITYPE,IDENT,SUBTASK,        X
              MAXSNO,ERRNO,RETCODE),VL
         L    R9,RETCODE
         LTR  R9,R9
         BNM  TAKESOC
*
INITERR  DC   CL21'INITAPI COMMAND ERROR'
*
TAKESOC  DS   0H
*********************************************************************
*   Issue takeSOCKET                                                *
*********************************************************************
         CALL EZASOKET,(TAKEFUNC,S,CLIENT,ERRNO,RETCODE),VL
*
         L    R9,RETCODE
         LTR  R9,R9
         BNM  SENDTEXT
*
TAKERR   DC   CL16'TAKESOCKET ERROR'
*Set up to send "SERVER MSG #1"
SENDTEXT DS   0H
*
         MVC  S(L'S),RETCODE+2
         XC   BUF(LENG),BUF
         MVC  BUF(13),=CL13'SERVER MSG #1'
*Translate to ASCII, if necessary
*        CALL EZACIC04,(BUF,LENGTH),VL
*********************************************************************
*     Send "SERVER MSG #1"                                          *
*********************************************************************
         CALL EZASOKET,(SENDFUNC,S,FLAGS,NBYTE,BUF,ERRNO,RETCODE),    X
              VL
         L    R9,RETCODE
         LTR  R9,R9
         BNM  RECVTEXT
*
SENDERR1 DC   CL16'SEND ERROR'              Abend on error
RECVTEXT DS   0H
*********************************************************************
*     Receive client message #2                                      *
*********************************************************************
         CALL EZASOKET,(RECVFUNC,S,FLAGS,NBYTE,BUF,ERRNO,RETCODE),    X
              VL
* Translate to EBCDIC if necessary
*        CALL EZACIC05,(BUF,LENGTH),VL
*
         L    R9,RETCODE
         LTR  R9,R9
         BNM  CHECKTXT
*
         DC   CL16'RECEIVE ERROR'           Abend on error
*
```

```
CHECKTXT  DS   0H
*
          CLC  BUF(3),=CL3'END'              Test for end of message
          BNE  RECVTEXT                      If not eom, read again
*
*    Set up to send shutdown message
SENDEND   DS   0H
*
          XC   BUF(LENG),BUF
          MVC  BUF(13),=CL13'SERVER MSG #2'
*    Translate to ASCII if necessary
*         CALL EZACIC04,(BUF,LENGTH),VL
*********************************************************************
*    Send "SERVER MSG #2" to indicate shutdown                      *
*********************************************************************
          CALL EZASOKET,(SENDFUNC,S,FLAGS,NBYTE,BUF,ERRNO,RETCODE),    X
               VL
          L    R9,RETCODE
          LTR  R9,R9
          BNM  SOCKCLOS
*
SENDERR2  DC   CL16'SEND ERROR'              Abend on error
*
SOCKCLOS  DS   0H
*********************************************************************
*    Close the socket                                               *
*********************************************************************
          CALL EZASOKET,(CLOSFUNC,S,ERRNO,RETCODE),VL
*
          L    R9,RETCODE
          LTR  R9,R9
          BNM  TERMAPI
*
CLOSERR   DC   CL16'CLOSE ERROR'
*
TERMAPI   DS   0H
*********************************************************************
*    Terminate the API                                              *
*********************************************************************
          CALL EZASOKET,(TERMFUNC),VL
*
PROCTCP   DS    0H                       Talk to TCPIP Client
*                                             AND ALTERNATE
*                                             SUCESSFUL MSG
          XR    R9,R9                        CLEAR REG
          LA    R9,OTLEN                     LOAD LENGTH
          STH   R9,OTLTH                     STORE LEN THERE
          XC    OTRSV(L'OTRSV),OTRSV         CLEAR RESERVE DATA
          MVC   OTMSG(L'OTMSG),DCINMSG       MOVE IN MSG
          MVC   OTLITDT(L'OTLITDT),DCDATE    MOVE IN DATE
          MVC   OTLITIME(L'OTLITIME),DCTIME  MOVE IN TIME
          UNPK  OTDATE,CDATE                 MAKE TIME &amp. DATE
          OI    OTDATE+7,X'F0'                EBCDIC
          UNPK  OTTIME,CTIME
          OI    OTTIME+7,X'F0'
          XR    R9,R9                         GET READY
          L     R9,INPUTMSN                   INPUT COUNT
          CVD   R9,DLBWORK                    INPUT COUNT
          UNPK  OTINPUTN,DLBWORK              INPUT COUNT
          OI    OTINPUTN+7,X'F0'             FIX SIGN
          MVC   OTFILL(L'OTFILL),=28X'40'     FILL CHAR
```

```
          MVC   OTLTERM(L'OTLTERM),LTERMN         ADD TERMINAL
*
*
          CALL  ASMTDLI,(ISRTFUNCT,(3),(7),,USER1),VL
*
          XC   IOAREAOT(L'IOAREAOT),IOAREAOT
          BR    R6
*
ERRRTN    DS    0H                      SOME WRONG HERE
*
                CALL DFS0AER,((2),BADCALL,IOAREAIN,ERROPT),VL
*
GOBACK    DS    0H                      RETURN TO IMS
*
          L     R13,4(R13)
          RETURN (14,12),RC=0           RELOAD DL/I REGS &amp. RETURN
*
         DS    0D
PSBS     DS    3F
         SPACE 1
BASE     EQU   12
RC       EQU   15
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
         SPACE 1
*
          DS    0F
SAVEAREA  DC    18F'0'
*
GUFUNCT   DC    CL4'GU  '              GET UNIQUE CALL
GNFUNCT   DC    CL4'GN  '              GET NEXT
PURGFUNCT DC    CL4'PURG'              PURGE CALL
ISRTFUNCT DC    CL4'ISRT'              INSERT CALL
BADCALL   DC    CL8'BAD CALL'          BAD LIT
ERROPT    DC    F'0'                   1=nodump 0=dump
*
DCINMSG   DC    CL26' INPUT MESSAGE SUCESSFUL '
DCDATE    DC    CL6' DATE '
DCTIME    DC    CL6' TIME '
USER1     DC    CL8'USER1   '
USER2     DC    CL8'USER2   '
WTOR      DC    CL8'WTOR    '
*
INITFUNC  DC    CL16'INITAPI'
TAKEFUNC  DC    CL16'TAKESOCKET'
SENDFUNC  DC    CL16'SEND'
RECVFUNC  DC    CL16'RECV'
```

```
CLOSFUNC  DC    CL16'CLOSE'
TERMFUNC  DC    CL16'TERMAPI'
SELEFUNC  DC    CL16'SELECT'
*
WORKTCPIP DC    CL27'TCPIP WORK DATA BEGINS HERE'
APITYPE   DC    AL2(2)
MAXSOC    DC    AL2(MAX)
MAX       EQU   50
MAXSNO    DS    F'00'
*
IDENT     DS    0CL16
TCPNAME   DS    CL8
ASDNAME   DS    CL8
*
CLIENT    DS    0CL38
DOMAIN    DC    F'2'
NAME      DS    CL8
TASK      DS    CL8
RESERVED  DS    20B'0'
*
SUBTASK   DS    CL8
ERRNO     DS    F
RETCODE   DS    F
FLAGS     DC    F'0'
NBYTE     DC    F'50'
BUF       DS    CL(LENG)
LENG      EQU   50
LENGTH    DC    AL4(LENG)
TIMEOUT   DS    0D
SECONDS   DS    F
MILLISEC  DS    F
RSNDMASK   DS    CL(MAX)
WSNDMASK   DS    CL(MAX)
ESNDMASK   DS    CL(MAX)
RRETMASK   DS    CL(MAX)
WRETMASK   DS    CL(MAX)
ERETMASK   DS    CL(MAX)
S          DS    H
*
          DS    0D
DLBWORK   DS    D
          DS    0F
IOAREAIN  DS    0CL56                 I/O AREA INPUT
TIMLEN    DS    H                     Length of trans init msg
TIMRSV    DS    H                     reserved set to zeros
TIMID     DS    CL8                   LISTENER ID set to LISTNR
TIMLAS    DS    CL8                   LISTENER addr space name
TIMLTD    DS    CL8                   LISTENER taskid for takesocket
TIMSAS    DS    CL8                   SERVER addr space name
TIMSTD    DS    CL8                   SERVER TASK ID user in initapi
TIMSD     DS    H                     socket given in LISTENER used in
*                                                     tasksocket
TIMTCPAS  DS    CL8                   TCPIP addr space name
TIMDT     DS    H                     Data type of client
*                                     ASCII(0) or EBCDIC(1)
          DS    0F
IOAREAOT  DS    0CL119                I/O AREA OUTPUT
OTLTH     DS    BL2
OTRSV     DS    BL2
OTLTERM   DS    CL8
OTINPUTN  DS    CL8
```

```
OTMSG     DS    CL25
OTLITDT   DS    CL6
OTDATE    DS    CL8
OTLITIME  DS    CL6
OTTIME    DS    CL8
OTFILL    DS    CL28
OTLEN     EQU   (*-IOAREAOT)
*
IOPCB     DSECT                     I/O AREA
LTERMN    DS    CL8                 LOGICAL TERMINAL NAME
          DS    CL2                 RESERVED FOR IMS
STATUS    DS    CL2                 STATUS CODE
CDATE     DS    PL4                 CURRENT DATE YYDDD
CTIME     DS    PL4                 CURRENT TIME HHMMSST
INPUTMSN  DS    BL4                 SEQUENCE NUMBER
MSGOUTDN  DS    CL8                 MESSAGE OUT DESC NAME
USERID    DS    CL8                 USER ID OF SOURCE
*
ALTPCB1   DSECT                     ALTERNATE PCB
ALTERM1   DS    CL8                 DESTINATION NAME
          DS    CL2                 RESERVED FOR IMS
ALSTAT1   DS    CL2                 STATUS CODE
*
ALTPCB2   DSECT                     ALTERNATE PCB
ALTERM2   DS    CL8                 DESTINATION NAME
          DS    CL2                 RESERVED FOR IMS
ALSTAT2   DS    CL2                 STATUS CODE
*
          END
```

Sample Program Implicit-Mode

The following is an example of an implicit-mode client server program pair. The client program name is EZAIMSC1; you can find it in *hlq*.SEZAINST(EZAIMSC1). The server program name is EZASVAS1; its IMS trancode is DLSI101. The sample program is located in *hlq*.SEZAINST(EZASVAS1).

Program flow

The client begins execution and obtains the host name and port number from the startup parameters. It then issues SOCKET and CONNECT calls to establish connectivity to the specified host and port. Upon successful completion of the CONNECT, the client sends the TRM, which tells the Listener to schedule the specified transaction (DLSI101). Because implicit-mode protocol requires that all input data segments be transmitted before the server application is scheduled, the client follows the TRM with 2 segments of application data and an end-of-message (EOM) segment. The Listener schedules DLSI101 and places a TIM on the IMS message queue, followed by the 2 segments of application data. Finally, the Listener issues a GIVESOCKET call and waits for the server to take the socket.

When the requested server (EZASVAS1) begins execution, it issues a GU call to ASMADLI. Behind the scenes, the Assist module issues its own GU and retrieves the TIM from the IMS message queue. Using addressability information from the TIM, it issues INITAPI and takeSOCKET calls, which establish connectivity with the client.

Once connectivity is established, the Assist module issues a GN to the IMS message queue, which returns the first segment of application data sent by the client. This data is returned to the server mainline. (Thus, to the server mainline, the

first segment of application data is returned in response to its GU.) In the sample program, the first segment of application data is the data record: `THIS IS FIRST TEXT MESSAGE SEND TO SERVER`. This record is echoed back to the client by means of an IMS ISRT call to ASMADLI. The IMS Assist module intercepts the ISRT and issues a TCP/IP write() to echo the segment back to the client. The server mainline then issues a GN ASMADLI (which the Assist module intercepts and executes another GN ASMTDLI) to recieve the second segment of user data. This segment is also echoed back to the client, using an IMS ISRT call, which the Assist module intercepts and replaces with a TCP/IP write() to the client.

After the second client data segment, the message queue contains an EOM segment, denoting the client's end-of-message. When the server has echoed the second input segment to the client, it issues another GN to ASMADLI. ASMADLI receives an end-of-message indication from the message queue and passes a QD status code back to the server mainline.

At this point, the server mainline has completed processing that message and issues a GU to see whether another message has arrived for that trancode. This GU triggers the Assist module to send a final CSMOKY message to the client, indicating successful completion. It then issues another GU to the IMS message queue to determine whether another message for that trancode has been queued. If so, the server program repeats itself; if not, the server issues a GOBACK and ends.

Sample Implicit-Mode Client Program (C Language)

