OS/390 eNetwork Communications Server

IP IMS Sockets Guide

Version 2 Release 5

SC31-8519-00

OS/390 eNetwork Communications Server

IP IMS Sockets Guide

Version 2 Release 5

SC31-8519-00

Note:

Before using this information and the product it supports, be sure to read the general information under Appendix C, “Notices” on page 243.

First Edition (March 1998)

This edition applies to OS/390 V2R5 (program number 5647-A01).

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
Department CGMD
P.O. Box 12195
Research Triangle Park, North Carolina 27709
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.s390.ibm.com/os390
IBMLink:	CIBMORCF at RALVM13
IBM Mail Exchange:	USIB2HPD at IBMMAIL

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 IBM TCP/IP Version 3 Release 2 for MVS with IMS/ESA, Version 4 and above. It describes the call interface and the supporting functions.

This book addresses the following topics:

- IMS client/server application design
- IMS OTMA Connection Server
- The IMS Listener
- The IMS Assist function
- The IMS socket calls, including call syntax conventions

TCP/IP Version 3 Release 2 for MVS is an integral part of the OS/390 V2R5 family of products. For an overview and mapping of the documentation available for OS/390 V2R5, see the *OS/390 Information Roadmap*.

Who Should Use This Book

This book is intended for programmers who have some familiarity with IMS Transaction Manager and TCP/IP for MVS, and who need to develop IMS TCP/IP client/server applications. For more information about how programmers with different levels of experience should use this book see How to Use This Book.

How to Use This Book

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 TCP/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 5, “Principles of Operation” on page 77.
- Experienced TCP/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 6, “How to Write an IMS TCP/IP Client Program” on page 89 and Chapter 9, “CALL Instruction Application Programming Interface (API)” on page 113.
- 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 9, “CALL Instruction Application Programming Interface (API)” on page 113.

- IMS system programmers and communication programmers are responsible for the IMS system itself. These readers should read Chapter 8, “How to Customize and Operate the IMS Listener” on page 105.

Where to Find Related Information on the Internet

You may find the following information helpful.

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

You can read more about VTAM, TCP/IP, OS/390, and IBM on these Web pages:

Home Page	Uniform Resource Locator (URL)
VTAM	http://www.networking.ibm.com/vta/vtaprod.html
TCP/IP	http://www.networking.ibm.com/tcm/tcmprod.html
OS/390	http://www.s390.ibm.com/os390/
IBM eNetwork Communications Server	http://www.software.ibm.com/enetwork/commserver.html
IBM	http://www.ibm.com/

For definitions of the terms and abbreviations used in our books, you can view or download the latest *IBM Networking Softcopy Glossary* at the following URL:

http://www.networking.ibm.com/nsg/nsgmain.htm

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.

Summary of Changes

SC31-7186-03: TCP/IP Version 3 Release 2 for MVS

This is the fourth edition of this book. This book supports TCP/IP Version 3 Release 2 for MVS and the OS/390 V2R5 family of products.

The updates contained in this edition are effective only if the latest level of maintenance has been applied. New and changed information is indicated by a revision bar (|).

New Information

The following enhancements are new for this revision:

- The IMS TCP/IP Open Transaction Manager Access Connection (OTMA) server is a new function that allows TCP/IP clients to connect directly to IMS. Using the IMS TCP/IP OTMA Connection server with the EZAIMSO0 exit allows a remote client to connect directly to IMS. The server is available from the IMS Web page and the EZAIMSO0 exit is available as PTF UQ03104. See Chapter 3, "Using IMS TCP/IP OTMA Connection from TCP/IP Clients" on page 29 for a description of the IMS TCP/IP OTMA Connection server.
- "JCL for Linking the IMS Listener" on page 180 is a new section that includes two sample JCL programs for linking the IMS Listener.
- Return code 1044 describes the EIBMINVALIDTCB message.

Changed Information

The following information has changed for this revision:

- The description of return code 10332 no longer describes SELECTEX.
- The RETCODE value for the WRITE function, which is described in Chapter 9, "CALL Instruction Application Programming Interface (API)" on page 113.
- This book has been divided into parts to reflect the use of the IMS Listener and the IMS TCP/IP OTMA Connection Server:
  - Part 1 is a general introduction section.
  - Part 2 describes how to use the IMS OTMA TCP/IP Connection server. It also includes sample programs.
  - Part 3 describes the use of the traditional IMS Listener and IMS Assist Module.

SC31-7186-02: TCP/IP Version 3 Release 2 for MVS

This is the third edition of this book. This book supports TCP/IP Version 3 Release 2 for MVS and the OS/390 V2R5 family of products.

The updates contained in this edition are effective only if the latest level of maintenance has been applied. New and changed information is indicated by a revision bar (|).

New Information

The following enhancements are new for this revision:

- Improved C Sockets and Sockets Extended application program interfaces (APIs) (also referred to as High-Performance Native Sockets, or HPNS): The Sockets Extended APIs, which are the Macro and Call Instruction interfaces, no longer use the inter-user communication vehicle (IUCV) address space. This change improves performance and reduces CPU cycles in an MVS host transporting data with TCP/IP. Applications written on previous versions of TCP/IP for MVS can run unchanged.
- New calls for the Call Instruction interface:
  - The *GETIBMOPT* call returns the number of TCP/IP images installed on a given MVS system and their status, versions, and names.
  - The *READV* call reads data on a socket and stores it in a set of buffers.
  - The *RECVMSG* call receives messages on a socket and stores them in an array of message headers.
  - The *SELECTEX* call monitors a set of sockets, a time value, and an ECB or ECB list. It completes when either one of the sockets has activity, the time value expires, or an ECB is posted.
  - The *SENDMSG* call sends messages on a socket passed from an array of messages.
  - The *WRITEV* call writes data on a socket into a set of buffers.

Changed Information

- Chapter 2, Introduction to TCP/IP for IMS has been expanded to include a more complete overview of sockets programming and the client/server environment.
- The sample JCL for starting the IMS Listener has been moved from the Appendix to Chapter 8, “How to Customize and Operate the IMS Listener” on page 105.

Various minor editorial and technical updates have been applied to this edition.

SC31-7186-01: OS/390 V2R5 Release 1 - TCP/IP Version 3 Release 1 for MVS

This is the second edition of this book. This book supports TCP/IP Version 3 Release 2 for MVS and the OS/390 V2R5 family of products.

The updates contained in this edition are effective only if the latest level of maintenance has been applied. New and changed information is indicated by a revision bar (|).

Changed Information

Various minor editorial and technical updates have been applied to this edition.

SC31-7186-00: TCP/IP Version 3 Release 1 for MVS

This is a new book for TCP/IP Version 3 Release 1 for MVS. This book describes how to use IMS TCP/IP.

Part 1. IMS Overview

Chapter 1. Using TCP/IP with IMS

The purpose of this chapter is to introduce you to the IMS TCP/IP feature. The 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.*** IMS TCP/IP is an optional feature that can be added to the TCP/IP for MVS product. 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 for MVS product. (See *TCP/IP for MVS: User's Guide* 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 for MVS. 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 for MVS. TCP/IP passes the request to the IMS OTMA Connection server, or passes the request on 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 TCP/IP OTMA Connection server, which is similar in function to the IMS Listener in providing connectivity
- 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 TCP/IP OTMA Connection Server

The new IMS TCP/IP OTMA Connection server enables remote clients to connect to IMS to perform IMS transactions. The IMS Listener is still available and supported, but IMS TCP/IP OTMA Connection server is the recommended way for remote clients to connect to IMS.

Note: You should continue to use the IMS Listener for explicit-mode transactions.

The IMS TCP/IP OTMA Connection server is a feature of the host web services (HWS) component and is available with IMS Version 5.1. The IMS TCP/IP OTMA Connection server supports an exit routine, EZAIMSO0, that is used to interface between TCP/IP clients and existing IMS application programs.

Unlike the IMS Listener, the IMS TCP/IP OTMA Connection server maintains connection until the entire connection is complete. The IMS TCP/IP OTMA Connection server is capable of maintaining a variable number of concurrent connection requests.

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

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.

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 for MVS. Figure 1 shows how IMS TCP/IP gives a variety of remote users peer-to-peer communication with IMS applications. You can choose either the IMS OTMA Connection server or the IMS Listener for remote clients to complete transaction requests to IMS.

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 for MVS 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.

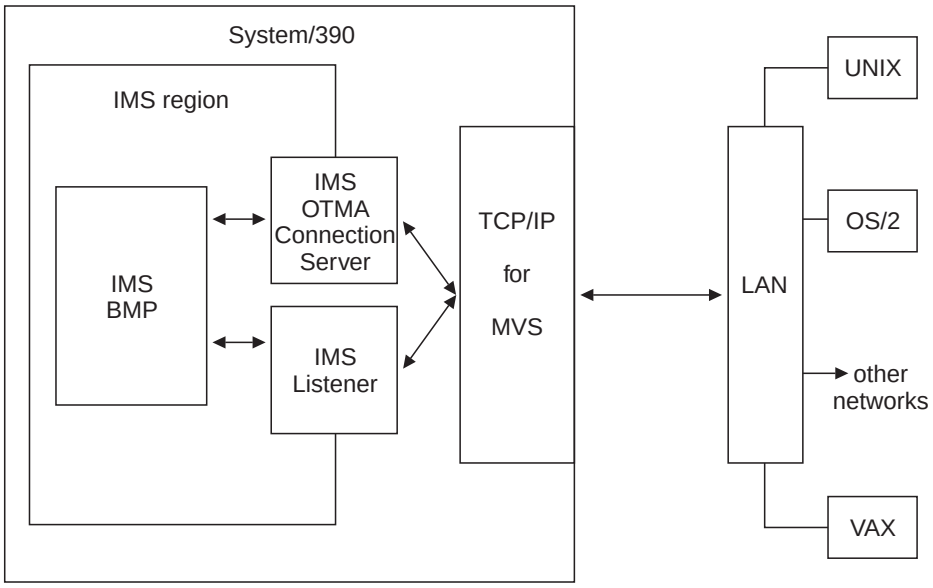


Figure 1. The Use of TCP/IP with IMS

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/IPoffers 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 for MVS 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 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.

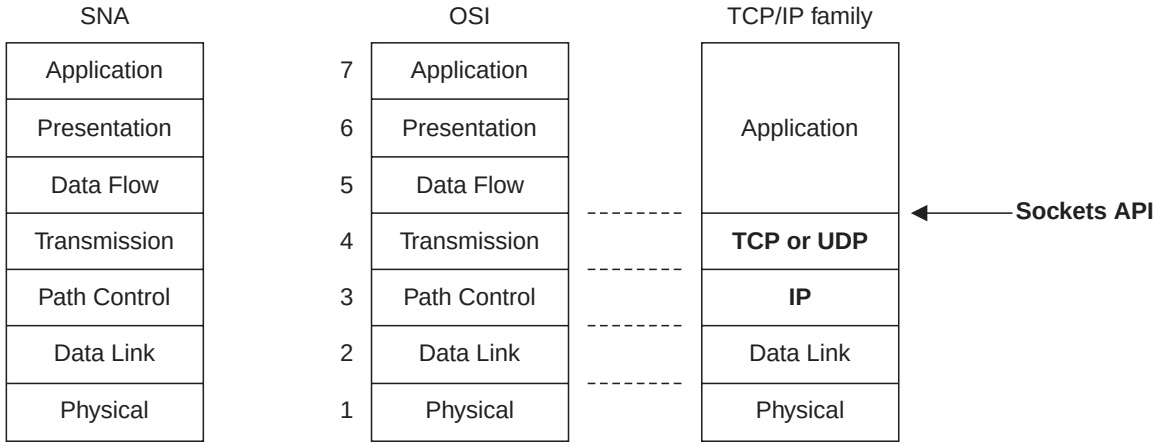


Figure 2. TCP/IP Protocols when compared to the OSI Model and SNA

The protocols implemented by TCP/IP for MVS 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

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 for MVS 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 *OS/390 eNetwork Communications Server: IP API 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; name Sockets Extended. See Chapter 9, “CALL Instruction Application Programming Interface (API)” on page 113 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

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

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.

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.

---

[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.

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 (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. 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. Most machine architectures (like IBM PC's and mainframes) define the lowest memory address to be the high-order bit, which is called *big endian*. However, some architectures 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. Since IBM systems are big-endian, the only time you need to be concerned about Network Byte Order is when a little-endian system attempts to make contact with an IMS system. In such a case, the burden for conversion to Network Byte Order is usually on the little-endian program.

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 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.

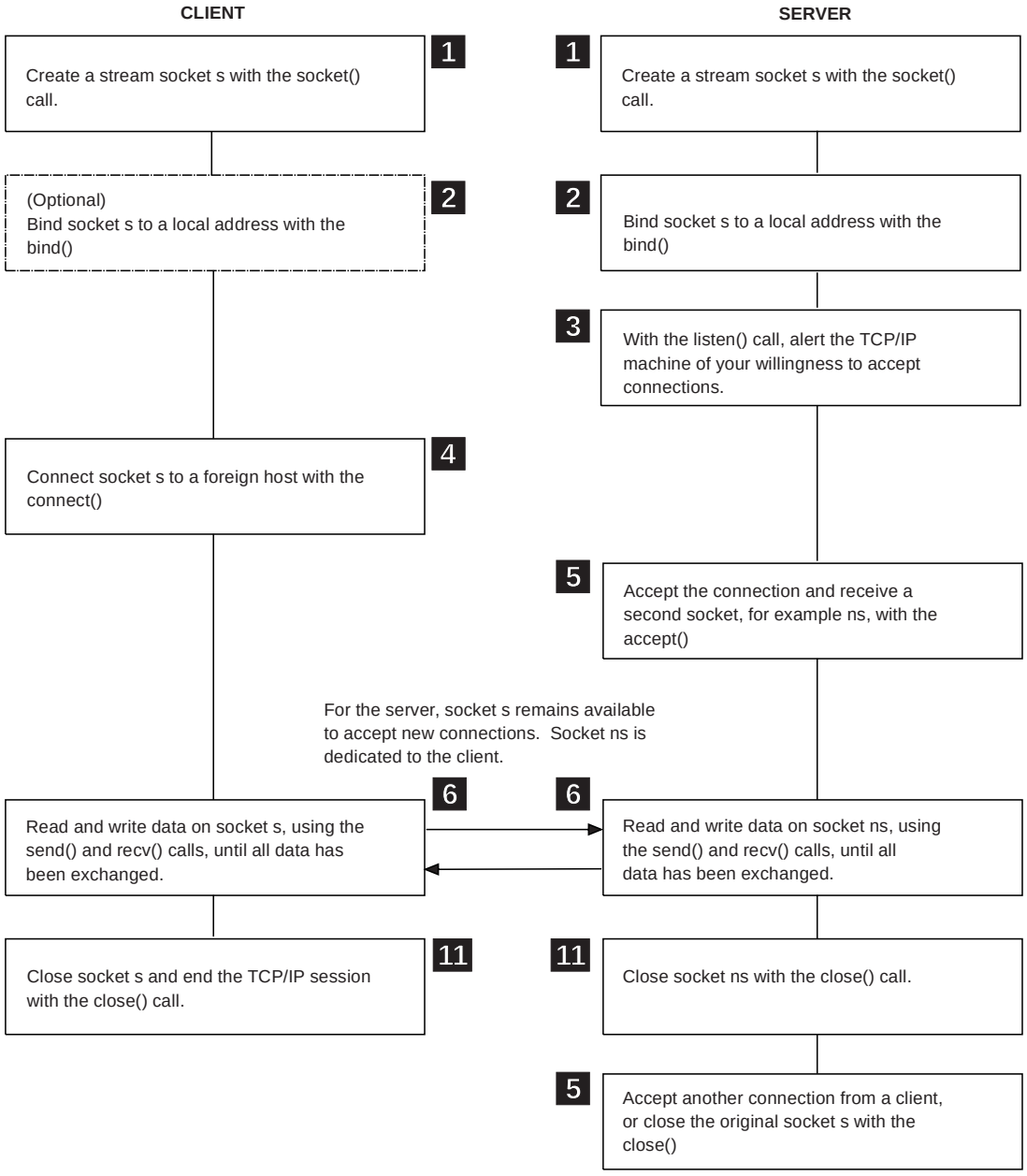


Figure 3. A Typical Client Server Session

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, once Client A starts a transaction with the server, Client B cannot make a call until A has finished.

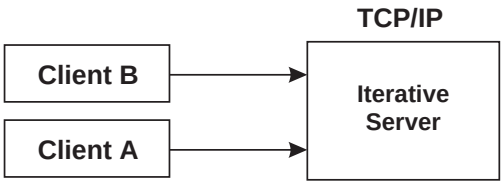


Figure 4. An Iterative Server

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 8, "How to Customize and Operate the IMS Listener" on page 105.

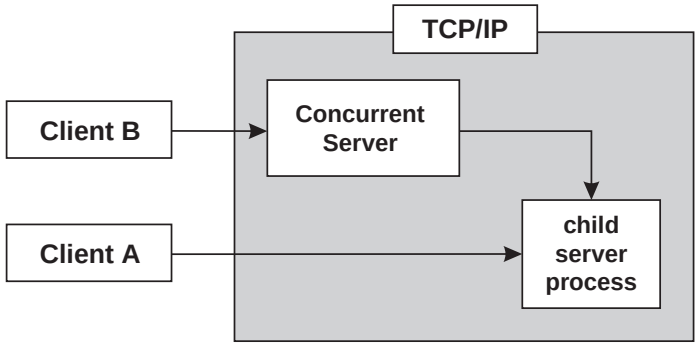


Figure 5. A Concurrent Server

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 9, "CALL Instruction Application Programming Interface (API)" on page 113.

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 TCPIP 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 0, the TCPIP 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 167.

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 117.

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 142.

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**

[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.

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 115.

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, MVS 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 22 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 167 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 119 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 143.
- For an example of the WRITE call, see “WRITE” on page 170.

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 118.