```
.*
.* Different than part at level OLDPROD OLDVER/OLDLVL.
/*
 * Include Files.
 */
/*  #define RESOLVE_VIA_LOOKUP */
#pragma runopts(NOSPIE NOSTAE)
#define lim 119
#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>
#include <socket.h>
#include <netdb.h>
#include <stdio.h>
/*
 * Client Main.
 */
main(argc, argv)
int argc;
char **argv;
{
    unsigned short port;       /* port client will connect to         */
    struct sktmsg
           {
              short msglen;
              short msgrsv;
              char  msgtrn??(8??);
              char  msgdat??(lim??);
           }  msgbuff;
    struct datmsg
           {
              short datlen;
```

```
          short datrsv;
          char  datdat??(lim??);
       }  datbuff;
char buf ??(lim??);        /* send receive buffer                */
struct hostent *hostnm;    /* server host name information       */
struct sockaddr_in server; /* server address                     */
int s;                     /* client socket                      */
int len;                   /* length for send                    */
/*
 * Check Arguments Passed. Should be hostname and port.
 */
if (argc != 3)
{
    printf("Invalid parameter count\n");
    exit(1);
}
printf("Usage: %s program name\n",argv??(0??));
/*
 * The host name is the first argument. Get the server address.
 */
printf("Usage: %s host name\n",argv??(1??));
hostnm = gethostbyname(argvŸ1¨);
if (hostnm == (struct hostent *) 0)
{
     printf("Gethostbyname failed\n");
    exit(2);
}
/*
 * The port is the second argument.
 */
printf("Usage: %s port name\n",argv??(2??));
port = (unsigned short) atoi(argvŸ2¨);
/*
 * Put the server information into the server structure.
 * The port must be put into network byte order.
 */
server.sin_family      = AF_INET;
server.sin_port        = htons(port);
server.sin_addr.s_addr = *((unsigned long *)hostnm->h_addr);
/*
 * Get a stream socket.
 */
if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
    tcperror("Socket()");
    exit(3);
}
/*
 * Connect to the server.
 */
if (connect(s, (struct sockaddr *)&amp.server, sizeof(server)) < 0)
{
    tcperror("Connect()");
    exit(4);
}
/*
 * Put a message into the buffer.
 */
msgbuff.msgdat??(0??)='\0';
msgbuff.msgrsv = 0;
msgbuff.msglen = 20;
```

```
strncat(msgbuff.msgtrn,"*TRNREQ*",
        lim-strlen(msgbuff.msgdat)-1);
strncat(msgbuff.msgdat,"DLSI101 ",
        lim-strlen(msgbuff.msgdat)-1);
len=20;
 if (send(s, (char *)&amp.msgbuff, len, 0) < 0)
 {
     tcperror("Send()");
     exit(5);
 }
 printf("\n");
 printf(msgbuff.msgdat);
 printf("send one complete\n");
 /*
  * Put a text message into the buffer.
  */
datbuff.datdat??(0??)='\0';
datbuff.datlen = 46;
datbuff.datrsv = 0;
strncat(datbuff.datdat,"THIS IS FIRST TEXT MESSAGE SEND TO SERVER ",
        lim-strlen(datbuff.datdat)-1);
len=46;
 if (send(s, (char *)&amp.datbuff, len, 0) < 0)
 {
     tcperror("Send()");
     exit(6);
 }
 printf("\n");
 printf(datbuff.datdat);
 printf("\n");
 printf("send for first text message complete\n");
 /*
  * Put a text message into the buffer.
  */
datbuff.datdat??(0??)='\0';
datbuff.datlen = 47;
strncat(datbuff.datdat,"THIS IS 2ND TEXT MESSAGE SENDING TO SERVER",
        lim-strlen(datbuff.datdat)-1);
len=47;
 if (send(s, (char *)&amp.datbuff, len, 0) < 0)
 {
     tcperror("Send()");
     exit(7);
 }
 printf("\n");
 printf(datbuff.datdat);
 printf("\n");
 printf("send for 2nd test message complete\n");
 /*
  * Put a end message into the buffer.
  */
datbuff.datdat??(0??)='\0';
datbuff.datlen = 4;
strncpy(datbuff.datdat," ",lim);
 len=4;
 if (send(s, (char *)&amp.datbuff, len, 0) < 0)
 {
     tcperror("Send()");
     exit(8);
 }
 printf("\n");
```

```
    printf(datbuff.datdat);
    printf("\n");
    printf("send for end message complete\n");
    /*
     * The server sends back the same message. Receive it into the
     * buffer.
     */
    strncpy(datbuff.datdat," ",lim);
    if (recv(s,(char *)&amp.datbuff, lim, 0) < 0)
    {
        tcperror("Recv()");
        exit(9);
    }
    printf("receive one text complete\n");
    printf(datbuff.datdat);
    printf("\n");
    /*
     * The server sends back the same message. Receive it into the
     * buffer.
     */
    strncpy(datbuff.datdat," ",lim);
    if (recv(s,(char *)&amp.datbuff, lim, 0) < 0)
    {
        tcperror("Recv()");
        exit(10);
    }
    printf("receive two text complete\n");
    printf(datbuff.datdat);
    printf("\n");
    /*
     * The server sends eof message. Receive it into the
     * buffer.
     */
    strncpy(datbuff.datdat," ",lim);
    if (recv(s,(char *)&amp.datbuff, 4, 0) < 0)
    {
        tcperror("Recv()");
        exit(11);
    }
    printf("receive eof complete\n");
    printf("\n");
    printf(datbuff.datdat);
    printf("\n");
    strncpy(datbuff.datdat," ",lim);
    if (recv(s,(char *)&amp.datbuff, 12, 0) < 0)
    {
        tcperror("Recv()");
        exit(12);
    }
    printf("receive CSMOKY complete\n");
    printf("\n");
    printf(datbuff.datdat);
    printf("\n");
    /*
     * Close the socket.
     */
    close(s);
    printf("Client Ended Successfully\n");
    exit(0);
}
```

Sample Implicit-Mode Server Program (Assembler Language)

```
EZASVAS1  CSECT                         ENTRY POINT
          USING EZASVAS1,BASE           ADDRESSABILITY
          SAVE  (14,12)                 SAVE DL/I REGS
          LR    BASE,15
          ST    R13,SAVEAREA+4          SAVE AREA CHAINING
          LA    R13,SAVEAREA            NEW SAVE AREA
          MVC   PSBS(L'PSBS*3),0(1)     SAVE PCB LIST
*
* REG 1 CONTAINS PTR TO PCB ADDR LIST
* REG 13 CONTAINS PTR TO DL/I SAVE AREA
* REG 14 CONTAINS PTR DL/I RETURN ADDRESS
* REG 15 CONTAINS PROGRAMS ENTRY POINT
*
          L     R2,0(R0,R1)             LOAD ADDR OF I/O PCB
*
          USING IOPCB,R2                ADDRESSABILITY
*
          L     R3,4(R0,R1)             LOAD ADDR OF ALT PCB
*
          USING ALTPCB1,R3              ADDRESSABILITY
*
          L     R4,8(R0,R1)             LOAD ADDR OF ALT PCB
          LA    R4,0(R0,R4)             REMOVE HIGH ORDER BIT
*
          USING ALTPCB2,R4              ADDRESSABILITY
*
          LA    R5,IOAREAIN
          LA    R7,IOAREAOT                       POINT TO OUTPUT AREA
*
GUCALL    DS    0H                      GET UNIQUE CALL
*
*
          CALL  ASMADLI,(GUFUNCT,(2),(5)),VL
*
          CLC   STATUS(L'STATUS),=CL2'QC'    IF NO MESSAGES
          BE    GOBACK                       RETURN TO IMS
*                                            ELSE NEXT INSTR
          CLC   STATUS(L'STATUS),=CL2'  '    IF BLANK OK
          BNE   ERRRTN                       SOME WRONG HERE
*                                            ELSE NEXT INSTR
*
          XR  R6,R6                          CLEAR REG
          LA  R6,GNCALL                      SET RETURN ADDRESS
          BAL R6,ISRTCALL                    GO INSERT SEGMENT
*
GNCALL    DS    0H                      GET NEXT CALL
*
*
          CALL  ASMADLI,(GNFUNCT,(2),(5)),VL
*
          CLC   STATUS(L'STATUS),=CL2'QD'    IF NO MORE SEGMENTS
          BE    GUCALL                       RETURN TO IMS
          CLC   STATUS(L'STATUS),=CL2'  '    IF NO MORE SEGMENTS
          BNE   ERRRTN                       SOME WRONG HERE
*
          XR  R6,R6                          CLEAR REG
          LA  R6,GNLOOP                      SET RETURN ADDRESS
```

```
          BAL R6,ISRTCALL                   GO INSERT SEGMENT
*
GNLOOP    B     GNCALL
*
ISRTCALL  DS    0H                      INSERT - WRITE TO TERMINAL
*                                               AND ALTERNATE
*                                               SUCESSFUL MSG
          XR    R9,R9                           CLEAR REG
          LA    R9,OTLEN                        LOAD LENGTH
          STH   R9,OTLTH                        STORE LEN THERE
          XC    OTRSV(L'OTRSV),OTRSV            CLEAR RESERVE DATA
          MVC   OTMSG(L'OTMSG),DCINMSG          MOVE IN MSG
          MVC   OTLITDT(L'OTLITDT),DCDATE        "    " DATE
          MVC   OTLITIME(L'OTLITIME),DCTIME      "    " TIME
          UNPK  OTDATE,CDATE                     MAKE TIME &amp. DATE
          OI    OTDATE+7,X'F0'                   EBCDIC
          UNPK  OTTIME,CTIME
          OI    OTTIME+7,X'F0'
          XR    R9,R9                            GET READY
          L     R9,INPUTMSN                      INPUT COUNT
          CVD   R9,DLBWORK                       INPUT COUNT
          UNPK  OTINPUTN,DLBWORK                 INPUT COUNT
          OI    OTINPUTN+7,X'F0'                FIX SIGN
          MVC   OTFILL(L'OTFILL),=28X'40'        FILL CHAR
          MVC   OTLTERM(L'OTLTERM),LTERMN        ADD TERMINAL
*
* For LTERM USER1....
*
          CALL  ASMADLI,(ISRTFUNCT,(2),(7)),VL
*
* For LTERM USER2....
*
          XC   IOAREAOT(L'IOAREAOT),IOAREAOT
          BR   R6
*
ERRRTN    DS    0H                      SOME WRONG HERE
*
          CALL  DFS0AER,((2),BADCALL,IOAREAIN,ERROPT),VL
*
GOBACK    DS    0H                      RETURN TO IMS
*
          L     R13,4(R13)
          RETURN (14,12),RC=0           RELOAD DL/I REGS &amp. RETURN
*
         DS    0D
PSBS     DS    3F
         SPACE 1
BASE     EQU   12
RC       EQU   15
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
         SPACE 1
*
          DS    0F
SAVEAREA  DC    18F'0'
GUFUNCT   DC    CL4'GU  '              GET UNIQUE CALL
GNFUNCT   DC    CL4'GN  '              GET NEXT
PURGFUNCT DC    CL4'PURG'              PURGE CALL
ISRTFUNCT DC    CL4'ISRT'              INSERT CALL
BADCALL   DC    CL8'BAD CALL'          BAD LIT
ERROPT    DC    F'1'                   1=NODUMP 2=DUMP
DCINMSG   DC    CL26' INPUT MESSAGE SUCESSFUL '
DCDATE    DC    CL6' DATE '
DCTIME    DC    CL6' TIME '
USER1     DC    CL8'USER1   '
USER2     DC    CL8'USER2   '
WTOR      DC    CL8'WTOR    '
*
          DS    0D
DLBWORK   DS    D
          DS    0F
IOAREAIN  DS    CL119                  I/O AREA INPUT
          DS    0F
IOAREAOT  DS    0CL119                 I/O AREA OUTPUT
OTLTH     DS    BL2
OTRSV     DS    BL2
OTLTERM   DS    CL8
OTINPUTN  DS    CL8
OTMSG     DS    CL25
OTLITDT   DS    CL6
OTDATE    DS    CL8
OTLITIME  DS    CL6
OTTIME    DS    CL8
OTFILL    DS    CL46
OTLEN     EQU   (*-IOAREAOT)
*
IOPCB     DSECT                        I/O AREA
LTERMN    DS    CL8                    LOGICAL TERMINAL NAME
          DS    CL2                    RESERVED FOR IMS
STATUS    DS    CL2                    STATUS CODE
CDATE     DS    PL4                    CURRENT DATE YYDDD
CTIME     DS    PL4                    CURRENT TIME HHMMSST
INPUTMSN  DS    BL4                    SEQUENCE NUMBER
MSGOUTDN  DS    CL8                    MESSAGE OUT DESC NAME
USERID    DS    CL8                    USER ID OF SOURCE
*
ALTPCB1   DSECT                        ALTERNATE PCB
ALTERM1   DS    CL8                    DESTINATION NAME
          DS    CL2                    RESERVED FOR IMS
ALSTAT1   DS    CL2                    STATUS CODE
*
ALTPCB2   DSECT                        ALTERNATE PCB
ALTERM2   DS    CL8                    DESTINATION NAME
          DS    CL2                    RESERVED FOR IMS
ALSTAT2   DS    CL2                    STATUS CODE
*
*
          END
```

Sample Program - IMS MPP Client

Most of the discussion in this book assumes that the IMS system is the server; however, some applications require that the server be a TCP/IP host. The following is an example of a program in which the *client* is an IMS MPP, and the *server* is a TCP/IP host.

For simplicity, we have coded both client and server to execute on an MVS host. The client (EZAIMSC3) is initated by a 3270-driven IMS MPP; the server (EZASVAS3) is a TSO job which is already running when the client starts.

The samples are located in *hlq*.SEZAINST(EZAIMSC3) and *hlq*.SEZAINST(EZASVAS3).

Program Flow

A TSO Submit command is used to start the server. Once started, it executes the TCP/IP connection sequence for an iterative server (INITAPI, SOCKET, BIND, LISTEN, SELECT, and ACCEPT) and then waits for the client to request connection.

Note that the BIND call returns a socket descriptor which is then used to listen for a connection request. The ACCEPT call also returns a socket descriptor, which is used for the application data connection. Meanwhile, the original listener socket is available to receive additional connection requests.

The client is started by calling an IMS transaction which, in turn, executes the TCP/IP connection sequence for a client (INITAPI, SOCKET, and CONNECT).

Upon receiving the connection request from the client, the server issues a READ and waits for the client to WRITE the initial message. The server contains a READ/WRITE loop which echoes client transmissions until an ″END″ message is received. When this message is received, it sets a 'last record' switch, echoes the end message to the client, and terminates.

Note that in order for the server to terminate, it must close two sockets: one -- the socket on which it listens for connection requests; the other -- the socket on which the data transfers took place.

The client and server both include Write To Operator macros, which allow you to monitor progress through the application logic flow. At the end of this appendix you will find a sample of the WTO output from the client and the server.

Sample Client Program for non-IMS server

```
EZAIMSC3 CSECT
EZAIMSC3 AMODE ANY
EZAIMSC3 RMODE ANY
         GBLB  &TRACE  ASSEMBLER VARIABLE TO CONTROL TRACE GENERATION
&TRACE   SETB  1       1=TRACE ON  0=TRACE OFF
         GBLB  &SUBTR  ASSEMBLER VARIABLE TO CONTROL SUBTRACE
&SUBTR   SETB  0       1=SUBTRACE ON  0=SUBTRACE OFF
*-------------------------------------------------------------------*
*                                                                   *
*   MODULE NAME:  EZAIMSC3                                          *
*                                                                   *
```

```
*   MODULE FUNCTION: Sample program of an IMS MPP TCP client. This    *
*                    module connects with a TCP/IP server and         *
*                    exchanges msgs with it. The number of msgs       *
*                    exchanged is determined by a constant and        *
*                    the length of the messages is also determined    *
*                    by a constant.                                   *
*                    Note: If an error occurs during processing, this *
*                    module will send an error message to the system  *
*                    console and then Abends0c1.                      *
*                                                                     *
*   LANGUAGE:  Assembler                                              *
*                                                                     *
*   ATTRIBUTES: Reusable                                              *
*                                                                     *
*   INPUT: None                                                       *
*                                                                     *
*---------------------------------------------------------------------*
SOC0000  DS    0H
         USING *,R15              Tell assembler to use reg 15
         B     SOC00100           Branch to startup address
         DC    CL16'IMSTCPCLEYECATCH'
BUFLEN   EQU   1000               Set length of I/O buffers
R4BASE   DC    A(SOC0000+4096)
*---------------------------------------------------------------------*
*        Control Variables for this program                           *
*---------------------------------------------------------------------*
SOCMSGN  DC    F'005'             Number of messages to be exchanged
SOCMSGL  DC    F'200'             Length of messages to be exchanged
SERVPORT DC    H'5000'            Port Address of Server
SOCTASK  DC    F'0'               Task number for this client
SERVLEN  DC    H'0'               Length of server's name
SERVNAME DC    CL24' '            Internet name of server
SENDINT  DC    CL8'00000010'      Delay interval between sends
*---------------------------------------------------------------------*
*        Constants used for call functions                            *
*---------------------------------------------------------------------*
INITAPI  DC    CL16'INITAPI'
GETHSTID DC    CL16'GETHOSTID'
SOCKET   DC    CL16'SOCKET'
GHBN     DC    CL16'GETHOSTBYNAME'
CONNECT  DC    CL16'CONNECT'
READ     DC    CL16'READ'
WRITE    DC    CL16'WRITE'
CLOSE    DC    CL16'CLOSE'
TERMAPI  DC    CL16'TERMAPI'
*---------------------------------------------------------------------*
*        Beginning of program execution statements                    *
*---------------------------------------------------------------------*
SOC00100 DS    0H                 Beginning of program
         STM   R14,R12,12(R13)    Save callers registers
         LR    R3,R15             Move base reg to R3
         L     R4,R4BASE          Add R4 as second base reg
         DROP  R15                Tell assembler to drop R15 as base
         USING SOC0000,R3,R4      Tell assembler to use R3 and R4 as   X
                                  base registers
         LR    R7,R13             Save address of previous save area
         LA    R12,SOCSTG         Move address of program stg to R12
         LA    R13,SOCSTGL        Move length of program stge to R13
         SR    R14,R14            Clear R14
         SR    R15,R15            Clear R15
         MVCL  R12,R14            Clear program storage
```

```
         LA    R13,SOCSTG         Move address of program stg to R13
         USING SOCSTG,R13         Tell Assembler about storage
         ST    R7,SOCSAVEL        Save address of lower save area
         ST    R13,8(R7)          Complete save area chain
SOC00200 DS    0H
*
*   Build message for console
*
         MVC   MSG1D,MSG1C        Initialize first part of message
         L     R0,SOCTASK         Get task number
         CVD   R0,DWORK           Convert task number to decimal
         UNPK  MSGTD,DWORK+5(3)   Convert decimal to character
         OI    MSGTD+4,X'F0'      Clear sign
         MVC   MSG2D,MSG2CS       Move 'Started' to message
         LA    R6,MSG             Put text address in R6
         MVC   MSGLEN,=AL2(MSGTL) Put length of text in msg hdr.
         MVC   WTOLIST,WTOPROT    Move prototype WTO to list form
         WTO   TEXT=(R6),         Write message to operator           X
               MF=(E,WTOLIST)
*
*        Issue INITAPI Call to connect to interface
*
         MVC   SOCTASKC(3),=CL3'SOC'  Build Task Identifier
         MVC   SOCTASKC+3(5),MSGTD
         MVC   MSG2D,MSG2C1       Move 'INITAPI'to message
         MVC   MAXSOC,=H'50'      Initialize MAXSOC field
         MVC   ASTCPNAM,=CL8'TCPV3   '  Initialize TCP Name
         MVC   ASCLNAME,=CL8'TCPCLINT'  Initialize AS Name
*
         CALL  EZASOKET,                                          X
               (INITAPI,MAXSOC,ASIDENT,SOCTASKC,HISOC,ERRNO,      X
               RETCODE),                                          X
               VL                 Specify variable parameter list
*
         L     R6,RETCODE         Check for sucessful call
         C     R6,=F'0'           Is it less than zero
         BL    SOCERR             Yes, go display error and terminat
         AIF   (NOT &TRACE).TRACE01
*  TRACE ENTRY FOR INITAPI  TRACE TYPE = 1
         LA    R6,MSG             Put text address in R6
         MVC   MSGLEN,=AL2(MSGTL) Put length of text in msg hdr.
         WTO   TEXT=(R6),         Write message to operator           X
               MF=(E,WTOLIST)
.TRACE01 ANOP
*
*        Issue GETHOSTID Call  to obtain internet address of host
*
         MVC   MSG2D,MSG2C8       Move 'GTHSTID'to message
*
         CALL  EZASOKET,          Issue GETHOSTID Call               X
               (GETHSTID,SERVIADD),                               X
               VL                 Specify Variable parameter list
*
         AIF   (NOT &TRACE).TRACE08
*  TRACE ENTRY FOR GETHOSTID  TRACE TYPE = 8
         LA    R6,MSG             Put text address in R6
         MVC   MSGLEN,=AL2(MSGTL) Put length of text in msg hdr.
         WTO   TEXT=(R6),         Write message to operator           X
               MF=(E,WTOLIST)
.TRACE08 ANOP
*
```

```
*        Issue SOCKET Call to obtain a socket descriptor
*
         MVC   MSG2D,MSG2C2       Move 'SOCKET' to message
         MVC   AF,=F'2'           Address Family = Internet
         MVC   SOCTYPE,=F'1'      Type = Stream Sockets
         XC    PROTO,PROTO        Clear protocol field
*
         CALL  EZASOKET,          Issue SOCKET Call                   X
               (SOCKET,AF,SOCTYPE,PROTO,ERRNO,RETCODE),               X
               VL                 Specify variable parameter list
*
         L     R6,RETCODE         Check for sucessful call
         C     R6,=F'0'           Is it less than zero
         BL    SOCERR             Yes, go display error and terminat
         AIF   (NOT &TRACE).TRACE02
*  TRACE ENTRY FOR SOCKET  TRACE TYPE = 2
         LA    R6,MSG             Put text address in R6
         MVC   MSGLEN,=AL2(MSGTL) Put length of text in msg hdr.
         WTO   TEXT=(R6),         Write message to operator           X
               MF=(E,WTOLIST)
.TRACE02 ANOP
*
*        Get socket descriptor number
*
         L     R6,RETCODE         Descriptor number returned
         STH   R6,SOCDESC         Save it
*
*        Issue CONNECT Command to Connect to Server
*
         MVC   SSOCAF,=H'2'       Set AF=INET
         MVC   SSOCPORT,SERVPORT  Move Port Number
         MVC   SSOCINET,SERVIADD  Move Internet Address of Server
         MVC   MSG2D,MSG2C4       Move 'CONNECT' to message
*
         CALL  EZASOKET,          Issue CONNECT Call                  X
               (CONNECT,SOCDESC,SERVSOC,ERRNO,RETCODE),               X
               VL                 Specify variable parameter list
*
         L     R6,RETCODE         Check for sucessful call
         C     R6,=F'0'           Is it less than zero
         BL    SOCERR             Yes, go display error and terminat
         AIF   (NOT &TRACE).TRACE04
*  TRACE ENTRY FOR CONNECT  TRACE TYPE = 4
         LA    R6,MSG             Put text address in R6
         MVC   MSGLEN,=AL2(MSGTL) Put length of text in msg hdr.
         WTO   TEXT=(R6),         Write message to operator           X
               MF=(E,WTOLIST)
.TRACE04 ANOP
*
*        Send initial message to server
*
         MVC   BUFFER(L'MSG1),MSG1  Move Message to Buffer
         LA    R6,L'MSG1          Get length of message
         ST    R6,DATALEN         Put length in data field
         MVC   MSG2D,MSG2C5       Move 'WRITE' to message
*
         CALL  EZASOKET,          Issue WRITE Call                    X
               (WRITE,SOCDESC,DATALEN,BUFFER,ERRNO,RETCODE),          X
               VL
*
         L     R6,RETCODE         Check for sucessful call
```