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.

```
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 paramater 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
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 139.

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

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 TCPIP 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.

---

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

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

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.

An example of a concurrent server is the IMS Listener. It is described in Chapter 8, "How to Customize and Operate the IMS Listener" on page 105. Figure 5 on page 14 shows a concurrent server.

What You Need to Run IMS TCP/IP

IMS TCP/IP OTMA Server runs on an MVS/SP host system running IMS Version 5.1 and TCP/IP for MVS Version 3 Release 2 with PTF UQ03104.

IMS TCP/IP using the IMS Listener and IMS Assist Module is designed for use on an MVS/SP host system running: Version 4 and TCP/IP for MVS Version 3 Release 1 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 for MVS

TCP/IP for MVS 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`

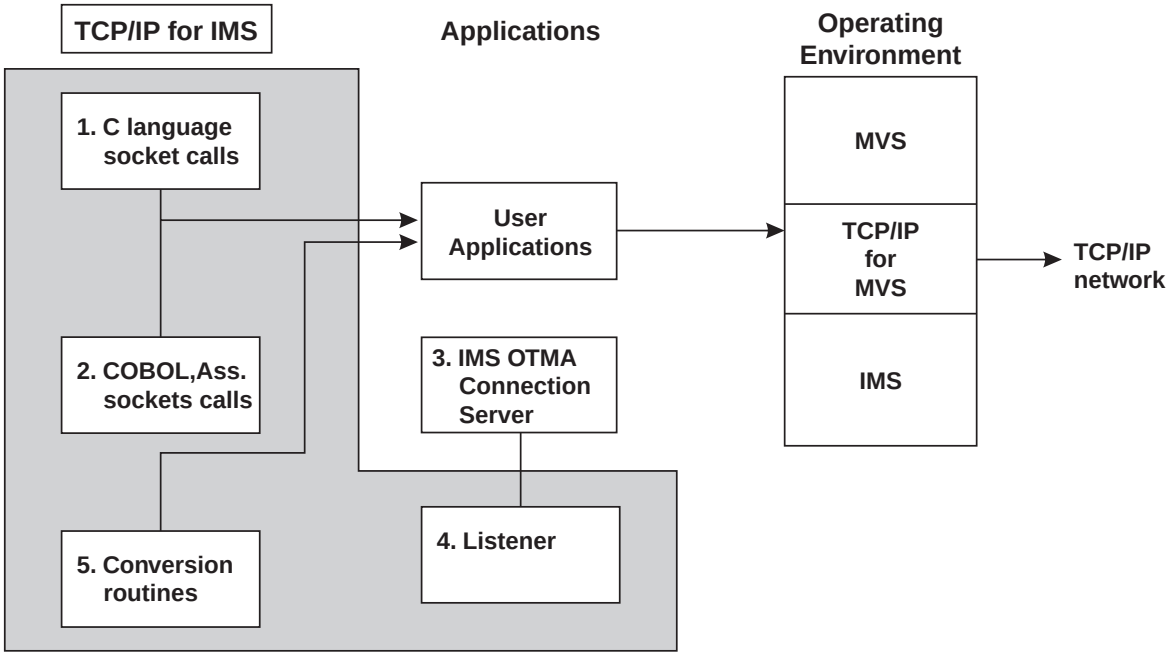


Figure 7. How User Applications Access TCP/IP Networks with IMS TCP/IP

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 IMS TCP/IP OTMA Connection Server (3)

The IMS OTMA Connection server, along with the EZAIMSO0 exit routine, allows a remote client to make connection requests to IMS.

The Listener (4)

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 (5)

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

- The IMS TCP/IP OTMA Conncection server, which is shipped by IMS (go to URL `www.software.ibm.com/data/ims/about/imsweb` to download the code).
- EZAIMS00, which is shipped by TCP/IP for MVS as PTF
- 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 OTMA Connection Server

Chapter 3. Using IMS TCP/IP OTMA Connection from TCP/IP Clients

This chapter describes the call sequences and input/output data formats used by the client program for IMS transaction using the IMS TCP/IP Open Transaction Manager Access (OTMA) connection server and the EZAIMSO0 exit. The IMS TCP/IP OTMA Connection server, along with the EZAIMSO0, is similar to using the IMS Listener, with the following differences:

- With the IMS OTMA Connection server, all output from the IMS transaction goes through the IMS message queue like normal IMS processing.
- TCP/IP clients connect to the IMS OTMA Connection server, instead of the IMS Listener. With the IMS OTMA Connection server, client transactions do not have to be modified and the details of the requestor are transparent.
- The IMS OTMA Connection Server supports RACF security.
- IMS transactions are no longer connected directly to the TCP/IP client, so the GIVESOCKET and TAKESOCKET functions are not used (which can improve performance).

The IMS OTMA connection server is available on an MVS or OS/390 platform if you have IMS/ESA Version 5. You can download the IMS TCP/IP OTMA Connection server component from the IMS Web Page at the following URL:

`www.software.ibm.com/data/ims/about/imsweb`

See the IMS OTMA Connection User's Guide on the IMS Web Page for installation and configuration information at the following URL:

`www.software.ibm.com/data/ims/about/imsweb/userg/index.html#`

The EZAIMSO0 exit is available with TCP/IP for MVS Version 3 Release 2 with PTF UQ03104.

IMS TCP/IP OTMA Connection Server Overview

The new IMS TCP/IP OTMA Connection server is a replacement of the IMS Listener and IMS Assist module for remote clients to connect to IMS, and is the recommended method for TCP/IP-to-MVS connections. The IMS Listener and IMS Assist Module are still supported, and are required for explicit-mode transactions.

Note: The IMS TCP/IP OTMA Connection server does not support explicit-mode transactions. You should continue to use the IMS Listener for explicit-mode transactions.

The IMS TCP/IP OTMA Connection server allows remote socket clients to connect to IMS using normal unmodified IMS transactions. These transactions might already exist and can be used by other types of clients, like an IBM 3270 terminal. The IMS OTMA Connection server can also use IMS-mode transactions that were written in previous releases of the the IMS Socket feature.

This server works with the IMS host web services (HWS) component as a concurrent server main process. When the client requests the services of an IMS message processing program (MPP), it connects to the IMS TCP/IP OTMA Con-

nection server on a host that supports the IMS transaction code of that MPP. The IMS TCP/IP OTMA Connection server receives the request, forwards a transaction over the OTMA interface to IMS. IMS schedules the requested MPP; the MPP starts, receives the input, processes the transaction, and inserts output messages to the IMS message queue. When the transaction reaches its synchronization point, IMS forwards the output messages over the OTMA interface to the IMS TCP/IP OTMA Connection server, and the server transmits the output to the remote client over the socket connection.

How the Connection Is Established

Support for TCP client is implemented as an exit routine, EZAIMSO0, in the IMS HWS. The following sections describe the function of the IMS TCP/IP OTMA Connection server, with EZAIMSO0, when connecting remote clients to IMS.

Requesting an IMS Transactions

The client initiates the request by passing an IMS-request message (IRM) as the first message segment to the IMS TCP/IP OTMA Connection server. The IRM determines which IMS receives the request and the name of the transaction to be scheduled.

IMS-Request Message Segment

To initiate a connection with an IMS server, the client first issues an IMS-request message segment (IRM), which tells the IMS OTMA TCP/IP Connection server which transaction to schedule. The value of the IMSdest field must match the ID keyword on a DATASTORE definition in the HWS configuration data set.

The remote client sends the IRM to the port number that you specified on the PORTID keyword on the TCP/IP definition in the HWS configuration data set.

The format of the IMS-request message segment (IRM) follows:

Field	Format	Meaning
IRMMask	DSECT	IMS request message dsect.
IRM_Len	H	Length of the IRM (in binary) including this field. This field is sent in network byte order.
IRM_Rsv	H	Reserved.
IRM_Id	CL8	Identifying string. Always *IRMREQ*. If the client data stream will be sent in ASCII, the IRMId field should also be transmitted in ASCII because the OTMA Listener uses this field to determine whether ASCII to EBCDIC translation is required.
IRM_TrnCod	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 OTMA Listener has determined that data will be transmitted in ASCII, it translates the transaction code to EBCDIC before any further processing is done.
IRM_IMSDestId	CL8	The IMS destination ID of the IMS, where the transaction will run. The IMS destination IDs are defined in the configuration file for the IMS TCP/IP OTMA Connection server. This field must match the ID keyword on a DATASTORE definition in the IMS host web server (HWS) configuration file.
IRM_Lterm	CL8	Optional field the client can use to supply an lterm name for the IMS transaction. The field must be set to blanks if not used.
IRM_Flag	X	Allows the client to pass information to the exit. Currently only one flag setting is supported.
IRM_MFSREQ	X'80'	TCP/IP client request the MVS MOD name be returned by IMS. Setting the MVSReq flag results in clients receiving a request mod message (RMM) as the first segment in the output buffer. See “Request-Mod Message Segment” on page 31 for a description of the RMMs.
IRM_Rsv2	CL3	Reserved.
IRM_UsrDat	0C	The beginning of user security data.
IRMMask_len	*	Size of IRM.

Request-Mod Message Segment

If you set the IRM_MVSReq flag to X'80, an request—mod message (RMM) segment is sent as an output stream to the client. The RMM indicates the MVS MOD name used by the IMS application.

Field	Format	Meaning
RMMask	DSECT	Request mod message dsect.
RMM_Len	H	Length of the RMM.
RRM_Rsv	H	Reserved.
RRM_Id	CL8	RMM id '*REQMOD*' DIdentifying string. (in ASCII and EBCDIC)
RRM_Modname	CL8	MFS MOD for this output.
RRMMask_len	*	Size of RRM.

Verifying the Transaction Request

The IMS TCP/IP OTMA Connection server performs several tests to ensure that the requested transaction should be accepted. The following actions depend on the results of the verification:

- If the transaction request is *rejected*, the IMS TCP/IP OTMA Connection server 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 sent back to the IMS TCP/IP OTMA Connection server (IMS TCP/IP OTMA Connection server does not return a status message to the client).

Request-Status Message Segment

If the IMS TCP/IP OTMA Connection server sends a request-status-messages segment to the client (indicating an error condition), the request-status message segment (RSM) contains a return and reason code indicating the type of error. 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 OTMA Listener has determined that the client is transmitting in ASCII.
RSMRetcod	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 OTMA listener. Codes greater than 100 can be assigned by the user-written security exit.

Request-Status Message Reason Codes: If the IMS TCP/IP OTMA Connection sends a request-status message (RSM) segment to the client (indicating an error condition), the RSM contains a return and reason code indicating the type of error:

- If the user-supplied security exit rejects a transaction request, it sets the return code and reason code, and returns control to EZAIMSO0. EZAIMSO0 builds the RSM to send to the client.
- If EZAIMSO0 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.

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

  8 A negative acknowledgment (NAK) was returned by IMS/OTMA, but no sense code or return code was returned.

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

- If the HWS detects an error that causes a transaction request to fail, a RSM with return code 12 (X'0C') and a reason or sense code from the following list are sent to the client:

1	NACK_NOT_IN_SESSION
2	NACK_BLOCKED
3	NACK_PROTOCOL_ERROR
4	NACK_BAD_CORRELATOR
5	NACK_DUPLICATE_SEGMENT
6	NACK_XSF_BADRC
7	NACK_OUT_OF_MTE
8	NACK_CLIENTBID_SEC_FAILED
9	NACK_UNKNOWN_OTMA_CMD
10	NACK_INPUT_IS_DATA
11	NACK_INVALID_MSG_TYPE
12	NACK_UNKNONW_RESPONSE_TYPE
13	NACK_INVALID_IMS_CONV_CONT
14	NACK_UNABLE_CREATE_TPIPE
15	NACK_TPIPE_STOPPED
16	NACK_NO_STATE_DATA
17	NACK_INVALID_COMMIT_MSG
18	NACK_PREFIX_TOO_BIG
19	NACK_NO_MSG_HASHTABLE_SIZE
20	NACK_INVALID_STATE
21	NACK_NO_MSG_HASHTABLE
22	NACK_MEM_NOT_ACTIVE
23	NACK_INVALID_SYNC_LEVEL
24	NACK_INVALID_TPIPE_NAME
25	NACK_INVALID_NEMBER_NAME
26	NACK_ERROR_IN_MESSAGE
27	NACK_IMS_IN_SHUTDOWN
28	NACK_INVALID_COMMIT_MODE
29	NACK_INVALID_UDATA_LEN
30	NACK_IMS_UDATA_LEN
31	NACK_INVALID_RECOV_SEG_NUM
32	NACK_NO_APPL-DATAA
33	NACK_NO_CHAIN_FLAG
34	NACK_UNABLE_FIND_TPIPE
36	NACK_CONV_IN_PROCESS
60	NANDCOMP - Component not found
61	NFNDFUNC - Function not found

62 Nfnddst - Datastore not found

63 DSCLOSE - HWS in shutdown

64 STP/CLSE - Datastore in stop or close process

65 DSCERR - Datastore communication error

66 STOPCMD - Datastore was stopped by command

67 COMMERR - Datastore communication error to pending clients.

Scheduling the Transaction

An IMS transaction is scheduled by the IMS OTMA Connection Server, where the IRM is quered for normal processing. The Complete-Status Message indicates the the transaction has completed successfully.

Complete-Status Message Segment

The complete-status message (CSM) segment is sent by the OTMA Listener 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 follows:

Field	Format	Description
	H	Length of 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.

IMS OTMA Connection Security Exit

The IMS OTMA Connection server can be link-edited with an installation developed security exit. The exit routine has the same name as the IMS Listener exit (IMSLSECX); therefore, you can use the same security exit routine for both the IMS Listener and IMS OTMA Connection server. See “IMSLSECX — IMS BMP and OTMA Listener Security Exit” on page 62 for a sample that includes and IMS Listener and IMS OTMA Connection server security exit.

If the IMS OTMA connection server calls the security exit, 2 extra parameters are passed to the exit routine. These 2 fields can be used by the exit routine to return a userid and a group ID, which are added to the OTMA headers. The userid that is passed by the security exit is the userid that IMS uses to authorize access to the IMS transaction and is the userid that is passed to the IMS application in the userid field of the IO PCB.

If you have created an IMS OTMA listener security exit routine and linked the exit with EZAIMSO0, this exit routine will be invoked for every IRM message. The user data area is passed to the exit routine and is based on installation standards for the layout of the user data area. The exit routine can perform various security tasks, such as verifying a userid and password. If you develop a security exit for the IMS OTMA Connection server and the IMS Listener, the following user data layout in the IRM must match the TRM user data:

USERID 8 byte userid of the client user

PASSWORD 8 byte userid of the client user

NEWPASW 8 byte optional new password of the client user

GROUP 8 byte optional RACF group ID

The IMS OTMA Connection server security exit returns a userid and a group ID (optional) to the IMS OTMA Connection server. If these two fields are blank, the server passes the transaction on to IMS without any userid information. The IMS OTMA Connection server can choose which userid and group ID to return to the OTMA listener.

The following sample JCL shows how you can include the security exit in the IMS OTMA Connection server.

```
//LINKOTMA EXEC PGM=IEWL,PARM='LIST,XREF,REUS'
//SYSPRINT DD SYSOUT=*
//AEZAMOD1 DD DSN=TCPIP.V3R2M0.AEZAMOD1,DISP=SHR
//SYSLMOD  DD DSN=TCPIP.IMSWEB2.LOAD,DISP=SHR
//SYSUT1   DD UNIT=SYSDA,DCB=BLKSIZE=1024,
//         SPACE=(1024,(200,20))
//SYSLIB   DD DSN=TCPIP.T18AIMS.RESLIB,DISP=SHR
//         DD DSN=TCPIP.TPI32.LOAD,DISP=SHR
//         DD DSN=TCPIP.V3R2M0.SEZACMTX,DISP=SHR
//         DD DSN=TCPIP.V3R2M0.SEZALINK,DISP=SHR
//         DD DSN=TCPIP.V3R2M0.SEZATCP,DISP=SHR
//EXIT     DD DSN=TCPIP.TPI32.LOAD,DISP=SHR
//SYSLIN   DD DSN=TCPIP.ITSC.OBJ(EZAIMSO0),DISP=SHR
//         DD DDNAME=SYSIN
//SYSIN    DD *

    ORDER CMCOPYR
  INCLUDE EXIT(IMSLSECX)
  INCLUDE AEZAMOD1(EZAAE00B)
  INCLUDE AEZAMOD1(EZAAE02A)
  INCLUDE AEZAMOD1(EZAAE05F)
    ENTRY EZAIMSO0
     MODE RMODE(24) AMODE(31)
     NAME EZAIMSO0(R)
/*
```

If you do not link a security exit with the IMS OTMA Connection server and you still want RACF to verify the userid, you must provide a default RACFID keyword in the HWS configuration file. If a RACFID keyword is available, the userid authorization is based on either a userid or group ID, which is passed in the IRM user data section, or on the RACFID keyword (if no user data information is available). The layout of the user data section is as follows:

USERID 8 byte userid of the client user

GROUP 8 byte optional RACF group ID

The userid is not verified, because there is no password passed from the client. If you need to verify the userid, you have to create an IMS OTMA Connection server security exit as described earlier.

If the IRM does not include any optional user data, but the installation has specified a RACFID keyword in the HWS configuration dataset, the IMS transactions that are

initiated by the OTMA listener is authorized against the userid specified on the RACFID keyword in the TCPIP statement.

Chapter 4. IMS OTMA Connection Server Sample Programs

This appendix contains the following sample for the IMS OTMA Connection server..

TPIIMSDP — Triple-Purpose IMS Server Program

This IMS message processing program may be used from the following clien types:

- 3270 based (MFS input).
- Sockets using the IMS Listener and accessing the server as an implicit-mode socket server program.
- Sockets using the IMS OTMA Connection server.

The TPIIMSDP sample program is started through the IMS transaction code TPIPART.

```
 Identification Division.
*========================*

*-----------------------------------------------------------------*
*                                                                 *
* Name:        TPIIMSDP - DI21PART database query program.        *
*                                                                 *
* Trancode:    TPIPART                                            *
*                                                                 *
* Function:    Receives a part number, fetches data from the      *
*              DI21PART database and sends a message back.        *
*              Works for both MFS 3270, implicit-mode sockets     *
*              and OTMA sockets.  Triple-purpose MPP!             *
*              Due to support for implicit-mode, we call          *
*              CBLADLI instead of CBLTDLI.  If input is from      *
*              3270 or from OTMA listener, the assist code        *
*              will just pass all DLI calls (both DB and DC)      *
*              through to the real DLI language interface         *
*              module.                                            *
*                                                                 *
* Interface:   - none -                                           *
*                                                                 *
* Logic:       1. Receive input message                           *
*              2. Look up PARTROOT and STANINFO segments          *
*              3. Format output message according to              *
*                 defined layout                                  *
*              4. Insert output message and terminate             *
*                                                                 *
* Returncode:  - none -                                           *
*                                                                 *
* Written:     March 1997, ITSO Raleigh                           *
*                                                                 *
```

```
* Modified:                                                          *
*                                                                   *
*-------------------------------------------------------------------*

 Program-id. TPIIMSDP.

*=====================*
 Environment Division.
*=====================*

*==============*
 Data Division.
*==============*

 Working-storage Section.
*-------------------------------------------------------------------*
* Status messages                                                   *
*-------------------------------------------------------------------*
 01  partnumber-unknown             pic x(79)
     Value 'Part number is not in database'.
 01  staninfo-unknown               pic x(79)
     Value 'Only basic information is available for part number'.
 01  dli-unknown.
     05  filler                     pic x(11) Value 'DLI status='.
     05  status-dli                 pic x(2).
     05  filler                     pic x(10) Value ' Function='.
     05  status-function            pic x(4).
     05  filler                     pic x(9)  Value ' Segment='.
     05  status-segment             pic x(8).
     05  filler                     pic x(1)  Value space.
     05  status-message             pic x(34) Value space.
 01  ioerr-unknown.
     05  filler                     pic x(11) Value 'DLI status='.
     05  ioerr-dli                  pic x(2).
     05  filler                     pic x(10) Value ' Function='.
     05  ioerr-function             pic x(4).
     05  filler                     pic x(8)  Value ' Assist='.
     05  ioerr-status               pic x(6)  Value space.
     05  filler redefines ioerr-status.
         10  ioerr-char             pic x(2).
         10  filler                 pic x(4).
     05  ioerr-num redefines ioerr-status
                                    pic -99999.
     05  filler                     pic x(1)  Value space.
     05  ioerr-message              pic x(31) Value space.
 01  echo-data                      pic x(36) Value space.

*-------------------------------------------------------------------*
* Work variables                                                    *
*-------------------------------------------------------------------*
 01  dli-gu                         pic x(4) Value 'GU'.
 01  dli-isrt                       pic x(4) Value 'ISRT'.
 01  dli-gn                         pic x(4) Value 'GN'.
 01  dli-gnp                        pic x(4) Value 'GNP'.
 01  mod-name                       pic x(8) Value 'ABCO02'.

*-------------------------------------------------------------------*
* SSA's for PARTROOT and STANINFO segments                          *
```

```
*-----------------------------------------------------------------*
 01  partroot-ssa.
     05  filler                    pic x(8) Value 'PARTROOT'.
     05  filler                    pic x(11) Value '(PARTKEY  ='.
     05  filler                    pic x(2) Value '02'.
     05  partroot-key              pic x(15) Value Space.
     05  filler                    pic x(1) Value ')'.
 01  staninfo-ssa.
     05  filler                    pic x(8) Value 'STANINFO'.
     05  filler                    pic x(1) Value ' '.

*-----------------------------------------------------------------*
* PARTROOT segment IO area                                        *
*-----------------------------------------------------------------*
 01  partroot-segment.
     05  filler                    pic x(2).
     05  partroot-partno           pic x(15).
     05  filler                    pic x(9).
     05  partroot-descr            pic x(20).
     05  filler                    pic x(4).

*-----------------------------------------------------------------*
* STANINFO segment IO area                                        *
*-----------------------------------------------------------------*
 01  staninfo-segment.
     05  staninfo-proc-code        pic x(2).
     05  staninfo-inv-code         pic x(1).
     05  staninfo-rev-number       pic x(2).
     05  filler                    pic x(24).
     05  staninfo-makedept         pic x(2).
     05  staninfo-makecost         pic x(2).
     05  filler                    pic x(2).
     05  staninfo-commodity-code   pic x(4).
     05  filler                    pic x(4).
     05  filler                    pic x(25).

*-----------------------------------------------------------------*
* Terminal segment input/output area (MID and MOD)                *
*-----------------------------------------------------------------*
 01  buffer.
     05  buffer-ll                 pic 9(4) Binary.
     05  buffer-zz                 pic 9(4) Binary.
     05  input-buffer.
         10  input-trancode        pic x(8).
         10  input-partno          pic x(15).
         10  filler                pic x(102).
     05  input-buffer-segment2 redefines input-buffer.
         10  input-echo-data       pic x(36).
         10  filler                pic x(89).
     05  output-buffer redefines input-buffer.
         10  output-partno         pic x(15).
         10  output-descr          pic x(20).
         10  output-proc-code      pic x(2).
         10  output-inv-code       pic x(1).
         10  output-revision-nbr   pic x(2).
         10  output-makedept       pic x(2).
         10  output-makecctr       pic x(2).
         10  output-commodity      pic x(2).
```

```
           10  output-status          pic x(79).
       05  output-buffer-2 redefines input-buffer.
           10  output-echo-data       pic x(36).
           10  filler                 pic x(89).
 01  buffer-x.
       05  buffer-ll-x                pic 9(4) Binary.
       05  buffer-zz-x                pic 9(4) Binary.
       05  input-buffer-x.
           10  input-trancode-x       pic x(8).
           10  input-partno-x         pic x(15).

 Linkage section.
*---------------------------------------------------------------*
* Input-Output PCB layout                                       *
*---------------------------------------------------------------*
 01  iopcb.
       05  iopcb-lterm                pic x(8).
       05  iopcb-assist-status-bin    pic s9(4) comp.
       05  iopcb-assist-status-char redefines
           iopcb-assist-status-bin    pic x(2).
           88  iopcb-assist-aib-error   value 'EA'.
           88  iopcb-assist-buffer-full value 'EB'.
           88  iopcb-assist-tim-only    value 'EC'.
       05  iopcb-status               pic x(2).
           88  iopcb-dli-stop         value 'QC'.
           88  iopcb-dli-ok           value '  '.
           88  iopcb-assist-error     value 'ZZ'.
       05  iopcb-cdate                pic s9(7) comp-3.
       05  iopcb-ctime                pic s9(7) comp-3.
       05  iopcb-input-msgno          pic 9(8) binary.
       05  iopcb-output-mod           pic x(8).
       05  iopcb-userid               pic x(8).

 01  altpcb1.
       05  altpcb1-lterm              pic x(8).
       05  filler                     pic x(2).
       05  altpcb1-status             pic x(2).

 01  altpcb2.
       05  altpcb2-lterm              pic x(8).
       05  filler                     pic x(2).
       05  altpcb2-status             pic x(2).

*---------------------------------------------------------------*
* DI21PART PCB layout                                           *
*---------------------------------------------------------------*
 01  di21part-pcb.
       05  filler                     pic x(10).
       05  dbpcb-status               pic x(2).
           88  dbpcb-dli-ok           Value '  '.
           88  dbpcb-dli-not-found    Value 'GE'.
       05  filler                     pic x(8).
       05  dbpcb-segment-feedback     pic x(8).

*===============================================================*
 Procedure Division using iopcb, altpcb1, altpcb2, di21part-pcb.
*===============================================================*
```

```
*-----------------------------------------------------------------*
* Receive one input segment.                                      *
*-----------------------------------------------------------------*

 Get-unique.

     Display '****GU on IOPCB*********************************'.
     Display 'Doing new GU on the IO-PCB...'

     Call 'CBLADLI' using dli-gu
        iopcb
        buffer-x.

     Display 'Just after new GU:'.
     Display '  iopcb-status   = ' iopcb-status.

     If iopcb-dli-stop then
        Display 'Exiting after QC status code.'
        go to exit-now.

     Display '  buffer-ll        = ' buffer-ll-x.
     Display '  buffer-zz        = ' buffer-zz-x.
     Display '  input-trancode   = ' input-trancode-x.
     Display '  input-buffer     = ' buffer-x.
     Display '  lterm-name       = ' iopcb-lterm.
     Display '     user-id       = ' iopcb-userid.

     if not iopcb-dli-ok then
        move dli-gu to ioerr-function
        Perform io-error thru io-error-exit
        go to exit-now.

*-----------------------------------------------------------------*
* Origin of input may be determined by analyzing the             *
* buffer-zz field.  If it is zero, input has not been            *
* processed by MFS and originates from a socket client.          *
* If buffer-zz is 1,2 or 3 input has been processed by MFS       *
* and the value corresponds to the MFS option in effect.         *
*-----------------------------------------------------------------*

     If buffer-zz-x = 0 then
        Display 'Input originates from socket client'
     else
        Display 'Input originates from 3270 terminal'.
     Display 'iopcb-lterm    = ' iopcb-lterm.
     Display 'iopcb-userid   = ' iopcb-userid.

*-----------------------------------------------------------------*
* Look up info in PARTROOT                                        *
*-----------------------------------------------------------------*

     move input-partno-x to partroot-key.

     Display 'Just before GN on IO-PCB.'

     Call 'CBLADLI' using dli-gn
        iopcb
        buffer.
```

```
     Display 'Just after GN:'.
     Display '  iopcb-status   = ' iopcb-status.

     Display '  buffer-ll      = ' buffer-ll.
     Display '  buffer-zz      = ' buffer-zz.
     Display '  input-trancode = ' input-trancode.
     Display '  input-buffer   = ' buffer.

     Move input-echo-data to echo-data.
*
     move space to output-buffer.
     Call 'CBLADLI' using dli-gu
        di21part-pcb
        partroot-segment
        partroot-ssa.
     Display 'GU partroot status  = ' dbpcb-status.
     if dbpcb-dli-not-found then
        move partnumber-unknown to output-status
        go to isrt-output.
     if not dbpcb-dli-ok then
        move dli-gu to status-function
        perform db-error thru db-error-exit
        go to isrt-output.

*-----------------------------------------------------------------*
* Look up info in STANINFO                                        *
*-----------------------------------------------------------------*

     Call 'CBLADLI' using dli-gnp
        di21part-pcb
        staninfo-segment
        staninfo-ssa.
     Display 'GNP staninfo status = ' dbpcb-status.
     if dbpcb-dli-not-found then
        move partroot-partno to output-partno
        move partroot-descr to output-descr
        move staninfo-unknown to output-status
        go to isrt-output.
     if not dbpcb-dli-ok then
        move dli-gnp to status-function
        perform db-error thru db-error-exit
        go to isrt-output.

*-----------------------------------------------------------------*
* Build output segment                                            *
*-----------------------------------------------------------------*

     move partroot-partno to output-partno.
     move partroot-descr to output-descr.
     move staninfo-proc-code to output-proc-code.
     move staninfo-rev-number to output-revision-nbr.
     move staninfo-inv-code to output-inv-code.
     move staninfo-makedept to output-makedept.
     move staninfo-makecost to output-makecctr.
     move staninfo-commodity-code to output-commodity.
     move space to output-status.
```

```
*-------------------------------------------------------------*
* Send output segment                                         *
*-------------------------------------------------------------*

 isrt-output.
     move 150 to buffer-ll.
     move zero to buffer-zz.