```
         C     R6,=F'0'           Is it less than zero
         BL    SOCERR             Yes, go display error and terminat
         AIF   (NOT &TRACE).TRACE05
*  TRACE ENTRY FOR WRITE  TRACE TYPE = 5
         MVC   MSGLEN,=AL2(MSGTL+18) Put length of text in msg hdr.
         MVC   MSG3D,ERR3C        ' RETCODE= '
         MVI   MSG3S,C'+'         Move sign
         L     R6,RETCODE         Get return code value
         CVD   R6,DWORK           Convert it to decimal
         UNPK  MSG4D,DWORK+4(4)   Unpack it
         OI    MSG4D+6,X'F0'      Correct the sign
         LA    R6,MSG             Put text address in R6
         WTO   TEXT=(R6),         Write message to operator           X
               MF=(E,WTOLIST)
.TRACE05 ANOP
*
*        Read response to initial message
*
         MVC   MSG2D,MSG2C6       Move 'READ' to message
         LA    R6,L'BUFFER        Get length of buffer
         ST    R6,DATALEN         Put length in data field
*
         CALL  EZASOKET,          Issue READ Call                     X
               (READ,SOCDESC,DATALEN,BUFFER,ERRNO,RETCODE),          X
               VL                 Specify variable parameter list
*
         L     R6,RETCODE         Check for sucessful call
         C     R6,=F'0'           Is it less than zero
         BL    SOCERR             Yes, go display error and terminat
         AIF   (NOT &.TRACE).TRACE06
*  TRACE ENTRY FOR READ  TRACE TYPE = 6
         MVC   MSGLEN,=AL2(MSGTL+18) Put length of text in msg hdr.
         MVC   MSG3D,ERR3C        ' RETCODE= '
         MVI   MSG3S,C'+'         Move sign
         L     R6,RETCODE         Get return code value
         CVD   R6,DWORK           Convert it to decimal
         UNPK  MSG4D,DWORK+4(4)   Unpack it
         OI    MSG4D+6,X'F0'      Correct the sign
         LA    R6,MSG             Put text address in R6
         WTO   TEXT=(R6),         Write message to operator           X
               MF=(E,WTOLIST)
.TRACE06 ANOP
*
*        Send second message to server
*
         MVC   BUFFER(L'MSG2),MSG2  Move Message to Buffer
         LA    R6,L'MSG2          Get length of message
         ST    R6,DATALEN         Put length in data field
         MVC   MSG2D,MSG2C5       Move 'WRITE' to message
*
         CALL  EZASOKET,          Issue WRITE Call                    X
               (WRITE,SOCDESC,DATALEN,BUFFER,ERRNO,RETCODE),         X
               VL
*
         L     R6,RETCODE         Check for sucessful call
         C     R6,=F'0'           Is it less than zero
         BL    SOCERR             Yes, go display error and terminat
         AIF   (NOT &TRACE).TRACE15
*  TRACE ENTRY FOR WRITE  TRACE TYPE = 5
         MVC   MSGLEN,=AL2(MSGTL+18) Put length of text in msg hdr.
         MVC   MSG3D,ERR3C        ' RETCODE= '
```

```
         MVI   MSG3S,C'+'         Move sign
         L     R6,RETCODE         Get return code value
         CVD   R6,DWORK           Convert it to decimal
         UNPK  MSG4D,DWORK+4(4)   Unpack it
         OI    MSG4D+6,X'F0'      Correct the sign
         LA    R6,MSG             Put text address in R6
         WTO   TEXT=(R6),         Write message to operator           X
               MF=(E,WTOLIST)
.TRACE15 ANOP
         L     R6,RETCODE         Check for sucessful call
         C     R6,=F'0'           Is it less than zero
         BL    SOCERR             Yes, go display error and terminat
*
*        Read response to second message
*
         MVC   MSG2D,MSG2C6       Move 'READ' to message
*
         CALL  EZASOKET,          Issue READ Call                     X
               (READ,SOCDESC,SOCMSGL,BUFFER,ERRNO,RETCODE),          X
               VL                 Specify variable parameter list
*
         L     R6,RETCODE         Check for sucessful call
         C     R6,=F'0'           Is it less than zero
         BL    SOCERR             Yes, go display error and terminat
*
         AIF   (NOT &TRACE).TRACE16
*  TRACE ENTRY FOR READ  TRACE TYPE = 6
         MVC   MSGLEN,=AL2(MSGTL+18) Put length of text in msg hdr.
         MVC   MSG3D,ERR3C        ' RETCODE= '
         MVI   MSG3S,C'+'         Move sign
         L     R6,RETCODE         Get return code value
         CVD   R6,DWORK           Convert it to decimal
         UNPK  MSG4D,DWORK+4(4)   Unpack it
         OI    MSG4D+6,X'F0'      Correct the sign
         LA    R6,MSG             Put text address in R6
         WTO   TEXT=(R6),         Write message to operator           X
               MF=(E,WTOLIST)
.TRACE16 ANOP
*
*        Send End message to server
*
         MVC   BUFFER(L'ENDMSG),ENDMSG  Move end message to buffer
         LA    R6,L'ENDMSG        Get length of message
         ST    R6,SOCMSGL         Put length in length field
         MVC   MSG2D,MSG2C5       Move 'WRITE' to message
*
         CALL  EZASOKET,          Issue WRITE Call                    X
               (WRITE,SOCDESC,SOCMSGL,BUFFER,ERRNO,RETCODE),         X
               VL
*
         L     R6,RETCODE         Check for sucessful call
         C     R6,=F'0'           Is it less than zero
         BL    SOCERR             Yes, go display error and terminat
         AIF   (NOT &TRACE).TRACE25
*  TRACE ENTRY FOR WRITE  TRACE TYPE = 5
         MVC   MSGLEN,=AL2(MSGTL+18) Put length of text in msg hdr.
         MVC   MSG3D,ERR3C        ' RETCODE= '
         MVI   MSG3S,C'+'         Move sign
         L     R6,RETCODE         Get return code value
         CVD   R6,DWORK           Convert it to decimal
         UNPK  MSG4D,DWORK+4(4)   Unpack it
```

```
         OI    MSG4D+6,X'F0'      Correct the sign
         LA    R6,MSG             Put text address in R6
         WTO   TEXT=(R6),         Write message to operator           X
               MF=(E,WTOLIST)
.TRACE25 ANOP
*
*        Read response to end message
*
         MVC   MSG2D,MSG2C6       Move 'READ' to message
*
         CALL  EZASOKET,          Issue READ Call                     X
               (READ,SOCDESC,SOCMSGL,BUFFER,ERRNO,RETCODE),          X
               VL                 Specify variable parameter list
*
         L     R6,RETCODE         Check for sucessful call
         C     R6,=F'0'           Is it less than zero
         BL    SOCERR             Yes, go display error and terminat
         AIF   (NOT &TRACE).TRACE26
*  TRACE ENTRY FOR READ  TRACE TYPE = 6
         MVC   MSGLEN,=AL2(MSGTL+18) Put length of text in msg hdr.
         MVC   MSG3D,ERR3C        ' RETCODE= '
         MVI   MSG3S,C'+'         Move sign
         L     R6,RETCODE         Get return code value
         CVD   R6,DWORK           Convert it to decimal
         UNPK  MSG4D,DWORK+4(4)   Unpack it
         OI    MSG4D+6,X'F0'      Correct the sign
         LA    R6,MSG             Put text address in R6
         WTO   TEXT=(R6),         Write message to operator           X
               MF=(E,WTOLIST)
.TRACE26 ANOP
*
*        Close socket
*
         MVC   MSG2D,MSG2C7       Move 'CLOSE' to message
*
         CALL  EZASOKET,          Issue CLOSE Call                    X
               (CLOSE,SOCDESC,ERRNO,RETCODE),                        X
               VL                 Specify variable parameter list
*
         L     R6,RETCODE         Check for sucessful call
         C     R6,=F'0'           Is it less than zero
         BL    SOCERR             Yes, go display error and terminat
         AIF   (NOT &TRACE).TRACE07
*  TRACE ENTRY FOR CLOSE TRACE TYPE = 7
         LA    R6,MSG             Put text address in R6
         MVC   MSGLEN,=AL2(MSGTL) Put length of text in msg hdr.
         WTO   TEXT=(R6),         Write message to operator           X
               MF=(E,WTOLIST)
.TRACE07 ANOP
*
*        Terminate Connection to API
*
         CALL  EZASOKET,          Issue TERMAPI Call                  X
               (TERMAPI),                                            X
               VL                 Specify variable parameter list
*
*        Issue console message for task termination
*
         MVC   MSG2D,MSG2CE       Move 'Ended' to message
         LA    R6,MSG             Put text address in R6
         MVC   MSGLEN,=AL2(MSGTL) Put length of text in msg hdr.
```

```
         WTO   TEXT=(R6),         Write message to operator           X
               MF=(E,WTOLIST)
*
*        Return to Caller
*
         L     R13,SOCSAVEL
         LM    R14,R12,12(R13)
         BR    R14
*
*        Write error message to operator and ABENDS0C1
*
SOCERR   DS    0H                 Write error message to operator
         MVC   ERR1D,MSG1D        'IMSTCPCL, TASK #'
         MVC   ERRTD,MSGTD        Move task number to message
         MVC   ERR2D,MSG2D        Call Type
         MVC   ERR3D,ERR3C        ' RETCODE= '
         MVI   ERR3S,C'-'         Move sign which is always minus
         MVC   ERR5D,ERR5C        ' ERRNO= '
         L     R6,RETCODE         Get return code value
         CVD   R6,DWORK           Convert it to decimal
         UNPK  ERR4D,DWORK+4(4)   Unpack it
         OI    ERR4D+6,X'F0'      Correct the sign
         L     R6,ERRNO           Get errno value
         CVD   R6,DWORK           Convert it to decimal
         UNPK  ERR6D,DWORK+4(4)   Unpack it
         OI    ERR6D+6,X'F0'      Correct the sign
         LA    R6,ERR             Put text address in R6
         MVC   ERRLEN,=AL2(ERRTL) Put length of text in msg hdr.
         WTO   TEXT=(R6),         Write message to operator           X
               MF=(E,WTOLIST)
ABEND    DS    0H
         DC    H'0'               Force ABEND
WTOPROT  WTO   TEXT=,             List form of WTO Macro              X
               MF=L
WTOPROTL EQU   *-WTOPROT          Length of WTO Prototype
MSG1C    DC    CL17'IMSTCPCL, TASK # '
MSG2CS   DC    CL8' STARTED'
MSG2CE   DC    CL8' ENDED  '
ERR3C    DC    CL10' RETCODE= '
ERR5C    DC    CL8' ERRNO= '
MSG2C1   DC    CL8' INITAPI'
MSG2C2   DC    CL8' SOCKET '
MSG2C4   DC    CL8' CONNECT'
MSG2C5   DC    CL8' WRITE  '
MSG2C6   DC    CL8' READ   '
MSG2C7   DC    CL8' CLOSE  '
MSG2C8   DC    CL8' GTHSTID'
MSG2C35  DC    CL8' SYNC   '
MSG1     DC    CL16'CLIENT MESSAGE 1'     First msg to server
MSG2     DC    CL16'CLIENT MESSAGE 2'     2nd msg to server
ENDMSG   DS    0CL48              End Message for Server
         DC    CL3'END'           End indicator for SRV1
         DC    CL45' '            Pad with blanks
         DS    0D
SOCSTG   DS    0F                 PROGRAM STORAGE
SOCSAVE  DS    0F                 Save Area
SOCSAVE1 DS    F                  Word for high-level languages
SOCSAVEL DS    F                  Address of previous save area
SOCSAVEH DS    F                  Address of next save area
SOCSAV14 DS    F                  Reg 14
SOCSAV15 DS    F                  Reg 15
```

```
SOCSAV0  DS    F                 Reg 0
SOCSAV1  DS    F                 Reg 1
SOCSAV2  DS    F                 Reg 2
SOCSAV3  DS    F                 Reg 3
SOCSAV4  DS    F                 Reg 4
SOCSAV5  DS    F                 Reg 5
SOCSAV6  DS    F                 Reg 6
SOCSAV7  DS    F                 Reg 7
SOCSAV8  DS    F                 Reg 8
SOCSAV9  DS    F                 Reg 9
SOCSAV10 DS    F                 Reg 10
SOCSAV11 DS    F                 Reg 11
SOCSAV12 DS    F                 Reg 12
SOCSAV13 DS    F                 Reg 13
MAXSOC   DS    H                 Maximum number of sockets for this   X
                                        application
SOCTASKC DS    CL8               Character task identifier
SOCDESC  DS    H                 Socket Descriptor Number
HISOC    DS    F                 Highest socket descriptor available
AF       DS    F                 Address family for socket call
SOCTYPE  DS    F                 Type of socket
NS       DS    F                 New socket number for socket call
SERVAL   DS    12F               Alias array for server
SERVSOC  DS    0F                Socket Address of Server
SSOCAF   DS    H                 Address Family of Server = 2
SSOCPORT DS    H                 Port number for Server
SSOCINET DS    F                 Internet address for Server
         DC    D'0'              Reserved
MSG      DS    0F                Message area
MSGLEN   DS    H                 Length of message
MSG1D    DS    CL17              'IMSTCPCL, TASK #'
MSGTD    DS    CL5               Task Number
MSG2D    DS    CL8               Last part of message
MSGE     EQU   *                 End of message
MSGTL    EQU   MSGE-MSG1D        Length of message text
MSG3D    DS    CL10              ' RETCODE = '
MSG3S    DS    C                 Sign which is always -
MSG4D    DS    CL7               Return code
ERR      DS    0F                Error message area
ERRLEN   DS    H                 Length of message
ERR1D    DS    CL17              'IMSTCPCL, TASK #'
ERRTD    DS    CL5               Task Number
ERR2D    DS    CL8               Last part of message
ERR3D    DS    CL10              ' RETCODE = '
ERR3S    DS    C                 Sign which is always -
ERR4D    DS    CL7               Return code
ERR5D    DS    CL8               ' ERRNO ='
ERR6D    DS    CL7               Error number
ERRE     EQU   *                 End of message
ERRTL    EQU   ERRE-ERR1D        Length of message text
BUFFER   DS    CL(BUFLEN)        Socket I/O Buffer
DATALEN  DS    F                 Length of buffer data
DWORK    DS    D                 Double word work area
RECNO    DS    PL4               Record Number
ERRNO    DS    F                 Error number returned from call
RETCODE  DS    F                 Return code from call
PROTO    DS    F                 Protocol field for socket
ASIDENT  DS    0F                Address space identifier for initapi
ASTCPNAM DS    CL8               Name of TCP/IP Address Space
SERVIADD DS    F                 Internet address for Server
ASCLNAME DS    CL8               Our name as known to TCP/IP
```

```
WTOLIST  DS    CL(WTOPROTL)       List form of WTO Macro
SOCSTGE  EQU   *                  End of Program Storage
SOCSTGL  EQU   SOCSTGE-SOCSTG     Length of Program Storage
         LTORG
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
GWABAR   EQU   13
         END
```

Sample Server Program for IMS MPP Client

```
EZASVAS3 CSECT
EZASVAS3 AMODE ANY
EZASVAS3 RMODE ANY
         GBLB  &TRACE  ASSEMBLER VARIABLE TO CONTROL TRACE GENERATION
&TRACE   SETB  1       1=TRACE ON  0=TRACE OFF
         GBLB  &SUBTR  ASSEMBLER VARIABLE TO CONTROL SUBTRACE
&SUBTR   SETB  0       1=SUBTRACE ON  0=SUBTRACE OFF
*-------------------------------------------------------------------*
*                                                                   *
*   MODULE NAME:  EZASVAS3                                          *
*                                                                   *
*   MODULE FUNCTION: Test module for Extended Sockets.  This module *
*                    accepts connection request from IMS client     *
*                    program named EZAIMSC3.                        *
*                                                                   *
*   LANGUAGE:  Assembler                                            *
*                                                                   *
*   ATTRIBUTES: Non-reusable                                        *
*                                                                   *
*                                                                   *
*                                                                   *
*-------------------------------------------------------------------*
SOC0000  DS    0H
         USING *,R15              Tell assembler to use reg 15
         B     SOC00100           Branch to startup address
         DC    CL14'SERVEREYECATCH'
ASIDENT  DS    0F                 Address Space Identifier for initapi
ASTCPNAM DC    CL8'TCPV3    '     Name of TCP/IP Address Space
ASCLNAME DC    CL8'CALLSRVER'     Our name as known to TCP/IP
TIMEOUT  DS    0F                 Timeout value for select
TIMESEC  DC    F'180'             Timeout value in seconds
TIMEMSEC DC    F'0'               Timeout value in milliseconds
BUFLEN   EQU   1000               Set length of I/O buffers
R4BASE   DC    A(SOC0000+4096)
```

```
SOC00100 DS    0H                Beginning of program
         STM   R14,R12,12(R13)   Save callers registers
         LR    R3,R15            Move base reg to R3
         L     R4,R4BASE         Add R4 as second base reg
         DROP  R15               Tell assembler to drop R15 as base
         USING SOC0000,R3,R4     Tell assembler to use R3 and R4 as   X
                                 base registers
         LA    R6,SOCSTG         Clear program storage
         LA    R7,SOCSTGL
         SR    R14,R14
         SR    R15,R15
         MVCL  R6,R14
         ST    R13,SOCSAVEH      Save address of higher save area
         LA    R7,SOCSAVE        Complete save area chain
         ST    R7,8(R13)         Tell caller where our save area is
         LA    R13,SOCSAVE       Point R13 at our save area
         MVI   ENDSW,X'00'       Clear end-of-transmission switch
*
*   Build message for console
*
         MVC   MSG1D,MSG1C       Initialize first part of message
         MVC   MSGTD,=CL5'00000' Move subtask number from clientid
         MVC   MSG2D,MSG2CS      Move 'Started' to message
         LA    R6,MSG            Put text address in R6
         MVC   MSGLEN,=AL2(MSGTL) Put length of text in msg hdr.
         MVC   WTOLIST,WTOPROT   Move prototype WTO to list form
         WTO   TEXT=(R6),        Write message to operator            X
               MF=(E,WTOLIST)
*
*        Issue INITAPI Call to connect to interface
*
         MVC   SOCTASKC,=CL8'TAS00000' Give subtask a name
         MVC   MSG2D,MSG2C00     Move 'INITAPI'to message
         MVC   MAXSOC,=H'50'     Initialize MAXSOC parameter
*
         CALL  EZASOKET,                                            X
               (INITAPI,MAXSOC,ASIDENT,SOCTASKC,HISOC,ERRNO,        X
               RETCODE),                                            X
               VL
*
         L     R6,RETCODE        Check for sucessful call
         C     R6,=F'0'          Is it less than zero
         BL    SOCERR            Yes, go display error and terminat
         AIF   (NOT &TRACE).TRACE00
*  TRACE ENTRY FOR INITAPI  TRACE TYPE = 0
         LA    R6,MSG            Put text address in R6
         MVC   MSGLEN,=AL2(MSGTL) Put length of text in msg hdr.
         WTO   TEXT=(R6),        Write message to operator            X
               MF=(E,WTOLIST)
.TRACE00 ANOP
*
*        Issue SOCKET Call to obtain socket to listen on
*
         MVC   MSG2D,MSG2C25     Move 'SOCKET'to message
         MVC   AF,=F'2'          Initialize AF to '2' (INET)
         MVC   SOCTYPE,=F'1'     Specify stream sockets
         MVC   PROTO,=F'0'       Protocol is ignored for stream
*
         CALL  EZASOKET,         Issue SOCKET CALL                  X
               (SOCKET,AF,SOCTYPE,PROTO,ERRNO,RETCODE),             X
               VL
```

```
*
         L     R6,RETCODE         Check for sucessful call
         C     R6,=F'0'           Is it less than zero
         BL    SOCERR             Yes, go display error and terminate
         AIF   (NOT &TRACE).TRACE25
*  TRACE ENTRY FOR SOCKET TRACE TYPE = 25
         LA    R6,MSG             Put text address in R6
         MVC   MSGLEN,=AL2(MSGTL) Put length of text in msg hdr.
         WTO   TEXT=(R6),         Write message to operator            X
               MF=(E,WTOLIST)
.TRACE25 ANOP
         L     R0,RETCODE         Get descriptor number of socket
         STH   R0,LISTSOC         Save it
*
*        Issue GETHOSTID call to determine our internet address
*
         MVC   MSG2D,MSG2C07      Move 'GETHSTID'to message
*
         CALL  EZASOKET,          Issue GETHOSTID Call                 X
               (GETHSTID,RETCODE),VL
*
         AIF   (NOT &TRACE).TRACE07
*  TRACE ENTRY FOR SOCKET TRACE TYPE = 07
         LA    R6,MSG             Put text address in R6
         MVC   MSGLEN,=AL2(MSGTL) Put length of text in msg hdr.
         WTO   TEXT=(R6),         Write message to operator            X
               MF=(E,WTOLIST)
.TRACE07 ANOP
         L     R0,RETCODE         Get internet address of host
         ST    R0,SINETADR        Save it
*
*        Issue BIND call to establish port
*
         MVC   MSG2D,MSG2C02      Move 'BIND' to message
         MVC   SPORT,=H'5000'     Move port number to structure
         MVC   SAF,=H'2'          Move AF (INET) to structure
*
         CALL  EZASOKET,          Issue BIND Call                      X
               (BIND,LISTSOC,SOCKNAME,ERRNO,RETCODE),                 X
               VL
         L     R6,RETCODE         Check for sucessful call
         C     R6,=F'0'           Is it less than zero
         BL    SOCERR             Yes, go display error and terminat
*
         AIF   (NOT &TRACE).TRACE02
*  TRACE ENTRY FOR BIND TRACE TYPE = 02
         LA    R6,MSG             Put text address in R6
         MVC   MSGLEN,=AL2(MSGTL) Put length of text in msg hdr.
         WTO   TEXT=(R6),         Write message to operator            X
               MF=(E,WTOLIST)
.TRACE02 ANOP
*
*
*        Issue LISTEN call to establish backlog of connection requests
*
         MVC   MSG2D,MSG2C13      Move 'LISTEN' to message
         MVC   BACKLOG,=F'5'      Set backlog to 5
*
         CALL  EZASOKET,          Issue LISTEN Call                    X
               (LISTEN,LISTSOC,BACKLOG,ERRNO,RETCODE),VL
         L     R6,RETCODE         Check for sucessful call
```

```
         C     R6,=F'0'           Is it less than zero
         BL    SOCERR             Yes, go display error and terminate
*
         AIF   (NOT &TRACE).TRACE13
*  TRACE ENTRY FOR LISTEN TRACE TYPE = 13
         LA    R6,MSG             Put text address in R6
         MVC   MSGLEN,=AL2(MSGTL) Put length of text in msg hdr.
         WTO   TEXT=(R6),         Write message to operator            X
               MF=(E,WTOLIST)
.TRACE13 ANOP
*
*        Issue SELECT call to wait on connection request
*
         MVC   MSG2D,MSG2C19      Move 'SELECT' to message
         MVC   SELSOC,=F'31'      Maximum number of sockets
         MVC   WSNDMASK,=F'0'     Not checking for writes
         MVC   ESNDMASK,=F'0'     Not checking for exceptions
         LA    R0,1               Put 1 in rightmost position of R0
         LH    R1,LISTSOC         Put listener socket number in R1
         SLL   R0,0(R1)           Create mask for read
         ST    R0,RSNDMASK        Put value in mask field
*
         CALL  EZASOKET,          Issue SELECT Call                    X
               (SELECT,SELSOC,TIMEOUT,RSNDMASK,WSNDMASK,ESNDMASK,     X
               RRETMASK,WRETMASK,ERETMASK,ERRNO,RETCODE),             X
               VL
         L     R6,RETCODE         Check for sucessful call
         C     R6,=F'0'           Is it less than zero
         BL    SOCERR             Yes, go display error and terminat
*
         AIF   (NOT &TRACE).TRACE19
*  TRACE ENTRY FOR SELECT TRACE TYPE = 19
         LA    R6,MSG             Put text address in R6
         MVC   MSGLEN,=AL2(MSGTL) Put length of text in msg hdr.
         WTO   TEXT=(R6),         Write message to operator            X
               MF=(E,WTOLIST)
.TRACE19 ANOP
*
*        Issue ACCEPT call to accept a new connection
*
         MVC   MSG2D,MSG2C01      Move 'ACCEPT' to message
         MVC   NS,=F'4'           Use socket 4 for connection socket
*
         CALL  EZASOKET,          Issue ACCEPT Call                    X
               (ACCEPT,LISTSOC,SOCKNAME,ERRNO,RETCODE),               X
               VL
         L     R6,RETCODE         Check for sucessful call
         C     R6,=F'0'           Is it less than zero
         BL    SOCERR             Yes, go display error and terminat
*
         AIF   (NOT &TRACE).TRACE01
*  TRACE ENTRY FOR ACCEPT TRACE TYPE = 01
         LA    R6,MSG             Put text address in R6
         MVC   MSGLEN,=AL2(MSGTL) Put length of text in msg hdr.
         WTO   TEXT=(R6),         Write message to operator            X
               MF=(E,WTOLIST)
.TRACE01 ANOP
         L    R0,RETCODE          Get descriptor number of new socket
         STH  R0,CONNSOC          Save it for future use
*
*        Issue READ call to get first message from client
```