     Display 'Just before ISRT of reply:'
     Display '  buffer-ll      = ' buffer-ll.
     Display '  buffer-zz      = ' buffer-zz.
     Display '  output buffer  = ' output-buffer.

     Call 'CBLADLI' using dli-isrt
        iopcb
        buffer
        mod-name.

     Display 'Just after ISRT of reply:'
     Display '  iopcb-status   = ' iopcb-status.

     if not iopcb-dli-ok then
        move dli-isrt to ioerr-function
        Perform io-error thru io-error-exit
        go to exit-now.
*
     Move space to output-buffer.
     move echo-data to output-echo-data.
     move 40 to buffer-ll.
     Display 'Just before ISRT of echo-segment:'.
     Display '  buffer-ll      = ' buffer-ll.
     Display '  buffer-zz      = ' buffer-zz.
     Display '  output buffer  = ' output-buffer.

     Call 'CBLADLI' using dli-isrt
        iopcb
        buffer.

     Display 'Just after ISRT of echo-segment:'.
     Display '  iopcb-status   = ' iopcb-status.

     if not iopcb-dli-ok then
        move dli-isrt to ioerr-function
        Perform io-error thru io-error-exit
        go to exit-now.
*
     go to get-unique.

*-------------------------------------------------------------*
* Handle bad DLI status from a DB call                        *
*-------------------------------------------------------------*

 db-error.
     move dbpcb-status to status-dli.
     move dbpcb-segment-feedback to status-segment.
     move 'DLI Call failed' to status-message.
     move dli-unknown to output-status.
 db-error-exit.
```

```
     exit.

*-------------------------------------------------------------*
* Handle bad DLI status from an IO call                       *
*-------------------------------------------------------------*

 io-error.
     move iopcb-status to ioerr-dli.
     move space to ioerr-status.
     If iopcb-assist-error then
        if iopcb-assist-status-bin < 0 then
           move iopcb-assist-status-bin to ioerr-num
        else
           move iopcb-assist-status-char to ioerr-char
        end-if
        move 'Socket error' to ioerr-message
     else
        move space to ioerr-char
        move 'IO PCB call failed' to ioerr-message
     end-if.
     Display ioerr-unknown.
 io-error-exit.
     exit.

*-------------------------------------------------------------*
* Terminate program                                           *
*-------------------------------------------------------------*
 exit-now.
     Goback.
```

TPIOTMAC — IMS OTMA Listener COBOL Client Program

This sample client program uses the server program that is described in “TPIIMSDP — Triple-Purpose IMS Server Program” on page 37.

```
 Identification Division.
*========================*
*-------------------------------------------------------------*
*                                                             *
* Name:        TPIOTMAC - Client to test transaction ESRVI via  *
*                         the IMS OTMA listener.              *
*                                                             *
* Function:    Sends message to server and receives reply.    *
*              The server program is started via the IMS OTMA *
*              Listener.  Transaction code is ESRVI           *
*                                                             *
*              All messages exchanged between client and server *
*              are preceeded by a 2 byte binary length field. *
*                                                             *
*              Client builds an IMS Request Message (IRM)     *
*              which is send to the IMS OTMA Listener.        *
*              Data segments are sent and echoed back from    *
*              the server.                                    *
*              Last data segment sent is an EOM message with  *
*              length field of 4.                             *
*              Client code includes logic to deal with        *
*              Request Status Messages (RSM), Request MOD     *
```

```
*              Messages (RMM), and Completed Status Messages    *
*              (CSM).                                           *
*                                                               *
* Interface:   Server host name is mvs18                        *
*              OTMA listener listens on port 3005               *
*                                                               *
* Logic:       1. Connect to IMS OTMA Listener                  *
*              2. Send IMS Request Message (IRM)                *
*              3. Send data to be echoed back incl. EOM seg.    *
*              4. Receive echoed data or/and RSM and RMM        *
*              6. Receive Transaction Completed Status          *
*                 Message                                       *
*              7. Close socket and terminate                    *
*                                                               *
* Returncode:  - none -                                         *
*                                                               *
* Written:     March 1997, at ITSO Raleigh                      *
*                                                               *
* Modified:                                                     *
*                                                               *
*---------------------------------------------------------------*

 Program-id. TPIOTMAC.

*======================*
 Environment Division.
*======================*

*===============*
 Data Division.
*===============*

 Working-storage Section.
*---------------------------------------------------------------*
* Socket interface function codes                               *
*---------------------------------------------------------------*
 01  soket-functions.
     05 soket-accept          pic x(16) value 'ACCEPT          '.
     05 soket-bind            pic x(16) value 'BIND            '.
     05 soket-close           pic x(16) value 'CLOSE           '.
     05 soket-connect         pic x(16) value 'CONNECT         '.
     05 soket-fcntl           pic x(16) value 'FCNTL           '.
     05 soket-getclientid     pic x(16) value 'GETCLIENTID     '.
     05 soket-getibmopt       pic x(16) value 'GETIBMOPT       '.
     05 soket-gethostbyaddr   pic x(16) value 'GETHOSTBYADDR   '.
     05 soket-gethostbyname   pic x(16) value 'GETHOSTBYNAME   '.
     05 soket-gethostid       pic x(16) value 'GETHOSTID       '.
     05 soket-gethostname     pic x(16) value 'GETHOSTNAME     '.
     05 soket-getpeername     pic x(16) value 'GETPEERNAME     '.
     05 soket-getsockname     pic x(16) value 'GETSOCKNAME     '.
     05 soket-getsockopt      pic x(16) value 'GETSOCKOPT      '.
     05 soket-givesocket      pic x(16) value 'GIVESOCKET      '.
     05 soket-initapi         pic x(16) value 'INITAPI         '.
     05 soket-ioctl           pic x(16) value 'IOCTL           '.
     05 soket-listen          pic x(16) value 'LISTEN          '.
     05 soket-read            pic x(16) value 'READ            '.
     05 soket-readv           pic x(16) value 'READV           '.
     05 soket-recv            pic x(16) value 'RECV            '.
```

```
     05 soket-recvfrom        pic x(16) value 'RECVFROM        '.
     05 soket-recvmsg         pic x(16) value 'RECVMSG         '.
     05 soket-select          pic x(16) value 'SELECT          '.
     05 soket-selectex        pic x(16) value 'SELECTEX        '.
     05 soket-send            pic x(16) value 'SEND            '.
     05 soket-sendmsg         pic x(16) value 'SENDMSG         '.
     05 soket-sendto          pic x(16) value 'SENDTO          '.
     05 soket-setsockopt      pic x(16) value 'SETSOCKOPT      '.
     05 soket-shutdown        pic x(16) value 'SHUTDOWN        '.
     05 soket-socket          pic x(16) value 'SOCKET          '.
     05 soket-takesocket      pic x(16) value 'TAKESOCKET      '.
     05 soket-termapi         pic x(16) value 'TERMAPI         '.
     05 soket-write           pic x(16) value 'WRITE           '.
     05 soket-writev          pic x(16) value 'WRITEV          '.
*---------------------------------------------------------------*
* Work variables                                                *
*---------------------------------------------------------------*
 01  errno                          pic 9(8) binary value zero.
 01  retcode                        pic s9(8) binary value zero.
 01  server-ipaddr-dotted           pic x(15) Value space.
 01  connect-status                 pic 9(4) Binary value zero.
     88  connect-done               value 1.
 01  close-status                   pic 9(4) Binary value zero.
     88  socket-was-closed          value 1.
*---------------------------------------------------------------*
* Variables used for the INITAPI call                           *
*---------------------------------------------------------------*
 01  maxsocc                        pic 9(4) Binary Value 2.
 01  apitype                        pic 9(4) Binary Value 2.
 01  initapi-ident.
     05  tcpname                    pic x(8) Value space.
     05  asname                     pic x(8) Value space.
 01  subtask                        pic x(8) value space.
 01  maxsno                         pic 9(8) Binary Value 1.
*---------------------------------------------------------------*
* Variables returned by the GETCLIENTID Call                    *
*---------------------------------------------------------------*
 01  clientid.
     05  clientid-domain            pic 9(8) Binary.
     05  clientid-name              pic x(8) value space.
     05  clientid-task              pic x(8) value space.
     05  filler                     pic x(20) value low-value.
*---------------------------------------------------------------*
* Variables used for the SOCKET call                            *
*---------------------------------------------------------------*
 01  afinet                         pic 9(8) Binary Value 2.
 01  soctype-stream                 pic 9(8) Binary Value 1.
 01  proto                          pic 9(8) Binary Value zero.
 01  socket-descriptor              pic 9(4) Binary Value zero.
*---------------------------------------------------------------*
* Variables used for the GETHOSTBYNAME Call                     *
*---------------------------------------------------------------*
 01  host-namelen                   pic 9(8) Binary Value 5.
 01  host-name                      pic x(7) Value 'mvs18'.
 01  host-entry-addr                pic x(4) Value low-value.
*---------------------------------------------------------------*
* Variables used for the call to EZACIC08                       *
*---------------------------------------------------------------*
```

```
 01  host-alias-seq                pic 9(4) Binary Value zero.
 01  host-addr-seq                 pic 9(4) Binary Value zero.
 01  host-name-length              pic 9(4) Binary Value zero.
 01  host-name-value               pic x(255) Value space.
 01  host-alias-count              pic 9(4) Binary Value zero.
 01  host-alias-length             pic 9(4) Binary Value zero.
 01  host-alias-value              pic x(255) Value space.
 01  host-addr-type                pic 9(4) Binary Value zero.
 01  host-addr-length              pic 9(4) Binary Value zero.
 01  host-addr-count               pic 9(4) Binary Value zero.
 01  host-addr-value               pic x(4) Value low-value.
 01  host-return-code              pic s9(8) Binary Value zero.
*---------------------------------------------------------------*
* Variables used for the CONNECT Call                           *
*---------------------------------------------------------------*
 01  server-socket-address.
     05  server-afinet             pic 9(4) Binary Value 2.
     05  server-port               pic 9(4) Binary Value 3005.
     05  server-ipaddr             pic x(4) Value low-value.
     05  filler                    pic x(8) value low-value.
*---------------------------------------------------------------*
* IMS Request Message segment  (IRM)                            *
*---------------------------------------------------------------*
 01  IRM-message.
     05  IRM-len                   pic 9(4) Binary Value 68.
     05  filler                    pic x(2) Value low-value.
     05  IRM-id                    pic x(8) Value '*IRMREQ*'.
     05  IRM-trancode              pic x(8) Value 'ESRVI'.
     05  IRM-dest                  pic x(8) Value 'DST18'.
     05  IRM-lterm                 pic x(8) Value 'SOCK001'.
     05  IRM-flags                 pic x(1) Value X'80'.
         88  TRM-MFSReq            Value X'80'.
     05  filler                    pic x(3) Value low-value.
     05  IRM-user-data.
         10  IRM-userid            pic x(8) Value 'ALFRED'.
         10  IRM-password          pic x(8) Value 'XXXXXXX'.
         10  IRM-new-password      pic x(8) Value space.
         10  IRM-groupid           pic x(8) Value space.
*---------------------------------------------------------------*
* Data segments to IMS server                                   *
*---------------------------------------------------------------*
 01  request-message-seg1.
     05  filler                    pic 9(4) Binary value 24.
     05  filler                    pic xx value low-value.
     05  filler                    pic x(20)
                                   Value 'Data segment 1'.
 01  request-message-seg2.
     05  filler                    pic 9(4) Binary value 24.
     05  filler                    pic xx value low-value.
     05  filler                    pic x(20)
                                   Value 'Data segment 2'.
 01  EOM-message.
     05  filler                    pic 9(4) Binary value 4.
     05  filler                    pic xx value low-value.
*---------------------------------------------------------------*
* Peek buffer and length fields for RECV peek call.             *
*---------------------------------------------------------------*
 01  recv-flag-read                pic 9(8) Binary value zero.
```

```
 01  recv-flag-peek                pic 9(8) Binary value 2.
 01  recv-flag                     pic 9(8) Binary value zero.
 01  recv-peek-len                 pic 9(8) Binary value 2.
 01  recv-peek-len-to-read         pic 9(4) Binary value zero.
*---------------------------------------------------------------*
* Buffers and length fields for read and write                  *
*---------------------------------------------------------------*
 01  read-request-len              pic 9(8) Binary Value zero.
 01  read-request-read             pic 9(8) Binary Value zero.
 01  read-request-remaining        pic 9(8) Binary Value zero.
 01  send-request-len              pic 9(8) Binary Value zero.
 01  send-request-sent             pic 9(8) Binary value zero.
 01  send-request-remaining        pic 9(8) Binary value zero.
 01  read-buffer.
     05  buffer-total              pic x(4096) Value space.
     05  RSM-message redefines buffer-total.
         10  filler                pic x(4).
         10  RSM-id                pic x(8).
             88  RSM-received      value '*REQSTS*'.
         10  RSM-return-code       pic 9(8) Binary.
             88  RSM-OK            value zero.
         10  RSM-reason-code       pic 9(8) Binary.
         10  filler                pic x(4076).
     05  CSM-message redefines buffer-total.
         10  filler                pic x(4).
         10  CSM-id                pic x(8).
             88  CSM-received      value '*CSMOKY*'.
         10  filler                pic x(4084).
     05  RMM-message redefines buffer-total.
         10  filler                pic x(4).
         10  RMM-id                pic x(8).
             88  RMM-received      Value '*REQMOD*'.
         10  RMM-MODname           pic x(8).
         10  filler                pic x(4076).
     05  read-buffer-bytes redefines buffer-total.
         10  read-buffer-byte      pic x occurs 4096 times.
     05  data-message redefines buffer-total.
         10  returned-len          pic 9(4) Binary.
         10  fille                 pic x(2).
         10  returned-message      pic x(40).
         10  filler                pic x(4052).
 01  send-buffer.
     05  send-buffer-total         pic x(8192) value space.
     05  send-buffer-seq redefines send-buffer-total
                                   pic x(8) occurs 1024 times.
     05  send-buffer-byte redefines send-buffer-total
                                   pic x occurs 8192 times.
*---------------------------------------------------------------*
* Error message for socket interface errors                     *
*---------------------------------------------------------------*
 01  ezaerror-msg.
     05  filler                    pic x(9) Value 'Function='.
     05  ezaerror-function         pic x(16) Value space.
     05  filler                    pic x value ' '.
     05  filler                    pic x(8) Value 'Retcode='.
     05  ezaerror-retcode          pic ---99.
     05  filler                    pic x value ' '.
     05  filler                    pic x(9) Value 'Errorno='.
```

```
     05  ezaerror-errno            pic zzz99.
     05  filler                    pic x value ' '.
     05  ezaerror-text             pic x(50) value ' '.

 Linkage Section.


*==================*
 Procedure Division.
*==================*

*-----------------------------------------------------------------*
* Initialize socket API                                           *
*-----------------------------------------------------------------*

     Move soket-initapi to ezaerror-function.
     Call 'TPICLNID' using asname subtask.
     Call 'EZASOKET' using soket-initapi
        maxsocc
        initapi-ident
        subtask
        maxsno
        errno
        retcode.
     If retcode < 0 then
        move 'Initapi failed' to ezaerror-text
        perform write-ezaerror-msg thru write-ezaerror-msg-exit
        go to exit-now.

*-----------------------------------------------------------------*
* Let us see the client-id                                        *
*-----------------------------------------------------------------*

     move soket-getclientid to ezaerror-function.
     Call 'EZASOKET' using soket-getclientid
        clientid
        errno
        retcode.
     If retcode < 0 then
        move 'Getclientid failed' to ezaerror-text
        perform write-ezaerror-msg thru write-ezaerror-msg-exit
        go to exit-term-api.
     Display 'Client ID = ' clientid-name ' ' clientid-task.

*-----------------------------------------------------------------*
* Get us a socket descriptor                                      *
*-----------------------------------------------------------------*

     move soket-socket to ezaerror-function.
     Call 'EZASOKET' using soket-socket
        afinet
        soctype-stream
        proto
        errno
        retcode.
     If retcode < 0 then
        move 'Socket call failed' to ezaerror-text
        perform write-ezaerror-msg thru write-ezaerror-msg-exit
```

```
           go to exit-term-api.
        Move retcode to socket-descriptor.

      *-------------------------------------------------------------*
      * Get host entry structure pointer based on host name         *
      *-------------------------------------------------------------*

        move soket-gethostbyname to ezaerror-function.
        Call 'EZASOKET' using soket-gethostbyname
           host-namelen
           host-name
           host-entry-addr
           retcode.
        If retcode < 0 then
           move 'Gethostbyaddr failed' to ezaerror-text
           perform write-ezaerror-msg thru write-ezaerror-msg-exit
           go to exit-close-socket.

      *-------------------------------------------------------------*
      * Get info out of the HOSTENT structure.                      *
      * Loop until either returned list of IP addresses is          *
      * exhausted or connect is successfull.                        *
      *-------------------------------------------------------------*

        Move zero to connect-status.
        Perform until ((host-addr-count = host-addr-seq and
           host-addr-seq > 0) or
           connect-done)
           If host-alias-seq > host-alias-count then
              subtract 1 from host-alias-seq
           end-if
           move 'EZACIC08' to ezaerror-function
           Call 'EZACIC08' using host-entry-addr
              host-name-length
              host-name-value
              host-alias-count
              host-alias-seq
              host-alias-length
              host-alias-value
              host-addr-type
              host-addr-length
              host-addr-count
              host-addr-seq
              host-addr-value
              host-return-code
           If host-return-code < 0 then
              move host-return-code to retcode
              move 'Host translation failed' to ezaerror-text
              perform write-ezaerror-msg thru
                 write-ezaerror-msg-exit
              go to exit-close-socket
           end-if

           Move host-addr-value to server-ipaddr

      *-------------------------------------------------------------*
      * Try to connect to IMS OTMA Listener on returned IP address  *
      *-------------------------------------------------------------*
```

```
         If host-return-code = 0 then
            Move soket-connect to ezaerror-function
            Call 'TPIINTOA' using server-ipaddr
               server-ipaddr-dotted
            Display 'Trying to connect to server: '
               server-ipaddr-dotted
            Call 'EZASOKET' using soket-connect
               socket-descriptor
               server-socket-address
               errno
               retcode
            If retcode < 0 then
               Move space to ezaerror-text
               Call 'TPIINTOA' using server-ipaddr
                  ezaerror-text
               perform write-ezaerror-msg thru
                  write-ezaerror-msg-exit
            else
               move 1 to connect-status
            end-if
         end-if
      end-perform.

      if retcode < 0 then
         move 'Connect failed' to ezaerror-text
         perform write-ezaerror-msg thru
            write-ezaerror-msg-exit
         Go to exit-close-socket.

      Display 'Connected to server at ' server-ipaddr-dotted.

*-----------------------------------------------------------------*
* Send IRM to IMS OTMA listener                                   *
*-----------------------------------------------------------------*

      Display 'IRM to IMS Listener: ' IRM-message
      move IRM-message to send-buffer
      move IRM-len to send-request-len
      perform write-tcp thru write-tcp-exit
      If send-request-sent < 0 then
         Display 'Write of TRM failed'
         go to exit-close-socket
      end-if

*-----------------------------------------------------------------*
* Send 2 data segments and an EOM segment to server               *
*-----------------------------------------------------------------*

      Display 'Sending data segment 1'.
      move request-message-seg1 to send-buffer.
      move 24 to send-request-len.
      perform write-tcp thru write-tcp-exit.
      If send-request-sent < 0 then
         Display 'Write of segment1 failed'
         go to exit-close-socket.
      Display 'Sending data segment 2'.
      move request-message-seg2 to send-buffer.
```

```
     move 24 to send-request-len.
     perform write-tcp thru write-tcp-exit.
     If send-request-sent < 0 then
        Display 'Write of segment2 failed'
        go to exit-close-socket.
     Display 'Sending EOM segment'.
     move EOM-message to send-buffer.
     move 4 to send-request-len.
     perform write-tcp thru write-tcp-exit.
     If send-request-sent < 0 then
        Display 'Write of EOM failed'
        go to exit-close-socket.

*-------------------------------------------------------------*
* Read segments from server                                   *
*-------------------------------------------------------------*

     Display 'Preparing to read response from IMS'.

     Perform until (CSM-received or
        (RSM-received and not RSM-OK) or
        socket-was-closed)
        Move recv-flag-peek to recv-flag
        Move 2 to read-request-len
        Perform read-tcp thru read-tcp-exit
        If read-request-read < 0 then
           Display 'Peek for length failed'
           go to exit-close-socket
        end-if

        move returned-len to read-request-len
        move recv-flag-read to recv-flag
        move space to read-buffer
        perform read-tcp thru read-tcp-exit
        if read-request-read < 0 then
           Display 'Read failed'
           go to exit-close-socket
        end-if

        Display 'Received segment = ' returned-message

        If RMM-received then
           Display 'RMM id=' RMM-id ' Mod-name=' RMM-MODname
        end-if

        If RSM-received then
           Display 'RSM id=' RSM-id ' Return-code='
              RSM-return-code
              ' Reason-code=' RSM-reason-code
           If not RSM-OK Then
              Display 'Transaction not scheduled.'
           end-if
        end-if

        If CSM-received then
           Display 'CSM id=' CSM-id
        end-if
```

```
     end-perform.

 exit-close-socket.
*-----------------------------------------------------------------*
* Close socket                                                    *
*-----------------------------------------------------------------*

     move soket-close to ezaerror-function
     Call 'EZASOKET' using soket-close
        socket-descriptor
        errno
        retcode.
     If retcode < 0 then
        move 'Close call failed' to ezaerror-text
        perform write-ezaerror-msg thru write-ezaerror-msg-exit.

 exit-term-api.
*-----------------------------------------------------------------*
* Terminate socket API                                            *
*-----------------------------------------------------------------*
     Call 'EZASOKET' using soket-termapi.

 exit-now.
*-----------------------------------------------------------------*
* Terminate program                                               *
*-----------------------------------------------------------------*
     Move zero to return-code.
     Goback.

 write-ezaerror-msg.
*-----------------------------------------------------------------*
* Subroutine                                                      *
* ----------                                                      *
*                                                                 *
* Write out an error message                                      *
*-----------------------------------------------------------------*
     move errno to ezaerror-errno.
     move retcode to ezaerror-retcode.
     display ezaerror-msg.
 write-ezaerror-msg-exit.
     exit.

*-----------------------------------------------------------------*
* Subroutine                                                      *
* ----------                                                      *
*                                                                 *
* Read data from a TCP socket                                     *
*                                                                 *
* Read-request-len tells how many bytes to read.                  *
*-----------------------------------------------------------------*

 Read-TCP.

     move soket-recv to ezaerror-function.
     move zero to read-request-read.
     move read-request-len to read-request-remaining.
     Perform until read-request-remaining = 0
        Call 'EZASOKET' using soket-recv
```

```
            socket-descriptor
            recv-flag
            read-request-remaining
            read-buffer-byte(read-request-read + 1)
            errno
            retcode
         If retcode < 0 then
            move 'Read call failed' to ezaerror-text
            perform write-ezaerror-msg thru
               write-ezaerror-msg-exit
            move -1 to read-request-read
            move zero to read-request-remaining
         else
            Display 'Number of bytes read=' retcode
            Add retcode to read-request-read
            Subtract retcode from read-request-remaining
            If retcode = 0 then
               Display 'Server probably closed socket'
               Move zero to read-request-remaining
               Move 1 to close-status
            end-if
         end-if
      end-perform.