```
*
         LA    R6,L'BUFFER        Get length of buffer
         ST    R6,DATALEN         Put length in data field
         MVC   MSG2D,MSG2C14      Move 'READ' to message
         XC    FLAGS,FLAGS        Clear the FLAGS field
*
         CALL  EZASOKET,          Issue READ Call                     X
               (READ,CONNSOC,DATALEN,BUFFER,ERRNO,RETCODE),VL
         L     R6,RETCODE         Check for sucessful call
         C     R6,=F'0'           Is it less than zero
         BL    SOCERR             Yes, go display error and terminat
*
         AIF   (NOT &TRACE).TRAC14A
*  TRACE ENTRY FOR READ TRACE TYPE = 14
         LA    R6,MSG             Put text address in R6
         MVC   MSGLEN,=AL2(MSGTL) Put length of text in msg hdr.
         WTO   TEXT=(R6),         Write message to operator           X
               MF=(E,WTOLIST)
.TRAC14A ANOP
*
*        Send Initial Message to client to continue transaction
*
         MVC   BUFFER(L'RESPMSG),RESPMSG  Move Message to Buffer
         LA    R6,L'RESPMSG       Get length of message
         ST    R6,DATALEN         Put length in data field
         XC    FLAGS,FLAGS        Clear FLAGS field
         MVC   MSG2D,MSG2C26      Move 'WRITE' to message
*
         CALL  EZASOKET,          Issue WRITE call                    X
               (WRITE,CONNSOC,DATALEN,BUFFER,ERRNO,RETCODE),VL
*
         L     R6,RETCODE         Check for sucessful call
         C     R6,=F'0'           Is it less than zero
         BL    SOCERR             Yes, go display error and terminat
         AIF   (NOT &TRACE).TRAC26A
*  TRACE ENTRY FOR WRITE TRACE TYPE = 22
         LA    R6,MSG             Put text address in R6
         MVC   MSGLEN,=AL2(MSGTL) Put length of text in msg hdr.
         WTO   TEXT=(R6),         Write message to operator           X
               MF=(E,WTOLIST)
.TRAC26A ANOP
SOC0300  DS    0H
*
*        Read Message from Client
*
         MVC   MSG2D,MSG2C14      Move 'READ' to message
         LA    R0,L'BUFFER        Get length of buffer
         ST    R0,DATALEN         Use it for data length
         XC    FLAGS,FLAGS        Clear FLAGS field
*
         CALL  EZASOKET,                                            X
               (READ,CONNSOC,DATALEN,BUFFER,ERRNO,RETCODE),VL
*
         L     R6,RETCODE         Check for sucessful call
         C     R6,=F'0'           Is it less than zero
         BNH   SOCERR             Yes, go display error and terminat
         AIF   (NOT &TRACE).TRAC14B
*  TRACE ENTRY FOR RECV  TRACE TYPE = 14
         LA    R6,MSG             Put text address in R6
         MVC   MSGLEN,=AL2(MSGTL) Put length of text in msg hdr.
         WTO   TEXT=(R6),         Write message to operator           X
```

```
              MF=(E,WTOLIST)
.TRAC14B ANOP
         CLC   BUFFER(3),=CL3'END' Was this last record
         BNE   SOC0350            No
         MVI   ENDSW,C'E'         Yes, set end-of-transmission switch
SOC0350  DS    0H
*
*        Send Response to Client
*
         MVC   MSG2D,MSG2C26      Move 'WRITE' to message
         MVC   DATALEN,RETCODE    Get message length from previous call
         XC    FLAGS,FLAGS        Clear FLAGS field
*
         CALL  EZASOKET,                                            X
               (WRITE,CONNSOC,DATALEN,BUFFER,ERRNO,RETCODE),VL
*
         L     R6,RETCODE         Check for sucessful call
         C     R6,=F'0'           Is it less than zero
         BNH   SOCERR             Yes, go display error and terminat
         AIF   (NOT &TRACE).TRAC26B
*  TRACE ENTRY FOR SEND   TRACE TYPE = 26
         LA    R6,MSG             Put text address in R6
         MVC   MSGLEN,=AL2(MSGTL) Put length of text in msg hdr.
         WTO   TEXT=(R6),         Write message to operator          X
               MF=(E,WTOLIST)
.TRAC26B ANOP
*
         CLI   ENDSW,C'E'         Have we received last record
         BNE   SOC0300            No, so go back and do another
*
*        Close sockets
*
         MVC   MSG2D,MSG2C03      Move 'CLOSE1' to message
*
         CALL  EZASOKET,          Issue CLOSE call for connection skt  X
               (CLOSE,CONNSOC,ERRNO,RETCODE),VL
*
         L     R6,RETCODE         Check for sucessful call
         C     R6,=F'0'           Is it less than zero
         BL    SOCERR             Yes, go display error and terminat
         AIF   (NOT &TRACE).TRACE03
*  TRACE ENTRY FOR CLOSE TRACE TYPE = 3
         LA    R6,MSG             Put text address in R6
         MVC   MSGLEN,=AL2(MSGTL) Put length of text in msg hdr.
         WTO   TEXT=(R6),         Write message to operator          X
               MF=(E,WTOLIST)
.TRACE03 ANOP
*
         MVC   MSG2D,MSG2C03A     Move 'CLOSE2' to message
*
         CALL  EZASOKET,          Issue CLOSE call for listen socket   X
               (CLOSE,LISTSOC,ERRNO,RETCODE),VL
*
         L     R6,RETCODE         Check for sucessful call
         C     R6,=F'0'           Is it less than zero
         BL    SOCERR             Yes, go display error and terminat
         AIF   (NOT &TRACE).TRAC103
*  TRACE ENTRY FOR CLOSE TRACE TYPE = 3
         LA    R6,MSG             Put text address in R6
         MVC   MSGLEN,=AL2(MSGTL) Put length of text in msg hdr.
         WTO   TEXT=(R6),         Write message to operator          X
```

```
              MF=(E,WTOLIST)
.TRAC103 ANOP
*
*        Terminate Connection to API
*
         CALL  EZASOKET,                                          X
               (TERMAPI),VL
*
*        Issue console message for task termination
*
         MVC   MSG2D,MSG2CE       Move 'Ended' to message
         LA    R6,MSG             Put text address in R6
         MVC   MSGLEN,=AL2(MSGTL) Put length of text in msg hdr.
         WTO   TEXT=(R6),         Write message to operator       X
               MF=(E,WTOLIST)
*
*        Return to Caller
*
         L     R13,SOCSAVEH
         LM    R14,R12,12(R13)
         BR    R14
*
*        Write error message to operator
*
SOCERR   DS    0H                 Write error message to operator
         MVC   ERR1D,MSG1D        'SERVER, TASK #'
         MVC   ERRTD,MSGTD        Move task number to message
         MVC   ERR2D,MSG2D        Call Type
         MVC   ERR3D,ERR3C        ' RETCODE= '
         MVI   ERR3S,C'-'         Move sign which is always minus
         MVC   ERR5D,ERR5C        ' ERRNO= '
         L     R6,RETCODE         Get return code value
         CVD   R6,DWORK           Convert it to decimal
         UNPK  ERR4D,DWORK+4(4)   Unpack it
         OI    ERR4D+6,X'F0'      Correct the sign
         L     R6,ERRNO           Get errno value
         CVD   R6,DWORK           Convert it to decimal
         UNPK  ERR6D,DWORK+4(4)   Unpack it
         OI    ERR6D+6,X'F0'      Correct the sign
         LA    R6,ERR             Put text address in R6
         MVC   ERRLEN,=AL2(ERRTL) Put length of text in msg hdr.
         WTO   TEXT=(R6),         Write message to operator       X
               MF=(E,WTOLIST)
*
*        Return to Caller
*
*        L     R13,SOCSAVEH
*        LM    R14,R12,12(R13)
*        BR    R14
ABEND    DS    0H
         DC    H'0'               Force ABEND
*-------------------------------------------------------------------*
*        Constants                                                  *
*-------------------------------------------------------------------*
WTOPROT  WTO   TEXT=,              List form of WTO Macro          X
               MF=L
WTOPROTL EQU   *-WTOPROT          Length of WTO Prototype
MSG1C    DC    CL17'SERVER,   TASK # '
MSG2CS   DC    CL8' STARTED'
MSG2CE   DC    CL8' ENDED  '
ERR3C    DC    CL10' RETCODE= '
```

```
ERR5C    DC    CL8' ERRNO= '
MSG2C00  DC    CL8' INITAPI'
MSG2C01  DC    CL8' ACCEPT '
MSG2C02  DC    CL8' BIND   '
MSG2C03  DC    CL8' CLOSE  '
MSG2C03A DC    CL8' CLOSE2 '
MSG2C07  DC    CL8' GTHSTID'
MSG2C13  DC    CL8' LISTEN '
MSG2C14  DC    CL8' READ   '
MSG2C19  DC    CL8' SELECT '
MSG2C25  DC    CL8' SOCKET '
MSG2C26  DC    CL8' WRITE  '
MSG2C32  DC    CL8' TAKESKT'
RESPMSG  DC    CL50'FIRST RESPONSE FROM SERVER  '
*-------------------------------------------------------------------*
*        Constants used for call types                              *
*-------------------------------------------------------------------*
INITAPI  DC    CL16'INITAPI'
BIND     DC    CL16'BIND'
LISTEN   DC    CL16'LISTEN'
ACCEPT   DC    CL16'ACCEPT'
READ     DC    CL16'READ'
SELECT   DC    CL16'SELECT'
WRITE    DC    CL16'WRITE'
SOCKET   DC    CL16'SOCKET'
CLOSE    DC    CL16'CLOSE'
GETHSTID DC    CL16'GETHOSTID'
TERMAPI  DC    CL16'TERMAPI'
*-------------------------------------------------------------------*
*        Program Storage Area                                       *
*-------------------------------------------------------------------*
SOCSTG   DS    0F               PROGRAM STORAGE
SOCSAVE  DS    0F               Save Area
SOCSAVE1 DS    F                Word for high-level languages
SOCSAVEH DS    F                Address of previous save area
SOCSAVEL DS    F                Address of next save area
SOCSAV14 DS    F                Reg 14
SOCSAV15 DS    F                Reg 15
SOCSAV0  DS    F                Reg 0
SOCSAV1  DS    F                Reg 1
SOCSAV2  DS    F                Reg 2
SOCSAV3  DS    F                Reg 3
SOCSAV4  DS    F                Reg 4
SOCSAV5  DS    F                Reg 5
SOCSAV6  DS    F                Reg 6
SOCSAV7  DS    F                Reg 7
SOCSAV8  DS    F                Reg 8
SOCSAV9  DS    F                Reg 9
SOCSAV10 DS    F                Reg 10
SOCSAV11 DS    F                Reg 11
SOCSAV12 DS    F                Reg 12
SOCSAV13 DS    F                Reg 13
PARMADDR DS    F                Address of parameter list
GWAADDR  DS    F                Address of Global Work Area
TIEADDR  DS    F                Address of Task Information Element
LISTSOC  DS    H                Socket number used for listen
CONNSOC  DS    H                Socket number created by accept
SOCMSGN  DS    F                Number of messages to be exchanged
SOCMSGL  DS    F                Length of messages to be exchanged
SOCTASKC DS    CL8              Character task identifier
HISOC    DS    F                Highest socket descriptor available
```