 Read-TCP-exit.
      exit.

*---------------------------------------------------------------*
* Subroutine                                                    *
* ----------                                                    *
*                                                               *
* Send data over a TCP connection                               *
*                                                               *
* Send-request-len tells how many bytes to write.               *
*---------------------------------------------------------------*
 Write-TCP.

      move soket-write to ezaerror-function.
      move send-request-len to send-request-remaining.
      move 0 to send-request-sent.
      Perform until send-request-remaining = 0
         Display 'Writing so many bytes: ' send-request-remaining
         Call 'EZASOKET' using soket-write
            socket-descriptor
            send-request-remaining
            send-buffer-byte(send-request-sent + 1)
            errno
            retcode
         If retcode < 0 then
            move 'Write call failed' to ezaerror-text
            perform write-ezaerror-msg thru
               write-ezaerror-msg-exit
            move -1 to send-request-sent
            move zero to send-request-remaining
         else
            add retcode to send-request-sent
            subtract retcode from send-request-remaining
         end-if
```

```
            end-perform.

        Write-TCP-exit.
            exit.
```

The following output is trace output from a test execution of TPIOTMAC:

```
Client ID = ALFREDCC 006E1B88
Trying to connect to server: 9.24.104.74
Connected to server at 9.24.104.74
IRM to IMS Listener:     *IRMREQ*ESRVI   DST18   SOCK001     ALFRED  XXXXXXXX
Writing so many bytes: 0000000068
Sending data segment 1
Writing so many bytes: 0000000024
Sending data segment 2
Writing so many bytes: 0000000024
Sending EOM segment
Writing so many bytes: 0000000004
Preparing to read response from IMS
Number of bytes read=0000000002
Number of bytes read=0000000020
Received segment = *REQMOD*ABCO02
RMM id=*REQMOD* Mod-name=ABCO02
Number of bytes read=0000000002
Number of bytes read=0000000032
Received segment = ESRVI   Data segment 1
Number of bytes read=0000000002
Number of bytes read=0000000024
Received segment = Data segment 2
Number of bytes read=0000000002
Number of bytes read=0000000012
Received segment = *CSMOKY*
CSM id=*CSMOKY*
```

TPICPART — IMS OTMA Listener C Client Program

This C program is used to test the IMS server program decsribed in “TPIIMSDP — Triple-Purpose IMS Server Program” on page 37.

```
/*-------------------------------------------------------------*
*                                                             *
* Name:        TPICPART - Client to test transaction TPIPART  *
*                         started via IMS OTMA listener.      *
*                                                             *
* Function:    Sends message to server and receives reply.    *
*              The server program is started via the IMS OTMA *
*              Listener.  Transaction code is TPIPART.        *
*                                                             *
*              All messages exchanged between client and server *
*              are preceeded by a 2 byte binary length field. *
*                                                             *
*              Client builds an IMS Request Message (IRM)     *
*              which is send to the IMS OTMA Listener.        *
*              Data segments are sent and echoed back from    *
*              the server.                                    *
*              Last data segment sent is an EOM message with  *
*              length field of 4.                             *
*              Client code includes logic to deal with        *
```

```
*              Request Status Messages (RSM), Request MOD      *
*              Messages (RMM), and Completed Status Messages   *
*              (CSM).                                          *
*                                                              *
* Interface:   Server host name, port number and part number   *
*              to query are passed as run-time arguments to    *
*              this program.                                   *
*                                                              *
* Logic:       1. Connect to IMS OTMA Listener                 *
*              2. Send IMS Request Message (IRM)               *
*              3. Send segment with part number to query       *
*              4. Send data to be echoed back and EOM segment  *
*              5. Receive reply to part database query         *
*              6. Receive echoed data                          *
*              7. Receive Transaction Completed Status         *
*                 Message                                      *
*              8. Close socket and terminate                   *
*                                                              *
* Returncode:  - none -                                        *
*                                                              *
* Written:     March 1997, at ITSO Raleigh                     *
*                                                              *
* Modified:                                                    *
*                                                              *
*--------------------------------------------------------------*/
/*
 * Include Files.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>
#include <socket.h>
#include <netdb.h>
/*
 * Client Main.
 */
main(int argc, char**argv)
{
    /*
     *  Transaction Request Message (IRM)
     *  Sent by us to the IMS listener to
     *  initiate IMS transaction
     */
    struct IRM_message {
       unsigned short ll;
       unsigned short zz;
       char           irmreq [8];
       char           trancode [8];
       char           datastore [8];
       char           lterm [8];
       unsigned long  flag;
       char           ims_userid [8];
       char           ims_password [8];
    } IRM;
    /*
```

```
 *  Request Status Message (RSM)
 *  May be sent by the IMS Listener
 */
struct RSM_message {
   unsigned short ll;
   unsigned short zz;
   char           id [8];
   unsigned long  rc;
   unsigned long  reason;
} RSM;
/*
 *  Request MOD Message (RMM)
 *  May be sent by the IMS Listener if
 *  application uses MOD name on ISRT calls.
 */
struct RMM_message {
   unsigned short ll;
   unsigned short zz;
   char           rmm_id [8];
   char           Mod [8];
} RMM;
/*
 *  Completed Status Message (CSM)
 *  Sent by the OTMA listener, when
 *  transaction has completed.
 */
struct CSM_message {
   unsigned short ll;
   unsigned short zz;
   char           csmoky [8];
} CSM;
/*
 *  Segment buffer for receiving data
 */
struct segment_buffer {
   unsigned short ll;
   unsigned short zz;
   char           buf [200];
} segment;
/*
 *  Segment buffer for input segment to IMS
 */
struct input_segment_buffer {
   unsigned short ll;
   unsigned short zz;
   char           partno [15];
} input_segment;
/*
 *  Segment buffer for input segment number two to IMS
 */
struct input_segment_buffer_2 {
   unsigned short ll;
   unsigned short zz;
   char           echo_data [36];
} input_segment_2;
/*
 *  Segment buffer for parts database query reply segment
 */
```

```
struct output_segment_buffer {
   unsigned short ll;
   unsigned short zz;
   char           partno [15];
   char           descr  [20];
   char           proccode [2];
   char           invcode [1];
   char           revnbr [2];
   char           makedept [2];
   char           makecctr [2];
   char           commodity [2];
   char           status [79];
} output_segment;
/*
 *  Various work fields
 */
unsigned short port;       /* port client to connect to           */
char buf [200];  /* send receive buffer                 */
char part [15];  /* part number from PARM field         */
unsigned short lenbytes;   /* Length field                        */
struct hostent *hostnm;    /* server host name information        */
struct sockaddr_in server; /* server address                      */
int s;                     /* client socket                       */
struct clientid ourclientid;  /* Client ID structure              */
/*
 * Check Arguments Passed. Should be hostname and port and key
 */
if (argc != 4) {
    printf("Usage: %s hostname port part-key\n", argv[0]);
    exit(1);
}
/*
 * The host name is the first argument. Get the server address.
 */
hostnm = gethostbyname(argv[1]);
if (hostnm == (struct hostent *) 0) {
    printf("Gethostbyname failed\n");
    exit(2);
}
/*
 * The port is the second argument.
 */
port = (unsigned short) atoi(argv[2]);
/*
 * The part number is the third argument
 */
printf("Part number from PARM field is: %s\n", argv[3]);
strcpy(part,argv[3]);
/*
 * Build the IRM
 */
IRM.ll = htons(56);
IRM.zz = 0;
strcpy(IRM.irmreq,  "*IRMREQ*");
strcpy(IRM.trancode,"TPIPART ");
strcpy(IRM.datastore,"DST18   ");
strcpy(IRM.lterm,"SOCK002 ");
IRM.flag = 0x80000000;
```

```
strcpy(IRM.ims_userid,"ALFRED  ");
strcpy(IRM.ims_password,"NYGAARD ");
/*
 * 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);
}
printf("Socket sd       = %d\n", s);
/*
 * Let us see our Client ID
 */
if (getclientid(AF_INET, &ourclientid) < 0) {
    tcperror("Getclientid()");
    exit(4);
}
printf("ClientID Jobname     = %.*s\n",
        sizeof(ourclientid.name), ourclientid.name);
printf("ClientID Subtaskname = %.*s\n",
        sizeof(ourclientid.subtaskname), ourclientid.subtaskname);
/*
 * Connect to the server and send IRM
 */
if (connect(s, (struct sockaddr*) &server, sizeof(server)) < 0) {
    tcperror("Connect()");
    exit(4);
}
printf("Connected\n");
if (send(s, (char*) &IRM, sizeof(IRM), 0) < 0) {
    tcperror("Send()");
    exit(5);
}
printf("Send of IRM complete\n");
printf("IRM trancode    = %.8s\n", IRM.trancode);
/*
 * Build transaction input_segments and send them
 */
input_segment.ll = htons(sizeof(input_segment));
input_segment.zz = 0;
memcpy(input_segment.partno, "AN960C10       ", 15);
printf("input_segment.ll=%d - partno=%.15s\n",
        input_segment.ll, input_segment.partno);
if (send(s, (char*) &input_segment,
    sizeof(input_segment), 0) < 0) {
    tcperror("Send() of input_segment");
    exit(7);
}
input_segment_2.ll = 40;
input_segment_2.zz = 0;
memcpy(input_segment_2.echo_data, "Some data to echo", 36);
```

```
if (send(s, (char*) &input_segment_2,
    sizeof(input_segment_2), 0) < 0) {
    tcperror("Send() of input_segment_2");
    exit(7);
}
printf("input_segment_2.ll=%d - sent\n", input_segment_2.ll);
printf("Send data complete\n");
/*
 * Send End of Message segment
 */
segment.ll = htons(4);
segment.zz = 0;

if (send(s, (char*) &segment, 4, 0) < 0) {
    tcperror("Send()");
    exit(7);
}
printf("EOM segment sent\n");
/*
 * Receive first segment into buffer
 */
if (recv(s, (char*) &segment, 4, MSG_PEEK) < 0) {
    tcperror("Recv() Peek for 4 bytes");
    exit(6);
}
lenbytes = ntohs(segment.ll);
printf("Bytes ready to read is %d\n", lenbytes);
if (recv(s, (char*)&segment, lenbytes, 0) < 0) {
    tcperror("Recv()");
    exit(6);
}
/*
 * Check for an RSM segment
 */
if (!memcmp(segment.buf, "*REQSTS*", 8)) {
    memcpy(&RSM, &segment, ntohs( segment.ll ));
    printf("Receive of RSM complete\n");
    RSM.rc = ntohl(RSM.rc);
    RSM.reason = ntohl(RSM.reason);
    printf("RSM rc          = %d\n", RSM.rc);
    printf("RSM reason code = %d\n", RSM.reason);
    if (RSM.rc > 0) {
       printf("Negative response in RSM message - rc=%d\n", RSM.rc);
       exit(12);
    }
    if (recv(s, (char*) &segment, 4, MSG_PEEK) < 0) {
        tcperror("Recv() Peek for 4 bytes");
        exit(6);
    }
    lenbytes = ntohs( segment.ll );
    printf("Bytes ready to read is %d\n", lenbytes);
    if (recv(s, (char*) &segment, lenbytes, 0) < 0) {
        tcperror("Recv()");
        exit(6);
    }
}
/*
 * Check for an RMM segment
```

```
 */
if (!memcmp(segment.buf, "*REQMOD*", 8)) {
    memcpy(&RMM, &segment, ntohs( segment.ll ));
    printf("Receive of RMM complete\n");
    printf("MOD name is     = %s\n", RMM.Mod);
    if (recv(s, (char*) &segment, 4, MSG_PEEK) < 0) {
        tcperror("Recv() Peek for 4 bytes");
        exit(6);
    }
    lenbytes = ntohs( segment.ll );
    printf("Bytes ready to read is %d\n", lenbytes);
    if (recv(s, (char*) &segment, lenbytes, 0) < 0) {
        tcperror("Recv()");
        exit(6);
    }
}
printf("Full output segment is %s\n", segment.buf);
memcpy(&output_segment, &segment, ntohs( segment.ll ));
printf("Output data received\n");
printf("Partno    = %15.15s\n", output_segment.partno);
printf("Descr     = %20.20s\n", output_segment.descr);
printf("Proccode  = %2.2s\n", output_segment.proccode);
printf("Invcode   = %1.1s\n", output_segment.invcode);
printf("Revnbr    = %2.2s\n", output_segment.revnbr);
printf("Makedept  = %2.2s\n", output_segment.makedept);
printf("Makecctr  = %2.2s\n", output_segment.makecctr);
printf("Commodity = %2.2s\n", output_segment.commodity);
printf("Status    = %79.79s\n", output_segment.status);
/*
 * Receive echo data segment into buffer
 */
if (recv(s, (char*) &segment, 4, MSG_PEEK) < 0) {
    tcperror("Recv() Peek for 4 bytes");
    exit(6);
}
lenbytes = ntohs(segment.ll);
printf("Bytes ready to read is %d\n", lenbytes);
if (recv(s, (char*)&segment, lenbytes, 0) < 0) {
    tcperror("Recv()");
    exit(6);
}
printf("Full echo segment is %s\n", segment.buf);
/*
 * Receive CSM message
 */
if (recv(s, (char*) &CSM, sizeof(CSM), 0) < 0) {
    tcperror("Recv()");
    exit(6);
}
printf("recv returned %s\n", CSM.csmoky);

if (!memcmp(CSM.csmoky, "*CSMOKY*", 8)) {
    printf("Receive of CSM complete: %s\n", CSM.csmoky);
}
/*
 * Close the socket.
 */
close(s);
```

```
    printf("Client Ended Successfully\n");
    exit(0);
}
```

The following is trace output from a sample execution of TPICPART:

```
Part number from PARM field is: AN960C10
Socket sd       = 3
ClientID Jobname     = ALFREDC1
ClientID Subtaskname = 0b413000
Connected
Send of IRM complete
IRM trancode    = TPIPART
input_segment.ll=20 - partno=AN960C10
input_segment_2.ll=40 - sent
Send data complete
EOM segment sent
Bytes ready to read is 20
Receive of RMM complete
MOD name is     = ABCO02
Bytes ready to read is 150
Full output segment is AN960C10       WASHER              02

Output data received
Partno    = AN960C10
Descr     = WASHER
Proccode  = 02
Invcode   =
Revnbr    =
Makedept  =
Makecctr  =
Commodity =
Status    =
Bytes ready to read is 40
Full echo segment is Some data to echo
recv returned *CSMOKY*
Receive of CSM complete: *CSMOKY*
Client Ended Successfully
```

IMSLSECX — IMS BMP and OTMA Listener Security Exit

The following is a sample security exit that works with both the IMS Listener and the IMS OTMA Connection server.

```
*************************************************************************
*                                                                       *
*  Name:         IMSLSECX - IMS Sockets Listener security exit.        *
*                           Used by both the IMS BMP listener and      *
*                           the IMS OTMA listener.                     *
*                                                                       *
*  Function:     Validates stream socket connections to IMS.           *
*                                                                       *
*  Interface:    R1 -> parameter list with eight pointers:             *
*                 +0  ->  Fullword IP Address (In)                      *
*                 +4  ->  Halfword port number (In)                     *
*                 +8  ->  8 byte IMS tranaction code name (In)          *
*                 +12 ->  Halfword datatype (0, ASCII, 1 EBCDIC) (In)   *
*                 +16 ->  Fullword length of user data in TRM (In)      *
```

```
*                +20 ->  User data (In)                              *
*                +24 ->  Fullword return code (Out)                  *
*                +28 ->  Fullword reason code (Out)                  *
*                +32 ->  8 byte return user ID (Out)  - only OTMA    *
*                +36 ->  8 byte return group ID (out) - only OTMA    *
*                                                                    *
*               Security exit interface contains user data.  User    *
*               data is installation defined - in our case as 32     *
*               bytes with the following layout:                     *
*                                                                    *
*               8 bytes user ID                                      *
*               8 bytes password                                     *
*               8 bytes new password (optional)                      *
*               8 bytes RACF group ID (optinal)                      *
*                                                                    *
*  Logic:       1. Validates if all required parms are present.      *
*               2. Calls TPIRACF for user authentication and build   *
*                  of task level security environment.               *
*                  RACROUTE VERIFY passes an applid.  If caller is   *
*                  BMP Listener, it will be BMPLSTN - if caller is   *
*                  OTMA listener, it will be OTMALSTN.               *
*               3. Authorizes user's access to requested IMS tran    *
*                  code via call to TPIAUTH for resource class       *
*                  FACILITY and resource TPI.IMSSOCK.trancode        *
*               4. Deletes user security environment again and       *
*                  returns to IMS listener                           *
*                                                                    *
*  Abends:      User abend 1001: If RACROUTE REQUEST=DELETE fails,   *
*                                and we do not know if we continue   *
*                                under a user security environment,  *
*                                we abend.                           *
*               User abend 1002: Caller is neither BMP listener nor  *
*                                OTMA listener.                      *
*                                                                    *
*  Return codes: Return and reason codes set in the IMS listener     *
*               security exit interface area.                        *
*               RC=000 Reason=000: User authenticated OK and users   *
*                                  access to tran code authorized OK.*
*               RC=008 Reason=101: UserID and password missing in TRM*
*               RC=008 Reason=102: Invalid length of userdata in TRM *
*               RC=008 Reason=103: UserID not defined to RACF        *
*               RC=008 Reason=104: Invalid password                  *
*               RC=008 Reason=105: Password has expired              *
*               RC=008 Reason=106: New password is not valid         *
*               RC=008 Reason=107: User does not belong to group     *
*               RC=008 Reason=108: User is revoked                   *
*               RC=008 Reason=109: Access to group is revoked        *
*               RC=008 Reason=110: User not authorized to IMS Sockets*
*               RC=008 Reason=111: TPIRACF internal error            *
*               RC=008 Reason=112: TPIRACF internal error            *
*               RC=008 Reason=113: User not authorized to tran code  *
*                                                                    *
*                                                                    *
*  Written:     ITSO, Raleigh April 16, 1995                         *
*                                                                    *
*  Modified:    ITSO, Raliegh March 18, 1997 -                       *
*                     added support for OTMA Listener interface      *
*                                                                    *
```

```
*********************************************************************
*
WORKAREA DSECT                    *Program re-entrant work area
         DC    18F'0'             *Save area
STATBYTE DC    X'00'              *Status bits
OTMACALL EQU   BIT0               *Caller is OTMA
BMPCALL  EQU   BIT1               *Caller is BMP
         DC    XL3'00'            *Nice allignment
REQCODE  DC    A(0)               *TPIRACF Request Code
REQVER   EQU   0                  *REQUEST=VERIFY
REQDEL   EQU   8                  *REQUEST=DELETE
USERID   DC    CL8' '             *User ID
PWD      DC    CL8' '             *Password
NPWD     DC    CL8' '             *New password
GROUP    DC    CL8' '             *Group ID
APPLNAME DC    CL8'IMSLSTN'       *Application name (IMSLSTN)
AUTHRC   DC    A(0)               *Saved RC from TPIAUTH
RESNAME  DC    C'TPI.IMSSOCK.'    *Resource TPI.IMSSOCK.trancode
RESTRNNM DC    CL8' '             *Trancode
         DC    CL(80-(*-RESNAME))' '
DORD     DC    D'0'               *Work area
MACWORK  DC    256X'00'           *Macro work area
*
INTFAREA DSECT                    *Interface parm list mappings
LSIPADDR DC    A(0)               *-> Client IP address
LSPORT   DC    A(0)               *-> Client port number
LSTRNNAM DC    A(0)               *-> IMS transaction code name
LSDATTYP DC    A(0)               *-> Datatype (0 ASCII, 1 EBCDIC)
LSDATLEN DC    A(0)               *-> Length of user data in TRM
LSUSRDAT DC    A(0)               *-> User data area
LSRETCOD DC    A(0)               *-> Return code field
LSREACOD DC    A(0)               *-> Reason code field
LSOTMAU  DC    A(0)               *-> OTMA return user ID  <----+
LSOTMAG  DC    A(0)               *-> OTMA return group ID <----+
*                                                               !
*                                                    Only present if
*                                               caller is OTMA Listener
*
USERDATA DSECT                    *User data area in TRM or IRM
LSUSERID DC    CL8' '             *User ID
LSPWD    DC    CL8' '             *Password
LSNPWD   DC    CL8' '             *New Password (Optional)
LSGROUP  DC    CL8' '             *Group ID (Optional)
*
IMSLSECX INIT  'IMS Sockets Listener security exit',MODE=31,          C
               RENT=Y,WORKLEN=512
*
         USING WORKAREA,R13       *Base our reentrant work area
         LR    R11,R1             *Parameter pointer
         WTO   'IMSLSECX - Entered'
         USING INTFAREA,R11       *Adressability of parameters
*
* -------------------------------------------------------------------
*
* Initialize fields in work area.
*
* -------------------------------------------------------------------
*
```

```
         MVC   USERID,=CL8' '      *Initialize values to space
         MVC   PWD,=CL8' '         *Initialize values to space
         MVC   NPWD,=CL8' '        *Initialize values to space
         MVC   GROUP,=CL8' '       *Initialize values to space
         XC    STATBYTE,STATBYTE   *Nice and clean
*
* ---------------------------------------------------------------------
*
* Determine who caller is.  Done by including two weak external
* references in the code: one for EZAIMSLN (BMP Listener) and one
* for EZAIMSO0.  Only one of them is resolved, and the one that is
* resolved tells us which of the two listeners we are linked with.
*
* ---------------------------------------------------------------------
*
         ICM   R15,15,IMSLN        *Is caller BMP Listener ?
         BNZ   INITBMP             *- Yes
         ICM   R15,15,IMSO0        *Is caller OTMA Listener ?
         BNZ   INITOTMA            *- Yes
         WTO   'IMSLSECX - Caller is neither BMP nor OTMA Listener'
         ABEND 1002,DUMP
INITBMP  EQU   *
         MVC   APPLNAME,=CL8'BMPLSTN' *It is BMP Listener
         WTO   'IMSLSECX - Caller is BMP Listener'
         OI    STATBYTE,BMPCALL    *Indicate call from BMP
         B     INITAPPL            *
INITOTMA EQU   *
         MVC   APPLNAME,=CL8'OTMALSTN' *It is OTMA Listener
         WTO   'IMSLSECX - Caller is OTMA Listener'
         OI    STATBYTE,OTMACALL   *Indicate call from OTMA
INITAPPL EQU   *
         MVC   RESNAME,=C'TPI.IMSSOCK.' *Initialize
*
* ---------------------------------------------------------------------
*
* Check passed parameters and do any necessary conversion from
* ASCII to EBCDIC.
*
* ---------------------------------------------------------------------
*
         L     R2,LSDATLEN         *-> Fullword with L'userdata
         L     R9,0(R2)            *L'userdata
         C     R9,=A(16)           *User ID and Password must be there
         BL    TOFEWPRM            *Too few parameters passed
         BE    PARMOK              *Only userID and password is OK
         C     R9,=A(24)           *Exactly 24 bytes long
         BE    PARMOK              *- is OK - new password
         C     R9,=A(32)           *Or exactly 32 bytes long
         BNE   LENERR              *- is OK, if anything else: error
PARMOK   EQU   *
         L     R3,LSDATTYP         *-> Halfword with datatype
         L     R4,LSUSRDAT         *-> Userdata
         LH    R9,0(R3)            *Datatype
         LTR   R9,R9               *Is data ASCII ?
         BNZ   ISEBCDIC            *- No, data is EBCDIC
         CALL  TPITRANS,((R4),     *Translate user data area          C
               (R2),               *This length                       C
               STANDARD,           *Use STANDARD table                C
```

```
               ATOE),VL,           *From ASCII to EBCDIC              C
               MF=(E,MACWORK)      *
ISEBCDIC EQU   *
*
* ------------------------------------------------------------------
*
* Build TPIRACF parameters and call TPIRACF to verify user.
*
* ------------------------------------------------------------------
*
         MVC   USERID(32),=CL32' ' *Initialize all parms to space
         MVC   REQCODE,=A(REQVER)  *Issue RACROUTE REQUEST=VERIFY
         USING USERDATA,R4         *User data area addressability
         MVC   USERID,LSUSERID     *User ID from TRM
         MVC   PWD,LSPWD           *Password from TRM
         L     R9,0(R2)            *L'userdata
         C     R9,=A(16)           *Is there a new password ?
         BNH   DOVER               *- No, all parms are set
         MVC   NPWD,LSNPWD         *New password from TRM
         C     R9,=A(24)           *Is there a group ID ?
         BNH   DOVER               *- No, all parms are set
         MVC   GROUP,LSGROUP       *Group ID from TRM
DOVER    EQU   *
         CALL  TPIRACF,            *                                  C
               (REQCODE,           *RACROUTE REQUEST=VERIFY           C
               USERID,             *                                  C
               PWD,                *                                  C
               NPWD,               *                                  C
               GROUP,              *                                  C
               APPLNAME),VL,                                          C
               MF=(E,MACWORK)
         LTR   R15,R15             *Was VERIFY Successful?
         BNZ   VERFAIL             *- No, return error to client.
*
* ------------------------------------------------------------------
*
* Build TPIAUTH parameters and call TPIAUTH to test if user is
* authorized to TPI.IMSSOCK.trancode in the FACILITY resource class.
*
* ------------------------------------------------------------------
*
         L     R2,LSTRNNAM         *-> IMS Transaction code name
         MVC   RESTRNNM,0(R2)      *Move to FACILITY Class resource nm.
         CALL  TPIAUTH,            *Authorize call                    C
               (RESNAME,           *Resource name                     C
               AUTHACC),VL,        *Test for read access              C
               MF=(E,MACWORK)
         ST    R15,AUTHRC          *Save RC for a little later
*
* ------------------------------------------------------------------
*
* Delete user security environment again, so we restore address space
* security environment before we return to the IMS Listener.
*
* ------------------------------------------------------------------
*
         MVC   PWD(24),=CL24' '    *Space out unneeded parms
         MVC   REQCODE,=A(REQDEL)  *We want to delete sec. environment
```

```
         CALL  TPIRACF,            *                                C
               (REQCODE,           *RACROUTE REQUEST=DELETE         C
               USERID,             *                                C
               PWD,                *Space                           C
               NPWD,               *Space                           C
               GROUP,              *Space                           C
               APPLNAME),VL,                                        C
               MF=(E,MACWORK)
         LTR   R15,R15             *This should only give RC=0
         BZ    DELOK               *- which it did
         MVC   MACWORK(WTODELL),WTODEL *Move in WTO skeleton
         LR    R9,R15              *Save RC
         CVD   R15,DORD            *Convert
         OI    DORD+7,X'0F'        *- to something
         UNPK  MACWORK+WTODELRC(L'WTODELRC),DORD *Readable text
         WTO   MF=(E,MACWORK)      *Tell about it
         CH    R9,=AL2(253)        *Does not leave a security env.
         BE    DELOK               *- which is OK
         WTO   'IMSLSECX - User abend 1001 due to above return code'
         ABEND 1001,DUMP           *Others may leave user sec. active.
DELOK    EQU   *
         ICM   R9,B'1111',AUTHRC   *Return code from AUTH call
         BNZ   AUTHFAIL            *If not zero, auth failed.
         SR    R15,R15             *Set RC=0
         SR    R10,R10             *- and Reason code=0
* ------------------------------------------------------------------
*
* Check to see of caller is BMP listener or OTMA listener.
* If it is OTMA listener, we return the user ID and group ID to
* use in the OTMA headers.
*
* ------------------------------------------------------------------
         TM    STATBYTE,BMPCALL    *Was caller BMP Listener?
         BO    BMPLSTN             *- Yes, caller is BMP Listener
         L     R2,LSOTMAU          *-> Here to return OTMA user ID
         MVC   0(8,R2),USERID      *Use the passed user ID
         L     R2,LSOTMAG          *-> Here to return OTMA Group ID
         MVC   0(8,R2),GROUP       *This was the passed group ID
         WTO   'IMSLSECX - OTMA User ID and Group ID Returned'
BMPLSTN  EQU   *
         WTO   'IMSLSECX - Successfull processing, return to caller'
         B     RETURN              *And exit
*
* ------------------------------------------------------------------
*
* Error exit routines.  Set R15 to return code and R10 to
* reason code and go to common exit code.
*
* ------------------------------------------------------------------
*
AUTHFAIL EQU   *                   *User is not authorized
         LA    R10,NOTAUTH         *Not authorized to tran code
         LA    R15,8               *This is an error
         B     RETURN              *And exit
VERFAIL  EQU   *                   *User did not verify successfully
         LM    R5,R7,RCBXLE        *Prepare to set reason code
RCLOOP   EQU   *
         CH    R15,0(R5)           *This TPIRACF Return code ?
```

```
         BE    SETREAS             *- Yes, set corresponding reason
         BXLE  R5,R6,RCLOOP        *We use last entry as garbage can
SETREAS  EQU   *
         LH    R10,2(R5)           *Here is corresponding reason code
         LA    R15,8               *This is an error
         B     RETURN              *And exit
TOFEWPRM EQU   *
         LA    R15,8               *This is an error
         LA    R10,PARMERR1        *To few parameters
         B     RETURN              *Exit
LENERR   EQU   *
         LA    R15,8               *This is an error
         LA    R10,PARMERR2        *Wrong length
*
RETURN   EQU   *
         L     R2,LSRETCOD         *-> Return code field
         ST    R15,0(R2)           *Pass back return code
         L     R2,LSREACOD         *-> Reason code field
         ST    R10,0(R2)           *Pass back reason code
         TERM  RC=0                *Return to IMS Listener
         LTORG
*
* ------------------------------------------------------------------
*
* Macro list forms and work constants
*
* ------------------------------------------------------------------
*
* Reason codes
*
PARMERR1 EQU   101                 *At least user ID and password req.
PARMERR2 EQU   102                 *Length must be 16, 24 or 32.
NOTAUTH  EQU   113                 *User not authorized to trancode
*
RCBXLE   DC    A(START,4,LAST)     *TPIRACF RC to Reason code convert.
START    DC    AL2(4,103)          *User ID not defined to RACF
         DC    AL2(8,104)          *Invalid password
         DC    AL2(12,105)         *Password has expired
         DC    AL2(16,106)         *New password is not valid
         DC    AL2(20,107)         *User ID does not belong to group
         DC    AL2(24,108)         *User ID is revoked
         DC    AL2(28,109)         *Access to group is revoked
         DC    AL2(32,110)         *User ID is not authorized to appl
         DC    AL2(254,111)        *Internal error
LAST     DC    AL2(255,112)        *Some other error
*
AUTHACC  DC    CL8'READ'           *We want read access
*
STANDARD DC    CL8'STANDARD'       *Standard translate table
ATOE     DC    CL4'ATOE'           *From ASCII to EBCDIC
*
WTODEL   WTO   'IMSLSECX - RACROUTE REQUEST=DELETE Gave RC=xxxx',      C
               MF=L
WTODELL  EQU   *-WTODEL
WTODELRC EQU   48,4
*
IMSLN    DC    V(EZAIMSLN)         *The BMP Listener
IMSO0    DC    V(EZAIMSO0)         *The OTMA Listener
```

```
         WXTRN EZAIMSO0,EZAIMSLN   *Force these two to be weak refs.
*
         END
```

Part 3. Using The IMS Listener

Chapter 5. 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 6, “How to Write an IMS TCP/IP Client Program” on page 89 and Chapter 7, “How to Write an IMS TCP/IP Server Program” on page 97.

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 86, later in this chapter, and the sample program on “Sample Program—IMS MPP Client” on page 199.

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 for MVS 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.

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.

[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.

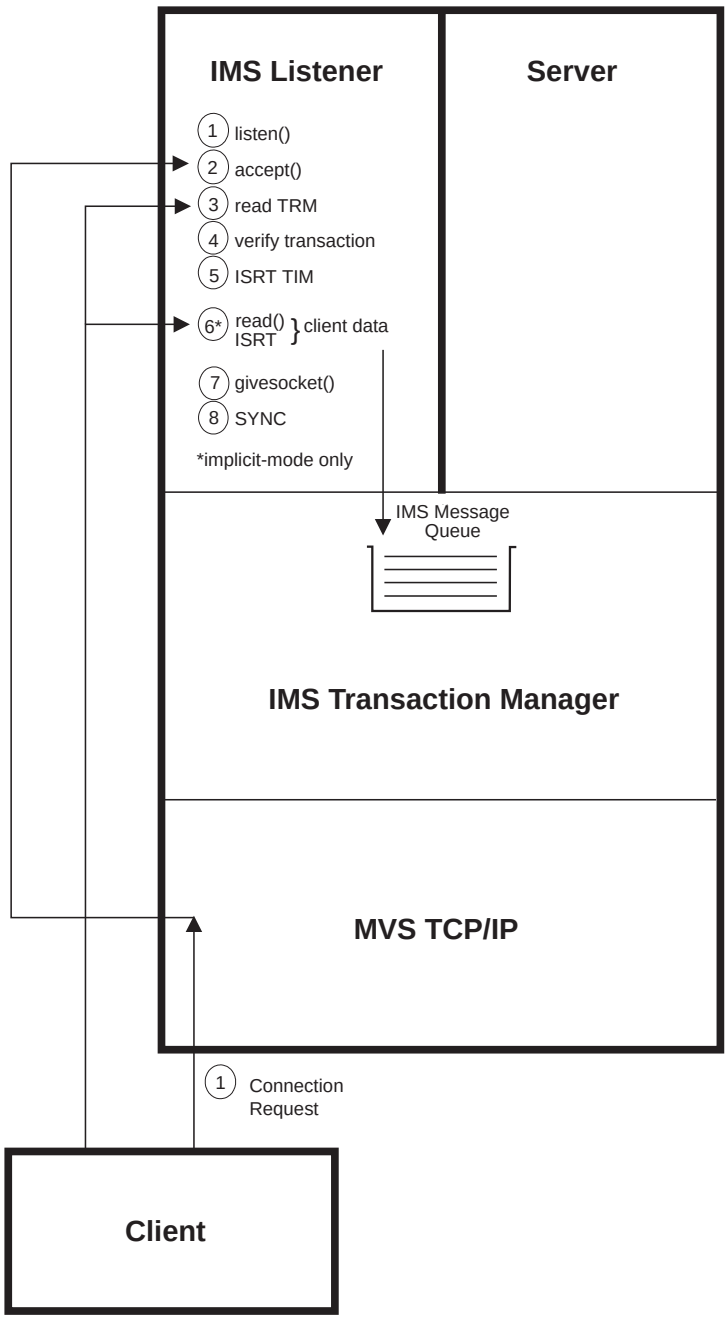


Figure 8. IMS TCP/IP Message Flow for Transaction Initiation

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.

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 estab-

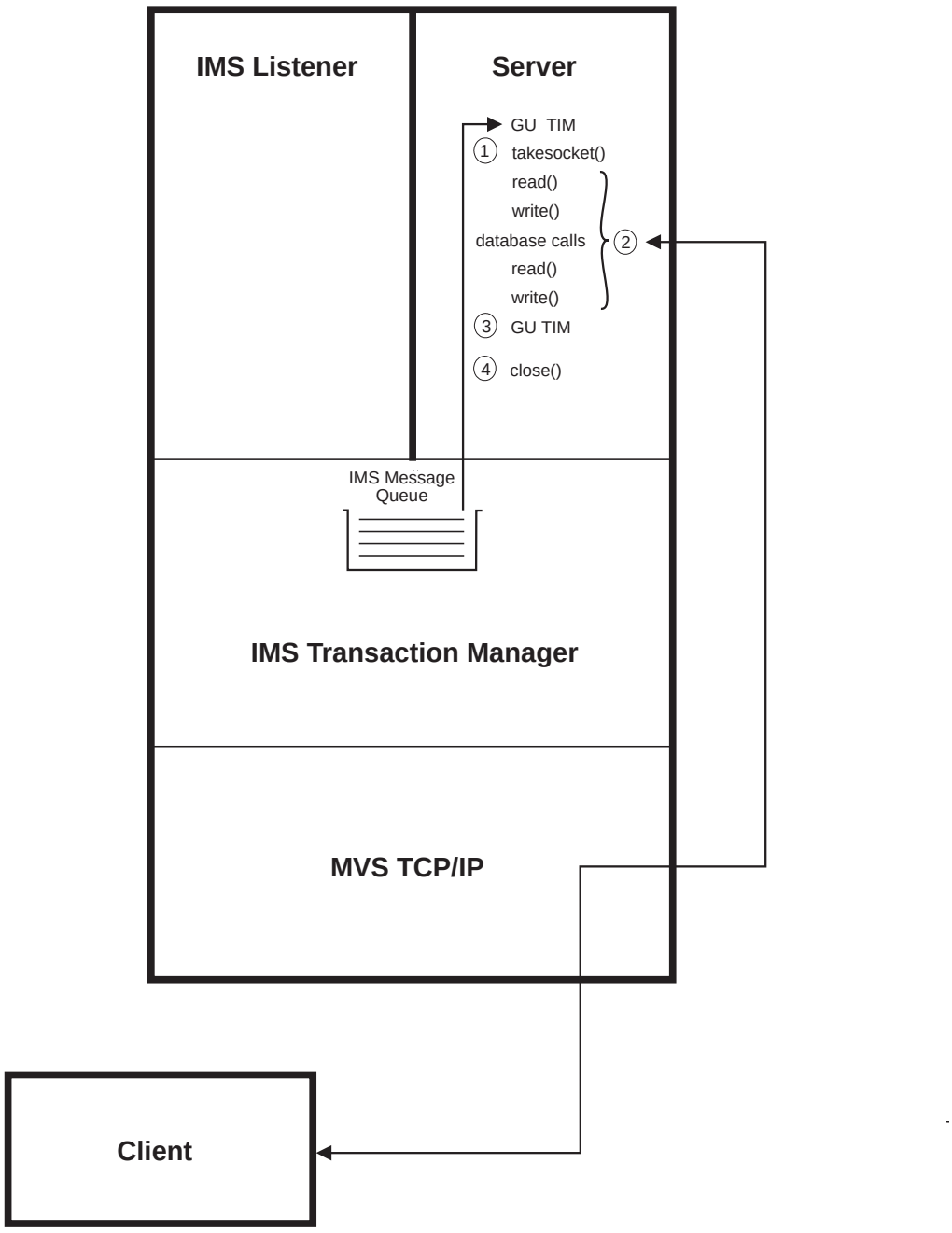


Figure 9. IMS TCP/IP Message Flow for Explicit-Mode Input/Output

lish 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.

[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.

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.

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.

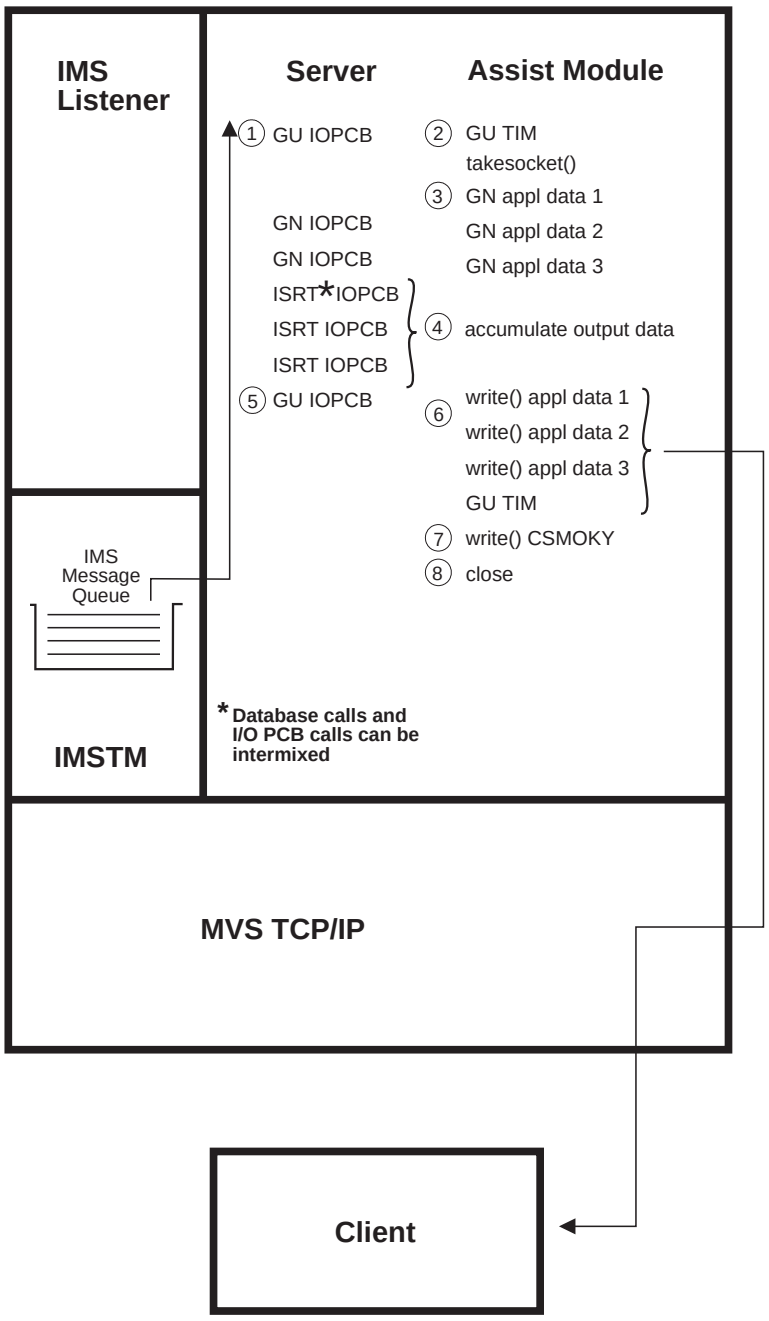


Figure 10. IMS TCP/IP Message Flow for Implicit Mode Input/Output

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 107.
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 107.

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.

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 6. 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 7, “How to Write an IMS TCP/IP Server Program” on page 97.

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 94 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.

---

[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.

End-of-Message Indicator

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

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 94 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 94 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 recieves 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]

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

Field	Format	Description
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 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 95).

- 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.
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.

  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
	H	Length of 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 7. 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 6, “How to Write an IMS TCP/IP Client Program” on page 89 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 9, “CALL Instruction Application Programming Interface (API)” on page 113 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 9, "CALL Instruction Application Programming Interface (API)" on page 113 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).

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

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.)
- 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 97). 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
	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.)

---

[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.

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 95).

- 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.)
- 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 221).

- 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 8. 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.

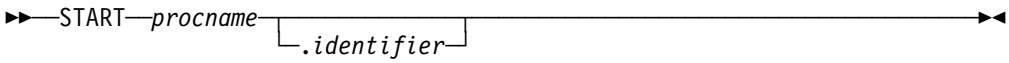
How to Start the IMS Listener

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 106.

```
.*
//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.

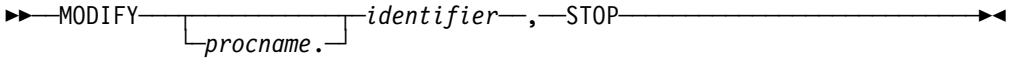


procname
The name of the cataloged procedure that defines the IMS Listener job to be started.

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 105, the `EZAIMSJL G.LSTNCFG DD` statement points to the Listener configuration file. This statement will be in the JCL sample you customize.

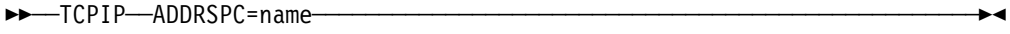
The configuration file contains three types of statements which must appear in the following order:

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.

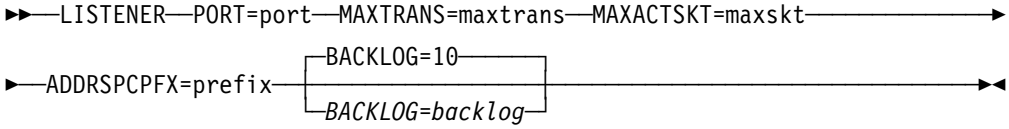


ADDRSPC= *name*

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*

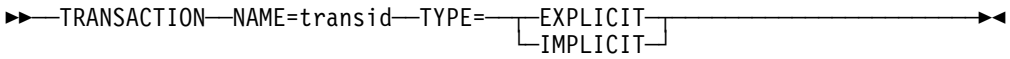
This parameter is optional and is used to specify the length of the backlog queue maintained in TCP/IP for connection requests that have not yet been assigned sockets by the Listener. Use an unsigned number from 1 to 32 767 inclusive. The default value is 10.

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 *OS/390 eNetwork Communications Server: IP API Guide* (SC31-8516) 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.
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 for MVS Definitions

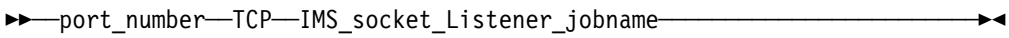
To run IMS, you need to modify the *tcpip*.PROFILE.TCPIP data set and the *hlq*.TCPIP.DATA [16]

data set that are part of the TCP/IP for MVS 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 110.

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 110 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.

[16] In this book, the abbreviation *hlq* stands for an installation-dependent *high level qualifier* which you must supply.

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

```
;
; 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 for MVS 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 106. 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 9. CALL Instruction Application Programming Interface (API)

This chapter describes the CALL instruction API for TCP/IP for MVS 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, see *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.

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

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.

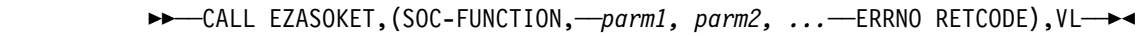
parm*n* A variable number of parameters depending on the type call.

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.



PL/I language call format

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

SOC-FUNCTION A 16-byte character field, left-justified and padded on the right with blanks. Set to the name of the call.

parm*n* A variable number of parameters depending on the type call.

ERRNO If RETCODE is negative, there is an error number in ERRNO. This field is used in most, but not all, of the calls. 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.

Programming Language Conversions

The parameter descriptions in this chapter are written using the VS COBOL II PIC language syntax and conventions. Use the syntax and conventions that are appropriate for the language you want to use. The following are examples of storage definition statements for COBOL, PL/I, and assembler language programs.