```
SERVLEN  DS    H
SERVSOC  DS    0F                Socket Address of Server
SERVAF   DS    H                 Address Family of Server = 2
SERVPORT DS    H                 Port Address of Server
SERVIADD DS    F                 Internet Address of Server
ENDSW    DS    C                 End of transmission switch
MSG      DS    0F                Message area
MSGLEN   DS    H                 Length of message
MSG1D    DS    CL17              'SERVER,   TASK #'
MSGTD    DS    CL5               Task Number
MSG2D    DS    CL8               Last part of message
MSGE     EQU   *                 End of message
MSGTL    EQU   MSGE-MSG1D        Length of message text
ERR      DS    0F                Error message area
ERRLEN   DS    H                 Length of message
ERR1D    DS    CL17              'SERVER,   TASK #'
ERRTD    DS    CL5               Task Number
ERR2D    DS    CL8               Last part of message
ERR3D    DS    CL10              ' RETCODE = '
ERR3S    DS    C                 Sign which is always -
ERR4D    DS    CL7               Return code
ERR5D    DS    CL8               ' ERRNO ='
ERR6D    DS    CL7               Error number
ERRE     EQU   *                 End of message
ERRTL    EQU   ERRE-ERR1D        Length of message text
*-------------------------------------------------------------------*
*     Name structure used by bind                                   *
*-------------------------------------------------------------------*
SOCKNAME DS    0F                Socket Name structure
SAF      DS    H                 The address family of the socket
SPORT    DS    H                 The port number of this socket
SINETADR DS    F                 The internet address of this socket
         DS    D                 Reserved
SOCKNAML EQU   *-SOCKNAME        Length of SOCKNAME Structure
CLIENTID DS    0F                Client Id structure
CDOMAIN  DS    F                 The domain of this client (2)
CNAME    DS    CL8               The major name of this client
CSUBTASK DS    CL8               The minor (subtask) name of this     X
                                 client
         DS    D                 Reserved
CLIENTL  EQU   *-CLIENTID
BUFFER   DS    CL(BUFLEN)        Socket I/O Buffer
DATALEN  DS    F                 Length of buffer data
DWORK    DS    D                 Double word work area
SENDINT  DS    D                 Time interval for send
RECNO    DS    PL4               Record Number
AF       DS    F                 Address family for socket call
NS       DS    F                 New socket number for socket call
SOCTYPE  DS    F                 Socket type for socket call
PROTO    DS    F                 Protocol for socket call
ERRNO    DS    F                 Error number returned from call
RETCODE  DS    F                 Return code from call
CINADDR  DS    F                 Internet address of client
CPORT    DS    F                 Port number of client
MAXSOC   DS    H                 Maximum # sockets for INITAPI
SELSOC   DS    F                 Maximum # sockets for SELECT
BACKLOG  DS    F                 Backlog value for LISTEN
FLAGS    DS    F                 FLAGS field for RECV and RECVFROM
RSNDMASK DS    F                 Read send mask for select
WSNDMASK DS    F                 Write send mask for select
ESNDMASK DS    F                 Exception send mask for select
```

```
RRETMASK DS    F                 Read return mask for select
WRETMASK DS    F                 Write return mask for select
ERETMASK DS    F                 Exception return mask for select
WTOLIST  DS    CL(WTOPROTL)      List form of WTO Macro
EZASMTI  EZASMI TYPE=TASK,                                          X
               STORAGE=CSECT     Generate task storage for interface
EZASMGW  EZASMI TYPE=GLOBAL,     Storage definition for GWA         X
               STORAGE=CSECT
SOCSTGE  EQU   *                 End of Program Storage
SOCSTGL  EQU   SOCSTGE-SOCSTG    Length of Program Storage
         LTORG
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
GWABAR   EQU   13
         END
```

WTO output from sample program

Client Output

```
13.29.18 JOB00084  IEF403I SOCCALLS - STARTED - TIME=13.29.18
13.29.18 JOB00084  +SERVER,   TASK # 00000 STARTED
13.29.19 JOB00084  +SERVER,   TASK # 00000 INITAPI
13.29.19 JOB00084  +SERVER,   TASK # 00000 SOCKET
13.29.19 JOB00084  +SERVER,   TASK # 00000 GTHSTID
13.29.19 JOB00084  +SERVER,   TASK # 00000 BIND
13.29.20 JOB00084  +SERVER,   TASK # 00000 LISTEN
13.29.41 JOB00084  +SERVER,   TASK # 00000 SELECT
13.29.41 JOB00084  +SERVER,   TASK # 00000 ACCEPT
13.29.41 JOB00084  +SERVER,   TASK # 00000 READ
13.29.41 JOB00084  +SERVER,   TASK # 00000 WRITE
13.29.41 JOB00084  +SERVER,   TASK # 00000 READ
13.29.41 JOB00084  +SERVER,   TASK # 00000 WRITE
13.29.41 JOB00084  +SERVER,   TASK # 00000 READ
13.29.42 JOB00084  +SERVER,   TASK # 00000 WRITE
13.29.42 JOB00084  +SERVER,   TASK # 00000 CLOSE
13.29.42 JOB00084  +SERVER,   TASK # 00000 CLOSE2
13.29.42 JOB00084  +SERVER,   TASK # 00000 ENDED
```

Server Output

```
13.27.45 JOB00082  IEF403I MESSAGE - STARTED - TIME=13.27.45
13.29.40 JOB00082  +IMSTCPCL, TASK # 00000 STARTED
13.29.41 JOB00082  +IMSTCPCL, TASK # 00000 INITAPI
13.29.41 JOB00082  +IMSTCPCL, TASK # 00000 GTHSTID
13.29.41 JOB00082  +IMSTCPCL, TASK # 00000 SOCKET
13.29.41 JOB00082  +IMSTCPCL, TASK # 00000 CONNECT
13.29.41 JOB00082  +IMSTCPCL, TASK # 00000 WRITE   RETCODE= +0000016
13.29.41 JOB00082  +IMSTCPCL, TASK # 00000 READ    RETCODE= +0000050
13.29.41 JOB00082  +IMSTCPCL, TASK # 00000 WRITE   RETCODE= +0000016
13.29.41 JOB00082  +IMSTCPCL, TASK # 00000 READ    RETCODE= +0000016
13.29.41 JOB00082  +IMSTCPCL, TASK # 00000 WRITE   RETCODE= +0000048
```