```
VS COBOL II PIC


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

COBOL PIC


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

PL/I DECLARE STATEMENT


  DCL   HALF       FIXED BIN(15),   HALFWORD BINARY VALUE
  DCL   FULL       FIXED BIN(31),   FULLWORD BINARY VALUE
  DCL   CHARACTER  CHAR(n)          CHARACTER FIELD OF n BYTES

ASSEMBLER DECLARATION


  DS    H                           HALFWORD BINARY VALUE
  DS    F                           FULLWORD BINARY VALUE
  DS    CLn                         CHARACTER FIELD OF n BYTES
```

Error Messages and Return Codes

For information about error messages, see *OS/390 eNetwork Communications Server: IP Messages Volume 1*.

For information about error codes that are returned by TCP/IP, see “Sockets Extended Return Codes” on page 229.

CALL Instructions for Assembler, PL/.I, and COBOL Programs

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 socket (s) remains 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.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'ACCEPT'.
    01  S               PIC 9(4) BINARY.
    01  NAME.
        03  FAMILY      PIC 9(4) BINARY.
        03  PORT        PIC 9(4) BINARY.
        03  IP-ADDRESS  PIC 9(8) BINARY.
        03  RESERVED    PIC X(8).
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

PROCEDURE
     CALL 'EZASOKET' USING SOC-FUNCTION S NAME ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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 “Sockets Extended Return Codes” on page 229, 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.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'BIND'.
    01  S               PIC 9(4) BINARY.
    01  NAME.
        03  FAMILY      PIC 9(4) BINARY.
        03  PORT        PIC 9(4) BINARY.
        03  IP-ADDRESS  PIC 9(8) BINARY.
        03  RESERVED    PIC X(8).
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

PROCEDURE
     CALL 'EZASOKET' USING SOC-FUNCTION S NAME ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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 zero 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 “Sockets Extended Return Codes” on page 229, 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 sytems you should consider the use of a SHUTDOWN call before you issue the CLOSE call. See “SHUTDOWN” on page 166 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 164.
  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.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'CLOSE'.
    01  S               PIC 9(4) BINARY.
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.


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

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

Parameter Values Returned to 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 Set by the Application

ERRNO A fullword binary field. If RETCODE is negative, this field contains an error number. See “Sockets Extended Return Codes” on page 229, for information about ERRNO return codes.

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

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

CONNECT

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

Stream Sockets

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

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

UDP Sockets

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

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

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

The blocking mode of the CONNECT call conditions its operation.

- If the socket is in blocking mode, the CONNECT call blocks the calling program until the connection is established, or until an error is received.
- If the socket is in nonblocking mode the return code indicates whether the connection request was successful.
  - A zero RETCODE indicates that the connection was completed.
  - A nonzero RETCODE with an ERRNO 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 152.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'CONNECT'.
    01  S               PIC 9(4) BINARY.
    01  NAME.
        03  FAMILY      PIC 9(4) BINARY.
        03  PORT        PIC 9(4) BINARY.
        03  IP-ADDRESS  PIC 9(8) BINARY.
        03  RESERVED    PIC X(8).
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.


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

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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 “Sockets Extended Return Codes” on page 229, 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 139 for another way to control a socket’s blocking mode.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'FCNTL'.
    01  S               PIC 9(4) BINARY.
    01  COMMAND         PIC 9(8) BINARY.
    01  REQARG          PIC 9(8) BINARY.
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.


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

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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 “Sockets Extended Return Codes” on page 229, 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 135 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.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'GETCLIENTID'.
    01  CLIENT.
        03  DOMAIN      PIC 9(8) BINARY.
        03  NAME        PIC X(8).
        03  TASK        PIC X(8).
        03  RESERVED    PIC X(20).
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

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

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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 "Sockets Extended Return Codes" on page 229, 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.

Note: The C runtime libraries are required when GETHOSTBYADDR is issued by your program.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'GETHOSTBYADDR'.
    01  HOSTADDR        PIC 9(8) BINARY.
    01  HOSTENT         PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

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

For equivalent PL/I and assembler language declarations, see "Programming Language Conversions" on page 114.

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.

Parameter Values Returned to the Application

HOSTENT A fullword containing the address of the HOSTENT structure.

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

Value	Description
0	Successful call
–1	An error occurred

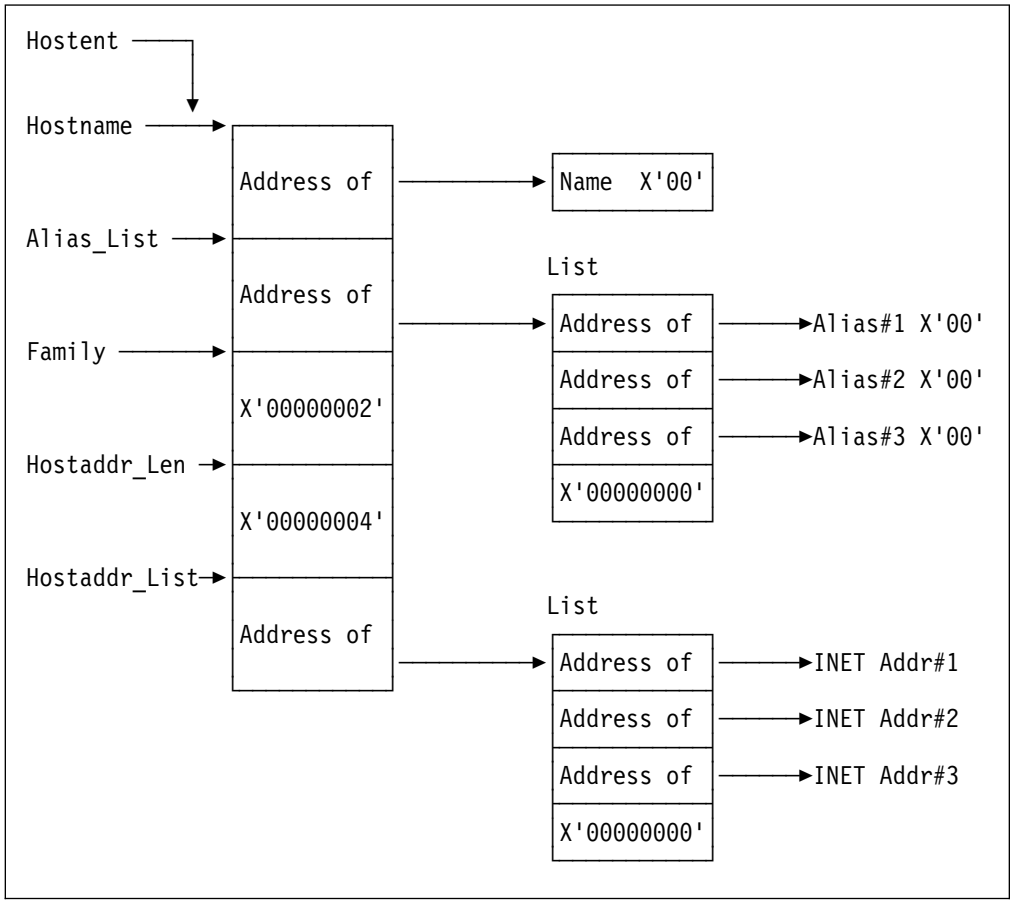


Figure 14. HOSTENT Structure Returned by the GETHOSTBYADDR Call

GETHOSTBYADDR returns the HOSTENT structure shown in Figure 14. This structure contains:

- The address of the host name that is returned by the call. The name length is variable and is ended by X'00'.
- The address of a list of addresses that point to the alias names returned by the call. This list is ended by the pointer X'00000000'. Each alias name is a variable length field ended by X'00'.
- The value returned in the FAMILY field is always 2 for AF_INET.

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

The HOSTENT structure uses indirect addressing to return a variable number of alias names and internet addresses. If you are coding in PL/I or assembler language, this structure can be processed in a relatively straight-forward manner. If you are coding in COBOL, this structure may be difficult to interpret. You can use the subroutine EZACIC08 to simplify interpretation of the information returned by the GETHOSTBYADDR and GETHOSTBYNAME calls. For more information about EZACIC08, see “EZACIC08” on page 175.

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 *OS/390 eNetwork Communications Server: IP Configuration*.
2. The C runtime libraries are required when GETHOSTBYNAME is issued by your program.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'GETHOSTBYNAME'.
    01  NAMELEN         PIC 9(8)  BINARY.
    01  NAME            PIC X(24).
    01  HOSTENT         PIC 9(8)  BINARY.
    01  RETCODE         PIC S9(8) BINARY.

PROCEDURE
     CALL 'EZASOKET' USING SOC-FUNCTION NAMELEN NAME
                     HOSTENT RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

Parameter Values Set by the Application

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

NAMELEN A value set to the length of the host name.

NAME A character string, up to 24 characters, set to a host name. This call returns the address of the HOSTENT structure for this name.

Parameter Values Returned to the Application

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

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

Value	Description
0	Successful call
–1	An error occurred

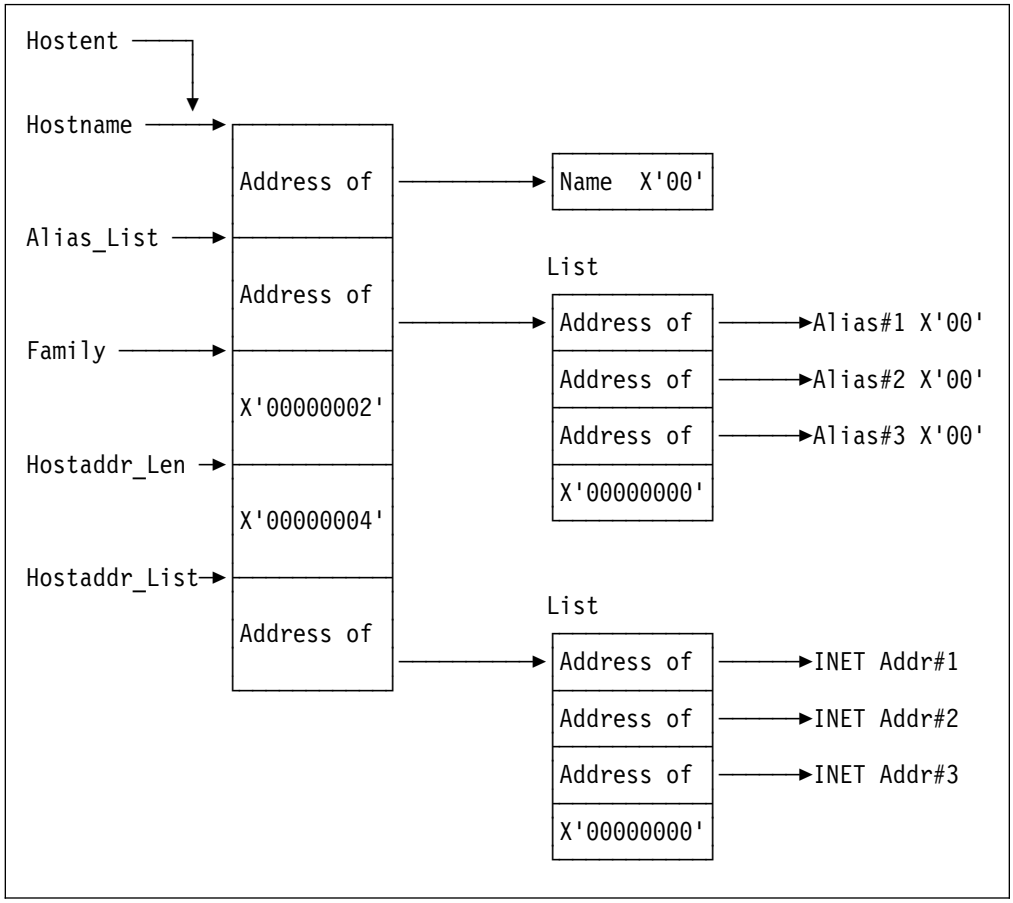


Figure 15. HOSTENT Structure Returned by the GETHOSTYBYNAME Call

GETHOSTBYNAME returns the HOSTENT structure shown in Figure 15. This structure contains:

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

The HOSTENT structure uses indirect addressing to return a variable number of alias names and internet addresses. If you are coding in PL/I or assembler language, this structure can be processed in a relatively straight-forward manner. If you are coding in COBOL, this structure may be difficult to interpret. You can use the subroutine EZACIC08 to simplify interpretation of the information returned by the GETHOSTBYADDR and GETHOSTBYNAME calls. For more information about EZACIC08, see “EZACIC08” on page 175.

GETHOSTID

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

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

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

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'GETHOSTNAME'.
    01  NAMELEN         PIC 9(8) BINARY.
    01  NAME            PIC X(24).
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

PROCEDURE
     CALL 'EZASOKET' USING SOC-FUNCTION NAMELEN NAME
                     ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

Parameter Values Set by the Application

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

NAMELEN A fullword binary field set to the length of the NAME field.

Parameter Values Returned to the Application

NAMELEN A fullword binary field set to the length of the host name.

NAME Indicates the receiving field for the host name. TCP/IP for MVS 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 “Sockets Extended Return Codes” on page 229, 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 for MVS 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 by using one of the following:

  SYSTCPD DD card
  jobname/userid.TCPIP.DATA
  zapname.TCPIP.DATA

For detailed information about the standard method, see *OS/390 eNetwork Communications Server: IP Planning and Migration Guide*.

```
 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.
```

Parameter Values Set by the Application

Parameter	Description
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 by the Application

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

On successful return, these buffer entries contain the status, names, and versions of up to 8 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'8000'	Active
X'4000'	Terminating
X'2000'	Down
X'1000'	Stopped or stopping

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 X'0302' for TCP/IP V3R2 for MVS.

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

NUM_IMAGES (4 bytes)		
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)
Status (2 bytes)	Version (2 bytes)	Name (8 bytes)

ERRNO A fullword binary field. If RETCODE is negative, this contains an error number. See “Sockets Extended Return Codes” on page 229, 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.

```
    WORKING STORAGE
        01  SOC-FUNCTION    PIC X(16)  VALUE IS 'GETPEERNAME'.
        01  S               PIC 9(4) BINARY.
        01  NAME.
            03  FAMILY      PIC 9(4) BINARY.
            03  PORT        PIC 9(4) BINARY.
            03  IP-ADDRESS  PIC 9(8) BINARY.
            03  RESERVED    PIC X(8).
        01  ERRNO           PIC 9(8) BINARY.
        01  RETCODE         PIC S9(8) BINARY.

    PROCEDURE
         CALL 'EZASOKET' USING SOC-FUNCTION S NAME ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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 "Sockets Extended Return Codes" on page 229, 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 zero.

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.

```
    WORKING STORAGE
        01  SOC-FUNCTION    PIC X(16)  VALUE IS 'GETSOCKNAME'.
        01  S               PIC 9(4) BINARY.
        01  NAME.
            03  FAMILY      PIC 9(4) BINARY.
            03  PORT        PIC 9(4) BINARY.
            03  IP-ADDRESS  PIC 9(8) BINARY.
            03  RESERVED    PIC X(8).
        01  ERRNO           PIC 9(8) BINARY.
        01  RETCODE         PIC S9(8) BINARY.

    PROCEDURE
         CALL 'EZASOKET' USING SOC-FUNCTION S NAME ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

Parameter Values Set by the Application

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

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

Parameter Values Returned to the Application

NAME 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 “Sockets Extended Return Codes” on page 229, 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.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'GETSOCKOPT'.
    01  S               PIC 9(4) BINARY.
    01  OPTNAME         PIC 9(8) BINARY.
        88  SO-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-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.
```

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

Parameter Values Set by the Application

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

S A halfword binary number specifying the socket descriptor for the socket requiring options.

OPTNAME Set OPTNAME to the required option before you issue GETSOCKOPT. The option are as follows:

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

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 DATABUFFERPOOLSIZE statement in the PROFILE.TCPIP data set. This value is adjusted to allow for protocol header information.

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 zero.
  - 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, to X'2' for SOCK-DGRAM, or to 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 zero value indicates that it is disabled.

  – The LINGER value indicates the amount of time (in seconds) TCP/IP will continue to attempt to send the data after the CLOSE call is issued. To *set* the Linger time, see “SETSOCKOPT” on page 164.

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 “Sockets Extended Return Codes” on page 229, 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 jobname 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 which use this socket. The socket descriptor that was passed to the TAKESOCKET call must not be used.
4. After issuing the GIVESOCKET command, the parent process issues a SELECT command that waits for the child to get the socket.

5. When the child gets the socket, the parent receives an exception condition that releases the SELECT command.
6. The parent process closes the socket.

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

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'GIVESOCKET'.
    01  S               PIC 9(4) BINARY.
    01  CLIENT.
        03  DOMAIN      PIC 9(8) BINARY.
        03  NAME        PIC X(8).
        03  TASK        PIC X(8).
        03  RESERVED    PIC X(20).
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

PROCEDURE
     CALL 'EZASOKET' USING SOC-FUNCTION S CLIENT ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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

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

TASK Specifies an 8-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 “Sockets Extended Return Codes” on page 229, 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.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'INITAPI'.
    01  MAXSOC          PIC 9(4) BINARY.
    01  IDENT.
        02  TCPNAME     PIC X(8).
        02  ADSNAME     PIC X(8).
    01  SUBTASK         PIC X(8).
    01  MAXSNO          PIC 9(8) BINARY.
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

PROCEDURE
     CALL 'EZASOKET' USING SOC-FUNCTION MAXSOC IDENT SUBTASK
     MAXSNO ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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 which should be set to the MVS jobname 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 jobname as part of your subtask name. This will ensure that, if you issue more than one INITAPI command from the same address space, each SUBTASK parameter will be unique.

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 “Sockets Extended Return Codes” on page 229, 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 141, for information about REQARG and RETARG.

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

       01  IFREQ,
         3 NAME                   PIC X(16).
         3 FAMILY                 PIC 9(4)  BINARY.
         3 PORT                   PIC 9(4)  BINARY.
         3 ADDRESS                PIC 9(8)  BINARY.
         3 RESERVED               PIC X(8).

       01  IFREQOUT,
         3 NAME                   PIC X(16).
         3 FAMILY                 PIC 9(4)  BINARY.
         3 PORT                   PIC 9(4)  BINARY.
         3 ADDRESS                PIC 9(8)  BINARY.
         3 RESERVED               PIC X(8).

       01  GRP_IOCTL_TABLE(100)
        02 IOCTL_ENTRY,
         3 NAME                   PIC X(16).
         3 FAMILY                 PIC 9(4)  BINARY.
         3 PORT                   PIC 9(4)  BINARY.
         3 ADDRESS                PIC 9(8)  BINARY.
         3 NULLS                  PIC X(8).

       01  IOCTL_REQARG           POINTER ;
       01  IOCTL_RETARG           POINTER ;
       01  ERRNO                  PIC 9(8) BINARY.
       01  RETCODE                PIC 9(8) BINARY.


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

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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 16 for the address format.

'SIOCGIFBRDADDR'
Requests the network interface broadcast address for a given interface name. See the NAME field in Figure 16 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 16.

- 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 16. 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 16 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 (Page 1 of 2). 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 16 on page 140 for format.)
SIOCGIFBRDADDR X'C020A712'	32	First 16 bytes—interface name. Last 16 bytes—not used	32	Network interface address (See Figure 16 on page 140 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 for MVS 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 16 on page 140 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
SIOCGIFNETMASK X'C020A715'	32	For IBM use only	32	For IBM use only

Table 3 (Page 2 of 2). IOCTL call arguments

COMMAND/CODE	SIZE	REQARG	SIZE	RETARG
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 “Sockets Extended Return Codes” on page 229, 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 17 contains an example of a COBOL II routine which can be used to work with such a structure.

Note: This call can only be programmed in languages which support address pointers.

```
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 17. 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.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'LISTEN'.
    01  S               PIC 9(4) BINARY.
    01  BACKLOG         PIC 9(8) BINARY.
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

PROCEDURE
    CALL 'EZASOKET' USING SOC-FUNCTION S BACKLOG ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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 221, 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 173 for a subroutine that will translate ASCII input data to EBCDIC.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'READ'.
    01  S               PIC 9(4) BINARY.
    01  NBYTE           PIC 9(8) BINARY.
    01  BUF             PIC X(length of buffer).
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

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

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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 “Sockets Extended Return Codes” on page 229, for information about ERRNO return codes.

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

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

READV

The READV 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.

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

01  MSG-HDR.
    03 MSG_NAME            POINTER.
    03 MSG_NAME_LEN        POINTER.
    03 IOVPTR              POINTER.
    03 IOVCNT              POINTER.
    03 MSG_ACCRIGHTS       PIC X(4).
    03 MSG_ACCRIGHTS_LEN   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.
```

Parameter Values Set by the Application

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 1.

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.

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

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

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 non-blocking mode, RECV returns a –1 and sets ERRNO to 35 (EWOULDBLOCK). See “FCNTL” on page 121 or “IOCTL” on page 139 for a description of how to set non-blocking mode.

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

Note: See “EZACIC05” on page 173 for a subroutine that will translate ASCII input data to EBCDIC.

```
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.
```

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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 “Sockets Extended Return Codes” on page 229, 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.

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 121 or “IOCTL” on page 139 for a description of how to set nonblocking mode.

Note: See “EZACIC05” on page 173 for a subroutine that will translate ASCII input data to EBCDIC.

```
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.
```

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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 “Sockets Extended Return Codes” on page 229, 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.

```
 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.
```

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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'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 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 by the Application

ERRNO A fullword binary field. If RETCODE is negative, this contains an error number.

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 either of the following:

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

Defining Which Sockets to Test

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

- When a socket is ready to read, either:
  - 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 right-most bit represents socket descriptor 0; the left-most bit represents socket descriptor 31, and so on. If your process uses 32 or fewer sockets, the bit string is one fullword. If your process uses 33 sockets, the bit string is two full words. 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 to a character. For more information, see “EZACIC06” on page 173.

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 '1' before issuing the SELECT call. When the SELECT call returns, the corresponding bits in the RRETMSK indicate sockets ready for reading.

Write Operations

A socket is selected for writing (ready to be written) when:

- TCP/IP can accept additional outgoing data.
- The socket is marked nonblocking and a previous CONNECT did not complete immediately. In this case, CONNECT returned an ERRNO with a value 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 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, RETCODE is set to 0.

```
    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
```

Bit masks are 32-bit fullwords with 1 bit for each socket. Up to 32 sockets fit into 1 32-bit mask (PIC X(4)). If you have 33 sockets, you must allocate 2 32-bit masks (PIC X(8)).

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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 zero).

TIMEOUT If TIMEOUT is a positive value, it specifies the maximum interval to wait for the selection to complete. If TIMEOUT-SECONDS is a negative value, the SELECT call blocks until a socket becomes ready. To poll the sockets and return immediately, specify the TIMEOUT value to be zero.

TIMEOUT is specified in the 2-word TIMEOUT as follows:

- TIMEOUT-SECONDS, word 1 of the TIMEOUT field, is the seconds component of the time-out value.
- TIMEOUT-MICROSEC, word 2 of the TIMEOUT field, is the microseconds component of the time-out value (0 through 999999).

For example, if you want SELECT to timeout after 3.5 seconds, set TIMEOUT-SECONDS to 3 and TIMEOUT-MICROSEC to 500000.

RSNDMSK A bit string sent to request read event status.

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

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

WSNDMSK A bit string sent to request write event status.

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

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

ESNDMSK A bit string sent to request exception event status.

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

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

Parameter Values Returned to the Application

RRETMSK A bit string returned with the status of read events. The length of the string should be equal to the maximum number of sockets to be checked. For each socket that is ready to read, the corresponding bit in the string will be set to 1; bits that represent sockets that are not ready to read will be set to 0.

WRETMSK A bit string returned with the status of write events. The length of the string should be equal to the maximum number of sockets to be checked. For each socket that is ready to write, the corresponding bit in the string will be set to 1; bits that represent sockets that are not ready to be written will be set to 0.

ERETMSK A bit string returned with the status of exception events. The length of the string should be equal to the maximum number of sockets to be checked. For each socket that has an exception status, the corresponding bit will be set to 1; bits that represent sockets that do not have exception status will be set to 0.

ERRNO A fullword binary field. If RETCODE is negative, the field contains an error number. See “Sockets Extended Return Codes” on page 229, for information about ERRNO return codes.

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

Value	Description
>0	Indicates the sum of all ready sockets in the three masks
0	Indicates that the SELECT time limit has expired
–1	Check ERRNO for an error code

SELECTEX

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

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

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

```
 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
```

Parameter Values Set by the Application

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 2-word TIMEOUT as follows:

- TIMEOUT-SECONDS, word 1 of the TIMEOUT field, is the seconds component of the time-out value.
- TIMEOUT-MICROSEC, word 2 of the TIMEOUT field, is the microseconds component of the time-out value (0 through 999999).

For example, if you want SELECTEX to timeout after 3.5 seconds, set TIMEOUT-SECONDS to 3 and TIMEOUT-MICROSEC to 500000.

RSNDMSK The bit-mask array to control checking for read interrupts. If this parameter is not specified or the specified bit-mask is zeros, the SELECT will not check for read interrupts. The length of this bit-mask array is dependent on the value in MAXSOC.

WSNDMSK The bit-mask array to control checking for write interrupts. If this parameter is not specified or the specified bit-mask is zeros, the SELECT will not check for write interrupts. The length of this bit-mask array is dependent on the value in MAXSOC.

ESNDMSK The bit-mask array to control checking for exception interrupts. If this parameter is not specified or the specified bit-mask is zeros, the SELECT will not check for exception interrupts. The length of this bit-mask array is dependent on the value in MAXSOC.

SELECB An ECB which, if posted, causes completion of the SELECTEX.

If the address of an ECB list is specified, you must set the high-order bit of the last entry in the ECB list to one to signify it is the last entry, and you must add the LIST keyword. The ECBs must reside in the caller's primary address space.

Parameter Values Returned by the Application

ERRNO A fullword binary field. If RETCODE is negative, this contains an error number.

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 non-zero 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	Error. Check ERRNO.

RRETMSK The bit-mask array returned by the SELECT if RSNDMSK is specified. The length of this bit-mask array is dependent on the value in MAXSOC.

WRETMSK The bit-mask array returned by the SELECT if WSNDMSK is specified. The length of this bit-mask array is dependent on the value in MAXSOC.

ERETMSK The bit-mask array returned by the SELECT if ESNDMSK is specified. The length of this bit-mask array is dependent on the value in MAXSOC.

SEND

The SEND call sends data on a specified connected socket.

The FLAGS field allows you to:

- Send out-of-band data, for example, interrupts, aborts, and data marked urgent. Only stream sockets created in the AF_INET address family support out-of-band data.
- Suppress use of local routing tables. This implies that the caller takes control of routing, 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 172 for a subroutine that will translate EBCDIC input data to ASCII.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'SEND'.
    01  S               PIC 9(4) BINARY.
    01  FLAGS           PIC 9(8) BINARY.
        88  NO-FLAG                VALUE IS 0.
        88  OOB                    VALUE IS 1.
        88  DONT-ROUTE             VALUE IS 4.
    01  NBYTE           PIC 9(8) BINARY.
    01  BUF             PIC X(length of buffer).
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

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

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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 “Sockets Extended Return Codes” on page 229, for information about ERRNO return codes.

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

Value	Description
0 or >0	A successful call. The value is set to the number of bytes transmitted.
–1	Check ERRNO for an error code

SENDMSG

The SENDMSG call sends messages on a socket with descriptor S passed in an array of messages.

```
 WORKING STORAGE
     01  SOC-FUNCTION    PIC X(16)  VALUE IS 'SENDMSG        '.
     01  S               PIC 9(4)   BINARY.
     01  MSG-HDR.
         03  MSG-NAME        USAGE IS POINTER.
         03  MSG-NAME-LEN    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.
```

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

Parameter Values Set by the Application

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.
ACCRLEN	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.
DONT-ROUTE	4	Do not route. Routing is provided by the calling program.

Parameter Values Returned by the Application

ERRNO A fullword binary field. If RETCODE is negative, this contains an error number.

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

Value	Description
0 or >0	A successful call. The value is set to the number of bytes transmitted.
–1	Check ERRNO for an error code.

SENDTO

SENDTO is similar to SEND, except that it includes the destination address parameter. The destination address allows you to use the SENDTO call to send datagrams on a UDP socket, regardless of whether or not 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 172 for a subroutine that will translate EBCDIC input data to ASCII.

```
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.
```

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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 “Sockets Extended Return Codes” on page 229, for information about ERRNO return codes.

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

Value	Description
0 or >0	A successful call. The value is set to the number of bytes transmitted.
–1	Check ERRNO for an error code

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.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'SETSOCKOPT'.
    01  S               PIC 9(4) BINARY.
    01  OPTNAME         PIC 9(8) BINARY.
        88  SO-REUSEADDR  VALUE  4.
        88  SO-KEEPALIVE  VALUE  8.
        88  SO-BROADCAST  VALUE  32.
        88  SO-LINGER     VALUE  128.
        88  SO-OOBINLINE  VALUE  256.
    01  OPTVAL          PIC 9(16) BINARY.
    01  OPTLEN          PIC 9(8) BINARY.
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

PROCEDURE
     CALL 'EZASOKET' USING SOC-FUNCTION S OPTNAME
                    OPTVAL OPTLEN ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

Parameter Values Set by the Application

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

S A halfword binary number set to the socket whose options are to be set.

OPTNAME Specify one of the following values.

SO-REUSEADDR Toggles local address re-useability. 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.

The default is DISABLED.

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 which support the receipt of packets.

The default is DISABLED.

SO-KEEPALIVE Toggles the TCP keep-alive mechanism for a stream socket. The default is disabled. When activated, the keep-alive mechanism periodically sends a packet on an otherwise idle connection. If the remote TCP does not respond to the packet or to retransmissions of the packet, the connection is terminated with the error ETIMEDOUT.

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.

OPTVAL Contains data which 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 zero to disable the option.
- For SO-LINGER, OPTVAL assumes the following structure:

```
ONOFF       PIC X(4).
LINGER      PIC 9(8) BINARY.
```

Set ONOFF to a nonzero value to enable the option; set it to zero to disable the option. Set the LINGER value to the amount of time (in seconds) TCP/IP will linger after the CLOSE call.

OPTLEN A fullword binary number specifying the length of the data returned in OPTVAL.

Parameter Values Returned to the Application

ERRNO A fullword binary field. If RETCODE is negative, the field contains an error number. See “Sockets Extended Return Codes” on page 229, 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.

```
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.
```

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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 “Sockets Extended Return Codes” on page 229, 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.

```
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.
```

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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 zero, 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 “Sockets Extended Return Codes” on page 229, for information about ERRNO return codes.

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

Value	Description
> **or** = **0**	Contains the new socket descriptor
–**1**	Check ERRNO for an error code

TAKESOCKET

The TAKESOCKET call acquires a socket from another program and creates a new socket. Typically, a child server issues this call using client ID and socket descriptor data which it obtained from the concurrent server. See “GIVESOCKET” on page 135 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.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'TAKESOCKET'.
    01  SOCRECV         PIC 9(4) BINARY.
    01  CLIENT.
        03  DOMAIN      PIC 9(8) BINARY.
        03  NAME        PIC X(8).
        03  TASK        PIC X(8).
        03  RESERVED    PIC X(20).
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

PROCEDURE
     CALL 'EZASOKET' USING SOC-FUNCTION  SOCRECV CLIENT
                     ERRNO RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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 “Sockets Extended Return Codes” on page 229 for information about ERRNO return codes.

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

Value	Description
> **or** = **0**	Contains the new socket descriptor
–**1**	Check ERRNO for an error code

TERMAPI

This call terminates the session created by INITAPI.

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

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

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'WRITE'.
    01  S               PIC 9(4) BINARY.
    01  NBYTE           PIC 9(8) BINARY.
    01  BUF             PIC X(length of buffer).
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

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

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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 “Sockets Extended Return Codes” on page 229, for information about ERRNO return codes.

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

Value	Description
0 or >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.

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

       01  MSG-HDR.
           03 MSG_NAME            POINTER.
           03 MSG_NAME_LEN        POINTER.
           03 IOVPTR              POINTER.
           03 IOVCNT              POINTER.
           03 MSG_ACCRIGHTS       PIC X(4).
           03 MSG_ACCRIGHTS_LEN   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.
```

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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.

RETCODE — A fullword binary field.

Value	Meaning
<**0**	Error. Check ERRNO.
0	Connection partner has closed connection.
>**0**	Number of bytes sent.

Data Translation Programs for the 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, etc; 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.

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

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

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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.

Examples
None.

EZACIC05

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

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

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

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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.

Examples
None.

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.

```
WORKING STORAGE
    01  TOKEN                PIC X(16) VALUE 'TCPIPBITMASKCOBL'.
    01  CH-MASK.
        05 CHAR-STRING        PIC X(nn).
    01  CHAR-ARRAY  REDEFINES CH-MASK.
        05  CHAR-ENTRY-TABLE  OCCURS nn TIMES.
            10  CHAR-ENTRY    PIC X(1).
    01  BIT-MASK.
        05  BIT-ARRAY-FWDS    PIC X(*) BINARY.
    01  COMMAND.
        05  CTOB              PIC X(4) VALUE 'CTOB'.
        05  BTOC              PIC X(4) VALUE 'BTOC'.
    01  BIT-LNGTH             PIC 9(8) BINARY VALUE '8'.
    01  RETCODE               PIC 9(8) BINARY.

PROCEDURE
     CALL 'EZACIC06' USING TOKEN BTOC BIT-MASK CH-MASK
           BIT-LENGTH RETCODE.
```

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

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 right-most bit representing socket 0. The socket positions in the character array are indexed starting with 1 making socket 0 index number 1 in the character array. You should keep this in mind when turning character positions on and off.

COMMAND — BTOC—Specifies bit string to character array translation.

CTOB—Specifies character array to bit string translation.

BIT-LNGTH — 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-ENTRY-TABLE.
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/I or assembler language, the HOSTENT structure can be processed in a relatively straight-forward manner. However, if you are coding in COBOL, HOSTENT can be more difficult to process and you should use the EZACIC08 subroutine to process it for you.

Here is how it works:

- GETHOSTBYADDR or GETHOSTBYNAME returns a HOSTENT structure that indirectly addresses the lists of alias names and internet addresses.
- Upon return from GETHOSTBYADDR or GETHOSTBYNAME your program calls EZACIC08 and passes it the address of the HOSTENT structure. EZACIC08 processes the structure and returns the following:
  1. The length of host name, if present
  2. The host name
  3. The number of alias names for the host
  4. The alias name sequence number
  5. The length of the alias name
  6. The alias name
  7. The host internet address type, always 2 for AF_INET
  8. The host internet address length, always 4 for AF_INET
  9. The number of host internet addresses for this host
  10. The host internet address sequence number.
  11. The host internet address.
- If the GETHOSTBYADDR or GETHOSTBYNAME call returns more than one alias name or host internet address (3 and 9 above), the application program should repeat the call to EZACIC08 until all alias names and host internet addresses have been retrieved.

```
WORKING STORAGE

    01  HOSTENT-ADDR       PIC 9(8) BINARY.
    01  HOSTNAME-LENGTH    PIC 9(4) BINARY.
    01  HOSTNAME-VALUE     PIC X(255)
    01  HOSTALIAS-COUNT    PIC 9(4) BINARY.
    01  HOSTALIAS-SEQ      PIC 9(4) BINARY.
    01  HOSTALIAS-LENGTH   PIC 9(4) BINARY.
    01  HOSTALIAS-VALUE    PIC X(255)
    01  HOSTADDR-TYPE      PIC 9(4) BINARY.
    01  HOSTADDR-LENGTH    PIC 9(4) BINARY.
    01  HOSTADDR-COUNT     PIC 9(4) BINARY.
    01  HOSTADDR-SEQ       PIC 9(4) BINARY.
    01  HOSTADDR-VALUE     PIC 9(8) BINARY.
    01  RETURN-CODE        PIC 9(8) BINARY.

PROCEDURE

   CALL 'EZASOKET' USING 'GETHOSTBYxxxx'
                   HOSTENT-ADDR
                   RETCODE.

   Where xxxx is ADDR or NAME.

   CALL 'EZACIC08' USING HOSTENT-ADDR HOSTNAME-LENGTH
                   HOSTNAME-VALUE HOSTALIAS-COUNT HOSTALIAS-SEQ
                   HOSTALIAS-LENGTH HOSTALIAS-VALUE
                   HOSTADDR-TYPE HOSTADDR-LENGTH HOSTADDR-COUNT
                   HOSTADDR-SEQ HOSTADDR-VALUE RETURN-CODE
```

For equivalent PL/I and assembler language declarations, see “Programming Language Conversions” on page 114.

Parameter Values set by the Application:

HOSTENT-ADDR This fullword binary field must contain the address of the HOSTENT structure (as returned by the GETHOSTBYxxxx call). This variable is the same as the variable HOSTENT in the GETHOSTBYADDR and GETHOSTBYNAME socket calls.

HOSTALIAS-SEQ This halfword field is used by EZACIC08 to index the list of alias names. When EZACIC08 is called, it adds 1 to the current value of HOSTALIAS-SEQ and uses the resulting value to index into the table of alias names. Therefore, for a given instance of GETHOSTBYxxxx, this field should be set to zero 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 zero for the initial call to EZACIC08. For all subsequent

calls to EZACIC08, this field should contain the HOSTADDR-SEQ number returned by the previous call.

Parameter Values Returned to the Application:

HOSTNAME-LENGTH This halfword binary field contains the length of the host name (if host name was returned).

HOSTNAME-VALUE This 255-byte character string contains the host name (if host name was returned).

HOSTALIAS-COUNT This halfword binary field contains the number of alias names returned.

HOSTALIAS-SEQ This halfword binary field is the sequence number of the alias name currently found in HOSTALIAS-VALUE.

HOSTALIAS-LENGTH This halfword binary field contains the length of the alias name currently found in HOSTALIAS-VALUE.

HOSTALIAS-VALUE This 255-byte character string contains the alias name returned by this instance of the call. The length of the alias name is contained in HOSTALIAS-LENGTH.

HOSTADDR-TYPE This halfword binary field contains the type of host address. For FAMILY type AF_INET, HOSTADDR-TYPE is always 2.

HOSTADDR-LENGTH This halfword binary field contains the length of the host internet address currently found in HOSTADDR-VALUE. For FAMILY type AF_INET, HOSTADDR-LENGTH is always set to 4.

HOSTADDR-COUNT This halfword binary field contains the number of host internet addresses returned by this instance of the call.

HOSTADDR-SEQ This halfword binary field contains the sequence number of the host internet address currently found in HOSTADDR-VALUE.

HOSTADDR-VALUE This fullword binary field contains a host internet address.

RETURN-CODE This fullword binary field contains the EZACIC08 return code:

Value	Description
0	Successful completion
-1	Invalid HOSTENT address

Chapter 10. 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 Linking an Implicit-Mode Server

The following JCL is an example (using an IMS-supplied procedure) of JCL that can be used to compile and link the Assist module into an assembler language implicit-mode server program. (Use the IMS procedure appropriate for your language). Note the requirement for AMODE(31); the requirement for linking to the MVS TCP/IP Socket API is satisfied by the EZAIMSAS INCLUDE statement.

```
//STEP1  EXEC  IMSASM,MBR=your implicit-mode application
//C.SYSIN  DD  DSN=IMS31.APPLSRC(&MBR;),DISP=SHR

//IMSLIB   DD  DSN=hlq.SEZALINK,DISP=SHR
//L.SYSLIN   DD  DSN=*.C.SYSLIN,DISP=(OLD,DELETE)
//           DD  *
   INCLUDE    IMSLIB(EZAIMSAS)
   INCLUDE    IMSRES(ASMTDLI)
   MODE       RMODE(24),AMODE(31)
   NAME       your implicit mode-application (R)
/*
```

JCL for Linking an Explicit-Mode Server

The following is an example (using an IMS-supplied procedure) of JCL that can be used to compile and link an assembler language explicit-mode server program. (Use the IMS procedure appropriate for your language). Note the requirement for AMODE(31) and the provision for linking to the MVS TCP/IP Socket API.

```
//STEP1  EXEC  IMSASM,MBR=your explicit application
//C.SYSIN  DD  DSN=IMS31.APPLSRC(&MBR;),DISP=SHR
//TCPLIB   DD  DSN=&hlq;.SEZATCP,DISP=SHR
//L.SYSLIN   DD  DSN=*.C.SYSLIN,DISP=(OLD,DELETE)
//           DD  *

   INCLUDE    TCPLIB(EZASOKET)
   INCLUDE    IMSRES(ASMTDLI)
   MODE       RMODE(24),AMODE(31)
   NAME       your explicit application(R)
//
```

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 *Program Directory* or the section on C compilers and run time libraries in *OS/390 eNetwork Communications Server: IP API 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
```

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", argvfff0"); */
         printf("Usage: %s hostname port\n", argv        fff0");
        exit(1);
    }
         printf("Usage: %s hostname port\n", argv        fff0");
    /*
     * The host name is the first argument. Get the server address.
     */
```

```
hostnm = gethostbyname(argvff1");
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(argvff2");
/*
 * 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. 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(argvff1");
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(argvff2");
/*
 * 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

Server Output

Part 4. 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.

Table 4 (Page 1 of 9). System Error Return Codes

Error Number	Message Name	Socket Type	Error Description	Programmer's Response
1	EPERM	All	Permission is denied. No owner exists.	Check that TPC/IP is still active; check protocol value of socket () call.
1	EDOM	All	Argument too large.	Check parameter values of the function call.
2	ENOENT	All	The data set or directory was not found.	Check files used by the function call.
2	ERANGE	All	The result is too large.	Check parameter values of the function call.
3	ESRCH	All	The process was not found. A table entry was not located.	Check parameter values and structures pointed to by the function parameters
4	EINTR	All	A system call was interrupted.	Check that the socket connection and TCP/IP are still active.
5	EIO	All	An I/O error occurred.	Check status and contents of source database if this occurred during a file access.
6	ENXIO	All	The device or driver was not found.	Check status of the device attempting to access.
7	E2BIG	All	The argument list is too long.	Check the number of function parameters.
8	ENOEXEC	All	An EXEC format error occurred.	Check that the target module on an exec call is a valid executable module.
9	EBADF	All	An incorrect socket descriptor was specified.	Check socket descriptor value. It might be currently not in use or incorrect.
9	EBADF	Givesocket	The socket has already been given. The socket domain is not AF_INET.	Check the validity of function parameters.
9	EBADF	Select	One of the specified descriptor sets is an incorrect socket descriptor.	Check the validity of function parameters.
9	EBADF	Takesocket	The socket has already been taken.	Check the validity of function parameters.
10	ECHILD	All	There are no children.	Check if created subtasks still exist.
11	EAGAIN	All	There are no more processes.	Retry the operation. Data or condition might not be available at this time.
12	ENOMEM	All	There is not enough storage.	Check validity of function parameters.
13	EACCES	All	Permission denied, caller not authorized.	Check access authority of file.

Table 4 (Page 2 of 9). System Error Return Codes

Error Number	Message Name	Socket Type	Error Description	Programmer's Response
13	EACCES	Takesocket	The other application (listener) did not give the socket to your application. Permission denied, caller not authorized.	Check access authority of file.
14	EFAULT	All	An incorrect storage address or length was specified.	Check validity of function parameters.
15	ENOTBLK	All	A block device is required.	Check device status and characteristics .
16	EBUSY	All	Listen has already been called for this socket. Device or file to be accessed is busy.	Check if the device or file is in use.
17	EEXIST	All	The data set exists.	Remove or rename existing file.
18	EXDEV	All	This is a cross-device link. A link to a file on another file system was attempted.	Check file permissions.
19	ENODEV	All	The specified device does not exist.	Check file name and if it exists.
20	ENOTDIR	All	The specified directory is not a directory.	Use a valid file that is a directory.
21	EISDIR	All	The specified directory is a directory.	Use a valid file that is not a directory.
22	EINVAL	All types	An incorrect argument was specified.	Check validity of function parameters.
23	ENFILE	All	Data set table overflow occurred.	Reduce the number of open files.
24	EMFILE	All	The socket descriptor table is full.	Check the maximum sockets specified in MAXDESC().
25	ENOTTY	All	An incorrect device call was specified.	Check specified IOCTL() values.
26	ETXTBSY	All	A text data set is busy.	Check the current use of the file.
27	EFBIG	All	The specified data set is too large.	Check size of accessed dataset.
28	ENOSPC	All	There is no space left on the device.	Increase the size of accessed file.
29	ESPIPE	All	An incorrect seek was attempted.	Check the offset parameter for seek operation.
30	EROFS	All	The data set system is Read only.	Access data set for read only operation.
31	EMLINK	All	There are too many links.	Reduce the number of links to the accessed file.
32	EPIPE	All	The connection is broken. For AF_IUCV socket write/send, peer has shutdown one or both directions.	Reconnect with the peer.
33	EDOM	All	The specified argument is too large.	Check and correct function parameters.
34	ERANGE	All	The result is too large.	Check function parameter values.
35	EWOULDBLOCK	Accept	The socket is in nonblocking mode and connections are not queued. This is not an error condition.	Reissue Accept().

Table 4 (Page 3 of 9). System Error Return Codes

Error Number	Message Name	Socket Type	Error Description	Programmer's Response
35	EWOULDBLOCK	Read Recvfrom	The socket is in nonblocking mode and read data is not available. This is not an error condition.	Issue a select on the socket to determine when data is available to be read or reissue the Read()/Recvfrom().
35	EWOULDBLOCK	Send Sendto Write	The socket is in nonblocking mode and buffers are not available.	Issue a select on the socket to determine when data is available to be written or reissue the Send(), Sendto(), or Write().
36	EINPROGRESS	Connect	The socket is marked nonblocking and the connection cannot be completed immediately. This is not an error condition.	See the Connect() description for possible responses.
37	EALREADY	Connect	The socket is marked nonblocking and the previous connection has not been completed.	Reissue Connect().
37	EALREADY	Maxdesc	A socket has already been created calling Maxdesc() or multiple calls to Maxdesc().	Issue Getablesize() to query it.
37	EALREADY	Setibmopt	A connection already exists to a TCP/IP image. A call to SETIBMOPT (IBMTCP_IMAGE), has already been made.	Only call Setibmopt() once.
38	ENOTSOCK	All	A socket operation was requested on a nonsocket connection. The value for socket descriptor was not valid.	Correct the socket descriptor value and reissue the function call.
39	EDESTADDRREQ	All	A destination address is required.	Fill in the destination field in the correct parameter and reissue the function call.
40	EMSGSIZE	Sendto Sendmsg Send Write	The message is too long. The default is 8192 and the maximum is 32,767. The LARGEENVELOPEPOOLSIZE statement in PROFILE.TCPIP may restrict this value.	Either correct the length parameter, or send the message in smaller pieces.
41	EPROTOTYPE	All	The specified protocol type is incorrect for this socket.	Correct the protocol type parameter.
42	ENOPROTOOPT	Getsockopt Setsockopt	The socket option specified is incorrect or the level is not SOL_SOCKET. Either the level or the specified optname is not supported.	Correct the level or optname.
42	ENOPROTOOPT	Getibmsockopt Setibmsockopt	Either the level or the specified optname is not supported.	Correct the level or optname.
43	EPROTONOSUPPORT	Socket	The specified protocol is not supported.	Correct the protocol parameter.
44	ESOCKTNOSUPPORT	All	The specified socket type is not supported.	Correct the socket type parameter.
45	EOPNOTSUPP	Accept Givesocket	The selected socket is not a stream socket.	Use a valid socket.
45	EOPNOTSUPP	Listen	The socket does not support the Listen call.	Change the type on the Socket() call when the socket was created. Listen() only supports a socket type of SOCK_STREAM.