```
13.29.42 JOB00082  +IMSTCPCL, TASK # 00000 READ    RETCODE= +0000048
13.29.42 JOB00082  +IMSTCPCL, TASK # 00000 CLOSE
13.29.42 JOB00082  +IMSTCPCL, TASK # 00000 ENDED
```

Part 3. Appendixes

Appendix A. Return Codes

This appendix covers the following return codes and error messages

- Error numbers from MVS TCP/IP
- Error codes from the Sockets Extended interface

Sockets Extended ERRNOs

This section contains the error condition codes that are returned in the ERRNO field by the API when using the macro or SOCKET interface.

Note: The return codes 10119 through 10130 return the IPUSER variable.

Table 4. Sockets Extended ERRNOs

Error Code	Problem Description	System Action	Programmer's Response
10100	An ESTAE macro did not complete normally.	End the call.	Call your MVS system programmer.
10101	A STORAGE OBTAIN failed.	End the call.	Increase MVS storage in the application's address space.
10108	The first call from TCP/IP was not INITAPI or TAKESOCKET.	End the call.	Change the first TCP/IP call to INITAPI or TAKESOCKET.
10110	LOAD of EZBSOH03 (alias EZASOH03) failed.	End the call.	Call the IBM Software Support Center.
10154	Errors were found in the parameter list for an IOCTL call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the IOCTL call. You might have incorrect sequencing of socket calls.
10155	The length parameter for an IOCTL call is less than or equal to zero.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the IOCTL call. You might have incorrect sequencing of socket calls.
10156	The length parameter for an IOCTL call is 3200 (32 x 100).	Disable the subtask for interrupts. Return an error code to the caller.	Correct the IOCTL call. You might have incorrect sequencing of socket calls.
10159	A zero or negative data length was specified for a READ or READV call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the length in the READ call.
10161	The REQARG parameter in the IOCTL parameter list is zero.	End the call.	Correct the program.
10163	A 0 or negative data length was found for a RECV, RECVFROM, or RECVMSG call.	Disable the subtask for interrupts. Sever the DLC path. Return an error code to the caller.	Correct the data length.
10167	The descriptor set size for a SELECT or SELECTEX call is less than or equal to zero.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the SELECT or SELECTEX call. You might have incorrect sequencing of socket calls.

Table 4. Sockets Extended ERRNOs (continued)

Error Code	Problem Description	System Action	Programmer's Response
10168	The descriptor set size *in bytes* for a SELECT or SELECTEX call is greater than 252. A number greater than the maximum number of allowed sockets (2000 is maximum) has been specified.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the descriptor set size.
10170	A zero or negative data length was found for a SEND or SENDMSG call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the data length in the SEND call.
10174	A zero or negative data length was found for a SENDTO call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the data length in the SENDTO call.
10178	The SETSOCKOPT option length is less than the minimum length.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the OPTLEN parameter.
10179	The SETSOCKOPT option length is greater than the maximum length.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the OPTLEN parameter.
10184	A data length of zero was specified for a WRITE call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the data length in the WRITE call.
10186	A negative data length was specified for a WRITE or WRITEV call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the data length in the WRITE call.
10190	The GETHOSTNAME option length is less than 24 or greater than the maximum length.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the length parameter.
10193	The GETSOCKOPT option length is less than the minimum or greater than the maximum length.	End the call.	Correct the length parameter.
10197	The application issued an INITAPI call after the connection was already established.	Bypass the call.	Correct the logic that produces the INITAPI call that is not valid.
10198	The maximum number of sockets specified for an INITAPI exceeds 2000.	Return to the user.	Correct the INITAPI call.
10200	The first call issued was not a valid first call.	End the call.	For a list of valid first calls, refer to the section on special considerations in the chapter on general programming .
10202	The RETARG parameter in the IOCTL call is zero.	End the call.	Correct the parameter list. You might have incorrect sequencing of socket calls.
10203	The requested socket number is a negative value.	End the call.	Correct the requested socket number.

Table 4. Sockets Extended ERRNOs (continued)

Error Code	Problem Description	System Action	Programmer's Response
10205	The requested socket number is a duplicate.	End the call.	Correct the requested socket number.
10208	The NAMELEN parameter for a GETHOSTBYNAME call was not specified.	End the call.	Correct the NAMELEN parameter. You might have incorrect sequencing of socket calls.
10209	The NAME parameter on a GETHOSTBYNAME call was not specified.	End the call.	Correct the NAME parameter. You might have incorrect sequencing of socket calls.
10210	The HOSTENT parameter on a GETHOSTBYNAME or GETHOSTBYADDR call was not specified.	End the call.	Correct the HOSTENT parameter. You might have incorrect sequencing of socket calls.
10211	The HOSTADDR parameter on a GETHOSTBYNAME or GETHOSTBYADDR call is incorrect.	End the call.	Correct the HOSTADDR parameter. You might have incorrect sequencing of socket calls.
10212	The resolver program failed to load correctly for a GETHOSTBYNAME or GETHOSTBYADDR call.	End the call.	Check the JOBLIB, STEPLIB, and linklib datasets and rerun the program.
10213	Not enough storage is available to allocate the HOSTENT structure.	End the call.	Increase the user storage allocation for this job.
10214	The HOSTENT structure was not returned by the resolver program.	End the call.	Ensure that the domain name server is available. This can be a nonerror condition indicating that the name or address specified in a GETHOSTBYADDR or GETHOSTBYNAME call could not be matched.
10215	The APITYPE parameter on an INITAPI call instruction was not 2 or 3.	End the call.	Correct the APITYPE parameter.
10218	The application programming interface (API) cannot locate the specified TCP/IP.	End the call.	Ensure that an API that supports the performance improvements related to CPU conservation is installed on the system and verify that a valid TCP/IP name was specified on the INITAPI call. This error call might also mean that EZASOKIN could not be loaded.
10219	The NS parameter is greater than the maximum socket for this connection.	End the call.	Correct the NS parameter on the ACCEPT, SOCKET or TAKESOCKET call.
10221	The AF parameter of a SOCKET call is not AF_INET.	End the call.	Set the AF parameter equal to AF_INET.
10222	The SOCTYPE parameter of a SOCKET call must be stream, datagram, or raw (1, 2, or 3).	End the call.	Correct the SOCTYPE parameter.
10223	No ASYNC parameter specified for INITAPI with APITYPE=3 call.	End the call.	Add the ASYNC parameter to the INITAPI call.

Table 4. Sockets Extended ERRNOs (continued)

Error Code	Problem Description	System Action	Programmer's Response
10224	The IOVCNT parameter is less than or equal to zero, for a READV, RECVMSG, SENDMSG, or WRITEV call.	End the call.	Correct the IOVCNT parameter.
10225	The IOVCNT parameter is greater than 120, for a READV, RECVMSG, SENDMSG, or WRITEV call.	End the call.	Correct the IOVCNT parameter.
10226	Invalid COMMAND parameter specified for a GETIBMOPT call.	End the call.	Correct the COMMAND parameter of the GETIBMOPT call.
10229	A call was issued on an APITYPE=3 connection without an ECB or REQAREA parameter.	End the call.	Add an ECB or REQAREA parameter to the call.
10300	Termination is in progress for either the CICS transaction or the sockets interface.	End the call.	None.
10330	A SELECT call was issued without a MAXSOC value and a TIMEOUT parameter.	End the call.	Correct the call by adding a TIMEOUT parameter.
10331	A call that is not valid was issued while in SRB mode.	End the call.	Get out of SRB mode and reissue the call.
10332	A SELECT call is invoked with a MAXSOC value greater than that which was returned in the INITAPI function (MAXSNO field).	End the call.	Correct the MAXSOC parameter and reissue the call.
10999	An abend has occurred in the subtask.	Write message EZY1282E to the system console. End the subtask and post the TRUE ECB.	If the call is correct, call your system programmer.
20000	An unknown function code was found in the call.	End the call.	Correct the SOC-FUNCTION parameter.
20001	The call passed an incorrect number of parameters	End the call	Correct the parameter list.
20002	The CICS Sockets Interface is not in operation.	End the call	Start the CICS Sockets Interface before executing this call.

Appendix B. How to Read a Syntax Diagram

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

Symbols and Punctuation

The following symbols are used in syntax diagrams:

►► Marks the beginning of the command syntax.

► Indicates that the command syntax is continued.

I Marks the beginning and end of a fragment or part of the command syntax.

►◄ Marks the end of the command syntax.

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

Parameters

The following types of parameters are used in syntax diagrams:

Required
Required parameters are displayed on the main path.

Optional
Optional parameters are displayed below the main path.

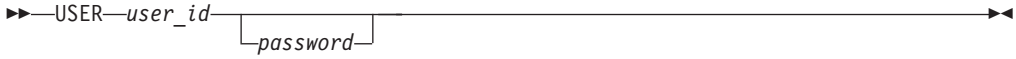
Default
Default parameters are displayed above the main path.

Parameters are classified as keywords or variables. Keywords are displayed in uppercase letters and can be entered in uppercase or lowercase. For example, a command name is a keyword.

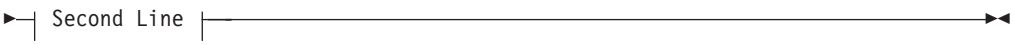
Variables are italicized, appear in lowercase letters, and represent names or values you supply. For example, a data set is a variable.

Syntax Examples

In the following example, the USER command is a keyword. The required variable parameter is *user_id*, and the optional variable parameter is *password*. Replace the variable parameters with your own values.



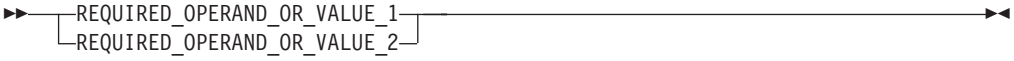
Longer than one line: If a diagram is longer than one line, the first line ends with a single arrowhead and the second line begins with a single arrowhead.



Required operands: Required operands and values appear on the main path line.

You must code required operands and values.

Choose one required item from a stack: If there is more than one mutually exclusive required operand or value to choose from, they are stacked vertically in alphanumeric order.



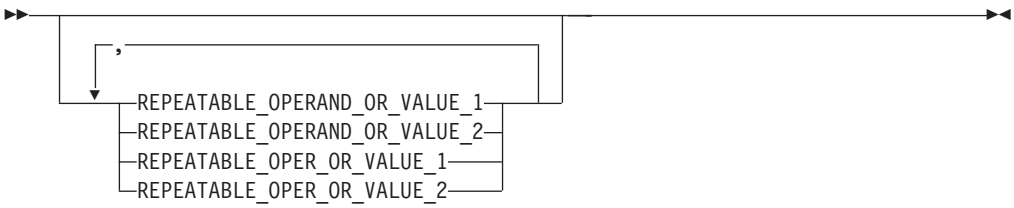
Optional values: Optional operands and values appear below the main path line.

You can choose not to code optional operands and values.

Choose one optional operand from a stack: If there is more than one mutually exclusive optional operand or value to choose from, they are stacked vertically in alphanumeric order below the main path line.

Repeating an operand: An arrow returning to the left above an operand or value on the main path line means that the operand or value can be repeated. The command means that each operand or value must be separated from the next by a comma.

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



If an operand or value can be abbreviated, the abbreviation is described in the text associated with the syntax diagram.

Case Sensitivity: TCP/IP commands are not case sensitive. You can code them in uppercase or lowercase.

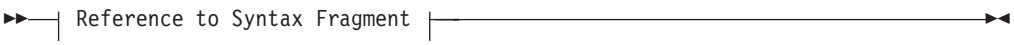
Nonalphanumeric characters: If a diagram shows a character that is not alphanumeric (such as parentheses, periods, commas, and equal signs), you must code the character as part of the syntax. In this example, you must code OPERAND=(001,0.001).

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

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

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

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



Syntax Fragment:

References to syntax notes appear as numbers enclosed in parentheses above the line. Do not code the parentheses or the number. An example of a syntax note identifier and note is shown below.

Notes:

1 An example of a syntax note.

Appendix C. Information Apars

This appendix lists information apars for IP and SNA books.

Notes:

1. Information apars contain updates to previous editions of the manuals listed below. Books updated for V1R2 are complete except for the updates contained in the information apars that may be issued after V1R2 books went to press.
2. Information apars are predefined for z/OS V1R2 Communications Server and may not contain updates.

IP Information Apars

Table 5 lists information apars for IP books.

Table 5. IP Information Apars

Title	z/OS CS V1R2	CS for OS/390 2.10 and z/OS CS V1R1	CS for OS/390 2.8	CS for OS/990 2.7	CS for OS/390 2.6	CS for OS/390 2.5
IP API Guide	ii12861	ii12371	ii11635	ii11558	ii11405	ii11144
IP CICS Sockets Guide	ii12862		ii11626	ii11559	ii11406	ii11145
IP Configuration			ii11620 ii12068 ii12353 ii12649	ii11555 ii11637 ii11995 ii12325	ii11402 ii11619 ii12066 ii12455	ii11159 ii11979 ii12315
IP Configuration Guide	ii12498	ii12362 ii12493				
IP Configuration Reference	ii12499	ii12363 ii12494 ii12712				
IP Diagnosis	ii12503	ii12366 ii12495	ii11628	ii11565	ii11411	ii11160 ii11414
IP Messages Volume 1	ii12857	ii12367	ii11630	ii11562	ii11408	ii11636
IP Messages Volume 2	ii12858	ii12368	ii11631	ii11563	ii11409	ii11281
IP Messages Volume 3	ii12859	ii12369	ii11632 ii12883	ii11564 ii12884	ii11410 ii12885	ii11158
IP Messages Volume 4	ii12860					
IP Migration	ii12497	ii12361	ii11618	ii11554	ii11401	ii11204
IP Network Print Facility	ii12864		ii11627	ii11561	ii11407	ii11150
IP Programmer's Reference	ii12505		ii11634	ii11557	ii11404	ii12496

Table 5. IP Information Apars (continued)

Title	z/OS CS V1R2	CS for OS/390 2.10 and z/OS CS V1R1	CS for OS/390 2.8	CS for OS/990 2.7	CS for OS/390 2.6	CS for OS/390 2.5
IP and SNA Codes	ii12504	ii12370	ii11917	Added TCP/IP codes to VTAM codes V2R6 ii11611	ii11361	ii11146 ii11097
IP User's Guide		ii12365	ii11625	ii11556	ii11403	ii11143
IP User's Guide and Commands	ii12501					
IP System Admin Guide	ii12502					
Quick Reference	ii12500	ii12364				

SNA Information Apars

Table 6 lists information apars for SNA books.

Table 6. SNA Information Apars

Title	z/OS CS V1R2	CS for OS/390 2.10 and z/OS CS V1R1	CS for OS/390 2.8	CS for OS/390 2.7	CS for OS/390 2.6	CS for OS/390 2.5
Anynet SNA over TCP/IP			ii11922	ii11633	ii11624	ii11623
Anynet Sockets over SNA			ii11921	ii11622	ii11519	ii11518
CSM Guide						
IP and SNA Codes		ii12370	ii11917	ii11611	ii11361	ii11097
SNA Customization	ii12872	ii12388	ii11923	ii11925 ii12008	ii11924 ii12007	ii11092 ii11621 ii12006
SNA Diagnosis	ii12490	ii12389	ii11915	ii11615	ii11357	ii11585
SNA Messages	ii12491	ii12382	ii11916	ii11610	ii11358	ii11096
SNA Network Implementation Guide	ii12487	ii12381	ii11911	ii11609 ii12683	ii11353 ii11493	ii11095
SNA Operation	ii12489	ii12384	ii11914	ii11612	ii11355	ii11098
SNA Migration	ii12486	ii12386	ii11910	ii11614	ii11359	ii11100
SNA Programming		ii12385	ii11920	ii11613	ii11360	ii11099
Quick Reference	ii12500	ii12364	ii11913	ii11616	ii11356	
SNA Resource Definition Reference	ii12488	ii12380 ii12567	ii11912 ii12568	ii11608 ii12569	ii11354 ii12259 ii12570	ii11094 ii11151 ii12260 ii12571
SNA Resource Definition Samples						

Appendix D. Notices

IBM may not offer all of the products, services, or features discussed in this document. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION ″AS IS″ WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs

and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Site Counsel
IBM Corporation
P.O.Box 12195
3039 Cornwallis Road
Research Triangle Park, North Carolina 27709-2195
U.S.A

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

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly

tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

This product includes cryptographic software written by Eric Young.

If you are viewing this information softcopy, photographs and color illustrations may not appear.

You can obtain softcopy from the z/OS Collection (SK3T-4269), which contains BookManager and PDF formats of unlicensed books and the z/OS Licensed Product Library (LK3T-4307), which contains BookManager and PDF formats of licensed books.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

ACF/VTAM
Advanced Peer-to-Peer Networking
AFP
AD/Cycle
AIX
AIX/ESA
AnyNet
APL2
APPN
AS/400
AT
BookManager
BookMaster
CBPDO
C/370
CICS
CICS/ESA
C/MVS
Common User Access
C Set ++
CT
CUA
DATABASE 2
DatagLANce
DB2
DFSMS
DFSMSdfp
DFSMShsm
DFSMS/MVS
DPI
Domino
DRDA
eNetwork
Enterprise Systems Architecture/370
ESA/390
ESCON
@server
ES/3090
ES/9000
ES/9370
EtherStreamer
Extended Services
FAA
Micro Channel
MVS
MVS/DFP
MVS/ESA
MVS/SP
MVS/XA
MQ
Natural
NetView
Network Station
Nways
Notes
NTune
NTuneNCP
OfficeVision/MVS
OfficeVision/VM
Open Class
OpenEdition
OS/2
OS/390
OS/400
Parallel Sysplex
Personal System/2
PR/SM
PROFS
PS/2
RACF
Resource Link
Resource Measurement Facility
RETAIN
RFM
RISC System/6000
RMF
RS/6000
S/370
S/390
SAA
SecureWay
Slate
SP
SP2
SQL/DS
System/360

FFST
FFST/2
FFST/MVS
First Failure Support Technology
GDDM
Hardware Configuration Definition
IBM
IBMLink
IBMLINK
IMS
IMS/ESA
InfoPrint
Language Environment
LANStreamer
Library Reader
LPDA
MCS
System/370
System/390
SystemView
Tivoli
TURBOWAYS
UNIX System Services
Virtual Machine/Extended Architecture
VM/ESA
VM/XA
VSE/ESA
VTAM
WebSphere
XT
z/Architecture
z/OS
zSeries
400
3090
3890

Lotus, Freelance, and Word Pro are trademarks of Lotus Development Corporation in the United States, or other countries, or both.

Tivoli and NetView are trademarks of Tivoli Systems Inc. in the United States, or other countries, or both.

DB2 and NetView are registered trademarks of International Business Machines Corporation or Tivoli Systems Inc. in the U.S., other countries, or both.

The following terms are trademarks of other companies:

ATM is a trademark of Adobe Systems, Incorporated.

BSC is a trademark of BusiSoft Corporation.

CSA is a trademark of Canadian Standards Association.

DCE is a trademark of The Open Software Foundation.

HYPERchannel is a trademark of Network Systems Corporation.

UNIX is a registered trademark in the United States, other countries, or both and is licensed exclusively through X/Open Company Limited.

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

ActionMedia, LANDesk, MMX, Pentium, and ProShare are trademarks of Intel Corporation in the United States, other countries, or both. For a complete list of Intel trademarks, see http://www.intel.com/tradmarx.htm.

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

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Readers' Comments — We'd Like to Hear from You

z/OS Communications Server
IP IMS Sockets Guide
Version 1 Release 2

Publication No. SC31-8830-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

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.

Name _____ Address _____

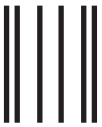
Company or Organization _____

Phone No. _____

Readers' Comments — We'd Like to Hear from You
SC31-8830-00

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 Corporation
Software Reengineering
Department G7IA/ Bldg 503
Research Triangle Park, NC
27709-9990

Fold and Tape | **Please do not staple** | Fold and Tape

Cut or Fold
Along Line

Program Number: 5694-A01

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC31-8830-00

Spine information:

z/OS Communications Server

z/OS V1R2.0 CS: IP IMS Sockets Guide

Version 1
Release 2