Table 4 (Page 4 of 9). System Error Return Codes

Error Number	Message Name	Socket Type	Error Description	Programmer's Response
45	EOPNOTSUPP	Getibmopt Setibmopt	The socket does not support this function call. This command is not supported for this function.	Correct the command parameter. See Getibmopt() for valid commands. Correct by ensuring a Listen() was not issued before the Connect().
46	EPFNOSUPPORT	All	The specified protocol family is not supported or the specified domain for the client identifier is not AF_INET=2.	Correct the protocol family.
47	EAFNOSUPPORT	Bind Connect Socket	The specified address family is not supported by this protocol family.	For Socket() , set the domain parameter to AF_INET. For Bind() and Connect(), set Sin_Family in the socket address structure to AF_INET.
47	EAFNOSUPPORT	Getclient Givesocket	The socket specified by the socket descriptor parameter was not created in the AF_INET domain.	The Socket() call used to create the socket should be changed to use AF_INET for the domain parameter.
48	EADDRINUSE	Bind	The address is in a timed wait because a LINGER delay from a previous close or another process is using the address.	If you want to reuse the same address, use Setsockopt() with SO_REUSEADDR. See Setsockopt(). Otherwise, use a different address or port in the socket address structure.
49	EADDRNOTAVAIL	Bind	The specified address is incorrect for this host.	Correct the function address parameter.
49	EADDRNOTAVAIL	Connect	The calling host cannot reach the specified destination.	Correct the function address parameter.
50	ENETDOWN	All	The network is down.	Retry when the connection path is up.
51	ENETUNREACH	Connect	The network cannot be reached.	Ensure that the target application is active.
52	ENETRESET	All	The network dropped a connection on a reset.	Reestablish the connection between the applications.
53	ECONNABORTED	All	The software caused a connection abend.	Reestablish the connection between the applications.
54	ECONNRESET	All	The connection to the destination host is not available.	
54	ECONNRESET	Send Write	The connection to the destination host is not available.	The socket is closing. Issue Send() or Write() before closing the socket.
55	ENOBUFS	All	No buffer space is available.	Check the application for massive storage allocation call.
55	ENOBUFS	Accept	Not enough buffer space is available to create the new socket.	Check TCPIP.PROFILE buffer allocation statements.
55	ENOBUFS	Send Sendto Write	Not enough buffer space is available to send the new message.	Check TCPIP.PROFILE buffer allocation statements.
55	ENOBUFS	Takesocket	There is a socket control block (SCB) or socket interface control block (SKCB) shortage in the TCPIP address space.	Check TCPIP.PROFILE buffer allocation statements.
56	EISCONN	Connect	The socket is already connected.	Correct the socket descriptor on Connect() or do not issue a Connect() twice for the socket.

Table 4 (Page 5 of 9). System Error Return Codes

Error Number	Message Name	Socket Type	Error Description	Programmer's Response
57	ENOTCONN	All	The socket is not connected.	Connect the socket before communicating.
58	ESHUTDOWN	All	A Send cannot be processed after socket shutdown.	Issue read/receive before shutting down the read side of the socket.
59	ETOOMANYREFS	All	There are too many references. A splice cannot be completed.	Call your system administrator.
60	ETIMEDOUT	Connect	The connection timed out before it was completed.	Ensure the server application is available.
61	ECONNREFUSED	Connect	The requested connection was refused.	Ensure server application is available and at specified port.
62	ELOOP	All	There are too many symbolic loop levels.	Reduce symbolic links to specified file.
63	ENAMETOOLONG	All	The file name is too long.	Reduce size of specified file name.
64	EHOSTDOWN	All	The host is down.	Restart specified host.
65	EHOSTUNREACH	All	There is no route to the host.	Set up network path to specified host and verify that host name is valid.
66	ENOTEMPTY	All	The directory is not empty.	Clear out specified directory and reissue call.
67	EPROCLIM	All	There are too many processes in the system.	Decrease the number of processes or increase the process limit.
68	EUSERS	All	There are too many users on the system.	Decrease the number of users or increase the user limit.
69	EDQUOT	All	The disk quota has been exceeded.	Call your system administrator.
70	ESTALE	All	An old NFS** data set handle was found.	Call your system administrator.
71	EREMOTE	All	There are too many levels of remote in the path.	Call your system administrator.
72	ENOSTR	All	The device is not a stream device.	Call your system administrator.
73	ETIME	All	The timer has expired.	Increase timer values or reissue function.
74	ENOSR	All	There are no more stream resources.	Call your system administrator.
75	ENOMSG	All	There is no message of the desired type.	Call your system administrator.
76	EBADMSG	All	The system cannot read the message.	Verify that CS for OS/390 installation was successful and that message files were properly loaded.
77	EIDRM	All	The identifier has been removed.	Call your system administrator.
78	EDEADLK	All	A deadlock condition has occurred.	Call your system administrator.
78	EDEADLK	Select Selectex	None of the sockets in the socket desriptor sets is either AF_NET or AF_IUCV sockets and there is not timeout or no ECB specified. The select/selectex would never complete.	Correct the socket descriptor sets so that a AF_NET or AF_IUCV socket is specified. A timeout or ECB value can also be added to avoid the select/selectex from waiting indefinitely.

Table 4 (Page 6 of 9). System Error Return Codes

Error Number	Message Name	Socket Type	Error Description	Programmer's Response
79	ENOLCK	All	No record locks are available.	Call your system administrator.
80	ENONET	All	The requested machine is not on the network.	Call your system administrator.
81	ERREMOTE	All	The object is remote.	Call your system administrator.
82	ENOLINK	All	The link has been severed.	Release the sockets and reinitialize the client-server connection.
83	EADV	All	An ADVERTISE error has occurred.	Call your system administrator.
84	ESRMNT	All	An SRMOUNT error has occurred.	Call your system administrator.
85	ECOMM	All	A communication error has occurred on a Send call.	Call your system administrator.
86	EPROTO	All	A protocol error has occurred.	Call your system administrator.
87	EMULTIHOP	All	A multihop address link was attempted.	Call your system administrator.
88	EDOTDOT	All	A cross-mount point was detected. This is not an error.	Call your system administrator.
89	EREMCHG	All	The remote address has changed.	Call your system administrator.
90	ECONNCLOSED	All	The connection was closed by a peer.	Check that the peer is running.
113	EBADF	All	Socket descriptor is not in correct range. The maximum number of socket descriptors is set by MAXDESC(). The default range is 0 through 49.	Reissue function with corrected socket descriptor.
113	EBADF	Bind socket	The socket descriptor is already being used.	Correct the socket descriptor.
113	EBADF	Givesocket	The socket has already been given. The socket domain is not AF_INET.	Correct the socket descriptor.
113	EBADF	Select	One of the specified descriptor sets is an incorrect socket descriptor.	Correct the socket descriptor. Set on Select() or Selectex().
113	EBADF	Takesocket	The socket has already been taken.	Correct the socket descriptor.
113	EBADF	Accept	A Listen() has not been issued before the Accept().	Issue Listen() before Accept().
121	EINVAL	All	An incorrect argument was specified.	Check and correct all function parameters.
145	E2BIG	All	The argument list is too long.	Eliminate excessive number of arguments.
1000	EIBMBADCALL	All	An incorrect socket-call constant was found in the IUCV header.	For AF_IUCV sockets, contact your system administrator.
1001	EIBMBADPARM	All	An IUCV header error, type OTHER, has occurred.	For AF_IUCV sockets, contact your system administrator.
1002	EIBMSOCKOUTOFRANGE	Socket	A socket number assigned by the client interface code is out of range.	Check the socket descriptor parameter.

Table 4 (Page 7 of 9). System Error Return Codes

Error Number	Message Name	Socket Type	Error Description	Programmer's Response
1003	EIBMSOCKINUSE	Socket	A socket number assigned by the client interface code is already in use.	Use a different socket descriptor.
1004	EIBMIUCVERR	All	The request failed because of an IUCV error. This error is generated by the client stub code.	Ensure IUCV/VMCF is functional.
1005	EOFFLOADBOXERROR	All	The Offload host encountered an error.	Ensure offload box is functional.
1006	EOFFLOADBOXRESET	All	The Offload host was restarted.	Ensure offload box is functional.
1007	EOFFLOADBOXDOWN	All	The Offload host went down.	Ensure offload box is functional.
1008	EIBMCONFLICT	All	This request conflicts with a request already queued on the same socket.	Cancel the existing call or wait for its completion before reissuing this call.
1009	EIBMCANCELLED	All	The request was cancelled by the CANCEL call.	Informational, no action needed.
1010	EIBMTHREADFAIL	All	Returned by the offload function when a beginthread failure occurs.	Ensure offload box is functional.
1011	EIBMBADTCPNAME	All	A TCP/IP name that is not valid was detected.	Correct the name specified in the IBM_TCPIMAGE structure.
1012	EIBMBADREQUESTCODE	All	A request code that is not valid was detected.	Contact your system administrator.
1013	EIBMBADCONNECTIONSTATE	All	A connection token that is not valid was detected; bad state.	Verify TCP/IP is active.
1014	EIBMUNAUTHORIZEDCALLER	All	An unauthorized caller specified an authorized keyword.	Ensure user ID has authority for the specified operation.
1015	EIBMBADCONNECTIONMATCH	All	A connection token that is not valid was detected. There is no such connection.	Verify TCP/IP is active.
1016	EIBMTCPABEND	All	An abend occurred when TCP/IP was processing this request.	Verify that TCP/IP has restarted.
1017	EIBMBADMSGID	All	A message ID that is not valid was detected.	Verify that TCP/IP is functional.
1018	EIBMNOCCB	All	A CCB could not be obtained for this request.	Verify that TCP/IP is functional.
1019	EIBMNORESPONSEFOUND	All	No response was found for the request issued previously.	Reissue the function call.
1020	EIBMINVALIDRESPONSE	All	The expected response was not received.	Reissue the function call.
1021	EIBMNOACB	All	An ACB could not be obtained for this request.	Ensure that TCP/IP is functional.
1022	EIBMINVALIDKEYWORD	All	A keyword combination that is not valid was used.	Check validity of function parameters.
1023	EIBMTERMERROR	All	A terminating error was encountered.	Call your system programmer.
1024	EIBMNOADDRSPACENAME	All	An API common module could not find the user address space name.	Call your system programmer.
1025	EIBMSRBMODE	All	A call was issued in SRB mode.	Reissue in authorized state.
1026	EIBMINVDELETE	All	Delete requestor did not create the connection.	Delete the request from the process that created it.

Table 4 (Page 8 of 9). System Error Return Codes

Error Number	Message Name	Socket Type	Error Description	Programmer's Response
1027	EIBMINVSOCKET	All	A connection token that is not valid was detected. No such socket exists.	Call your system programmer.
1028	EIBMINVTCPCONNECTION	All	Connection terminated by TCP/IP. The token was invalidated by TCP/IP.	Reestablish the connection to TCP/IP.
1029	EIBMBADATTACH	All	Task failed to attach.	Check ATTACH failure reasons, correct, and reissue the call.
1030	EIBMINVGETSTORAGE	All	Failed to obtain storage.	Correct any parameters on the call that would cause allocation of buffers to fail.
1031	EIBMINVFREESTORAGE	All	Failed to free storage.	Correct any parameters on the call that would cause allocation of buffers to fail.
1032	EIBMCALLINPROGRESS	All	Another call was already in progress.	Reissue after previous call has completed.
1033	EIBMCELLPOOLDELETED	All	Cell pool is marked for deletion.	Call your system programmer.
1034	EIBMINVSOCKETCALLNUM	All	The request exceeded the maximum number of requests allowed.	Check use of EZASMI MF=L to build unique parameter storage areas per outstanding request.
1035	EIBMERRORUNKNOWN	All	An unknown reason code was received.	Call your system programmer.
1036	EIBMNOACTIVETCP	All	TCP/IP is not installed or not active.	Correct TCP/IP name used.
1036	EIBMNOACTIVETCP	Select	EIBMNOACTIVETCP	Ensure TCP/IP is active.
1036	EIBMNOACTIVETCP	Getibmopt	No TCP/IP image was found.	Ensure TCP/IP is active.
1037	EIBMINVTSRBUSERDATA	All	The request control block contained data that is not valid.	Call your system programmer.
1038	EIBMINVUSERDATA	All	The request control block contained user data that is not valid.	Check your function parameters and call your system programmer.
1039	EIBMBADTCPIP	All	The client program was cancelled because TCP/IP is coming down.	Wait until TCP/IP is active again and restart the program.
1040	EIBMSELECTEXPOST	SELECTEX	SELECTEX passed an ECB that was already posted.	Check whether the user's ECB was already posted.
1041	EIBMBADPOSTCODE	All	An application was given a bad post code. Received by an outstanding (blocked) socket when TCP/IP is stopped.	Ensure that TCP/IP is active.
1042	EIBMDUPJOBNAME	INITAPI	An application attempted to create a connection to TCP/IP with a duplicate jobname and/or subtask name.	Correct jobname or subtask name.
1043	EIBMCLIENVRCVY	All	TCP/IP was reinitialized while an application was communicating with it. Recovery processing deleted the previous TCP/IP environmental information for the application. Subsequent INITAPI socket calls will succeed.	Reinitialize connection to TCP/IP and restart program.
1044	EIBMINVALIDTCB	All except INITAPI	Socket call was issued from a TCB other than the TCB that issued the INITAPI call.	Issue the function call from the correct task

Table 4 (Page 9 of 9). System Error Return Codes

Error Number	Message Name	Socket Type	Error Description	Programmer's Response
2001	EINVALIDRXSOCKETCALL	REXX	A syntax error occurred in the RXSOCKET parameter list.	Correct the parameter list passed to the REXX socket call.
2002	ECONSOLEINTERRUPT	REXX	A console interrupt occurred.	Retry the task.
2003	ESUBTASKINVALID	REXX	The subtask ID is incorrect.	Correct the subtask ID on the INITIALIZE call.
2004	ESUBTASKALREADYACTIVE	REXX	The subtask is already active.	Only issue the INITIALIZE call once in your program.
2005	ESUBTASKALNOTACTIVE	REXX	The subtask is not active.	Issue the INITIALIZE call before any other socket call.
2006	ESOCKNETNOTALLOCATED	REXX	The specified socket could not be allocated.	Increase the user storage allocation for this job.
2007	EMAXSOCKETSREACHED	REXX	The maximum number of sockets has been reached.	Increase the number of allocate sockets, or decreased the number of sockets used by your program.
2009	ESOCKETNOTDEFINED	REXX	The socket is not defined.	Issue the SOCKET call before the call that fails.
2011	EDOMAINSERVERFAILURE	REXX	A Domain Name Server failure occurred.	Call your MVS system programmer.
2012	EINVALIDNAME	REXX	An incorrect *name* was received from the TCP/IP server.	Call your MVS system programmer.
2013	EINVALIDCLIENTID	REXX	An incorrect *clientid* was received from the TCP/IP server.	Call your MVS system programmer.
2014	ENIVALIDFILENAME	REXX	An error occurred during NUCEXT processing.	Specify the correct translation table file name, or verify that the translation table is valid.
2016	EHOSTNOTFOUND	The host is not found.	REXX	Call your MVS system programmer.
2017	EIPADDRNOTFOUND	Address not found.	REXX	Call your MVS system programmer.

Sockets Extended Return Codes

This section covers the error condition codes that are returned by the sockets extended interface. These errors are due to a problem detected by the sockets extended API.

Table 5 (Page 1 of 9). Sockets Extended Return Codes

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. Insufficient TCP/IP storage available.	End the call.	Increase MVS storage in the TCP/IP address space.
10102	Interface could not find the first subsystem.	End the call.	Call your MVS system programmer.
10103	VMCF or IUCV is not available. The VMCF CVT address is 0.	End the call.	Check VMCF or IUCV specification and IPL MVS.
10104	The requested function is not valid.	End the call.	Correct the call and retry.

Table 5 (Page 2 of 9). Sockets Extended Return Codes

Error Code	Problem Description	System Action	Programmer's Response
10105	The number of parameters is incorrect or the return code address is negative.	End the call.	Correct the parameter list.
10106	The high-order bit in the parameter list is not set for the last parameter.	End the call.	Correct the parameter list.
10107	An incorrect subtask ID was obtained from a GETCLIENTID call.	End the call.	Correct the method used to get the subtask ID.
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.
10109	A duplicate subtask ID occurred in the path ownership table.	End the call.	Correct the subtask ID on INITAPI or TAKESOCKET.
10110	LOAD of EZASOH03 failed.	End the call.	Call the IBM Software Support Center.
10111	The IUCVMINI SET macro detected errors in IUCVMULT or EZACICMT.	End the call.	Call the IBM Software Support Center.
10112	The IUCVMCOM CONNECT macro detected errors.	End the call.	Start TCP/IP or, if TCP/IP is running, ensure that the startup name matches the TCP/IP job name. You might have used a duplicate job name.
10113	The IUCVMCOM SEND macro for the initial message detected errors.	End the call.	Call the IBM Software Support Center.
10114	During an IUCV CONNECT, IUCV sent an unsolicited message to an active task.	Disable the subtask and return an error to the user.	Retry the task.
10115	The option length for a GETSOCKOPT call is incorrect.	Return an error to the user.	Correct the GETSOCKOPT call.
10116	After a CONNECT, IUCV sent an unsolicited message to an active task.	Disable the interface.	Retry the task.
10117	The return code address in your parameter list is zero.	No action.	Correct the return code address.
10118	IUCV sent an unexpected interrupt.	Wait for TRUE to post the subtask.	Retry the task.
10119	TCP/IP severed the IUCV connection, IPUSER=IUCVCHECKRC is set.	End the call.	Call your system programmer.
10120	TCP/IP severed the IUCV connection, IPUSER=SHUTTINGDOWN is set.	End the call.	Change the program to handle the SHUTDOWN condition.
10121	TCP/IP severed the IUCV connection, IPUSER=BAD PATHID is set.	End the call.	Check the path ID.
10122	TCP/IP severed the IUCV connection, IPUSER=NULL SAVED NAME is set.	End the call.	Call the IBM Software Support Center.
10123	TCP/IP severed the IUCV connection, IPUSER=BAD INIT MSG LEN is set.	End the call.	If your program issues an INIT command (0), check the parameters.
10124	TCP/IP severed the IUCV connection, IPUSER=REQUIREDCONSTANT is set.	End the call.	Correct parameter 3 in your INIT call.

Table 5 (Page 3 of 9). Sockets Extended Return Codes

Error Code	Problem Description	System Action	Programmer's Response
10125	TCP/IP severed the IUCV connection, IPUSER=BAD API TYPE is set.	End the call.	Correct parameter 5 in your INIT call.
10126	TCP/IP severed the IUCV connection, IPUSER=RESTRICTED is set.	End the call.	Call your system programmer.
10127	TCP/IP severed the IUCV connection, IPUSER=NO MORE CCBS is set.	End the call.	Call your system programmer.
10128	TCP/IP severed the IUCV connection, IPUSER=NO CCB is set.	End the call.	Call your system programmer.
10129	TCP/IP severed the IUCV connection, IPUSER=KILL nnn is set.	End the call.	Call your system programmer.
10130	TCP/IP severed the IUCV connection, IPUSER=UNKNOWN REASON CODE is set.	End the call.	Call your system programmer.
10131	During a quiesce, IUCV sent an unsolicited message to an active task.	End the call.	Call your system programmer.
10132	During a quiesce, IUCV sent an unsolicited message to an inactive task.	Wait for the TRUE to post the subtask.	Call your system programmer.
10133	IUCV sent an unsolicited priority completion message interrupt to an active task.	End the call.	Call your system programmer.
10134	IUCV sent an unsolicited priority completion message interrupt to an inactive task.	Wait for TRUE to post the subtask.	Call your system programmer.
10135	An error occurred in an IUCV SEVER function.	End the call.	Call your system programmer.
10136	IUCV sent an unsolicited nonpriority completion message interrupt to an inactive task.	Wait for TRUE to post the subtask.	Call your system programmer.
10137	IUCV sent an unsolicited priority pending message interrupt to an active task.	End the call.	Call your system programmer.
10138	IUCV sent an unsolicited priority pending message interrupt to an inactive task.	Wait for TRUE to post the subtask.	Call your system programmer.
10139	IUCV sent an unsolicited external interrupt to the subtask.	Write message EZY1281E to the system console. Return an error code to the caller.	Call your system programmer.
10140	IUCV sent a solicited nonpriority pending message interrupt to the subtask.	Return an error code to the caller.	Call your system programmer.
10141	IUCV sent an unsolicited, nonpriority pending message interrupt to the subtask.	Wait for TRUE to post the subtask.	Call your system programmer.
10142	Errors were found in the parameter list for an ACCEPT call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the ACCEPT call. You might have incorrect sequencing of socket calls.

Table 5 (Page 4 of 9). Sockets Extended Return Codes

Error Code	Problem Description	System Action	Programmer's Response
10143	Errors were found in the parameter list for a BIND call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the BIND call. You might have incorrect sequencing of socket calls.
10144	Errors were found in the parameter list for a CLOSE call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the CLOSE call. You might have incorrect sequencing of socket calls.
10145	Errors were found in the parameter list for a CONNECT call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the CONNECT call. You might have incorrect sequencing of socket calls.
10146	Errors were found in the parameter list for an FCNTL call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the FCNTL call. You might have incorrect sequencing of socket calls.
10147	Errors were found in the parameter list for a GETCLIENTID call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the GETCLIENTID call. You might have incorrect sequencing of socket calls.
10148	Errors were found in the parameter list for a GETHOSTID call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the GETHOSTID call. You might have incorrect sequencing of socket calls.
10149	Errors were found in the parameter list for a GETHOSTNAME call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the GETHOSTNAME call. You might have incorrect sequencing of socket calls.
10150	Errors were found in the parameter list for a GETPEERNAME call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the GETPEERNAME call. You might have incorrect sequencing of socket calls.
10151	Errors were found in the parameter list for a GETSOCKNAME call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the GETSOCKNAME call. You might have incorrect sequencing of socket calls.
10152	Errors were found in the parameter list for a GETSOCKOPT call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the GETSOCKOPT call. You might have incorrect sequencing of socket calls.
10153	Errors were found in the parameter list for a GIVESOCKET call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the GIVESOCKET call. You might have incorrect sequencing of socket calls.
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.

Table 5 (Page 5 of 9). Sockets Extended Return Codes

Error Code	Problem Description	System Action	Programmer's Response
10157	The parameter list for an IOCTL call is incorrect.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the IOCTL call. You might have incorrect sequencing of socket calls.
10158	The parameter list for a LISTEN call is incorrect.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the LISTEN 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.
10160	A READ call specified a data length greater than 32KB.	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.
10162	The parameter list for a READ call is incorrect.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the READ call. You might have incorrect sequencing of socket calls.
10163	A 0 or negative data length was found for a RECV, RECVFROM, or RECVMSG call.	Disable the subtask for interrupts. Sever the IUCV path. Return an error code to the caller.	Correct the data length.
10164	The data length for a RECV, RECVFROM, or RECVMSG call exceeded the maximum.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the data length.
10165	Incorrect data was specified for an IUCV SEVER.	End the call.	Call the IBM Software Support Center.
10166	The parameter list for a RECV, RECVFROM, or RECVMSG call is incorrect.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the READ call. You might have incorrect sequencing of socket calls.
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.
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.
10169	The parameter list for a SELECT or SELECTEX call is incorrect.	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.
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.

Table 5 (Page 6 of 9). Sockets Extended Return Codes

Error Code	Problem Description	System Action	Programmer's Response
10171	The data length for a SEND call exceeded the maximum.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the data length in the SEND call.
10172	An error occurred while processing an IUCV SEVER command.	End the program.	Call your system programmer.
10173	The parameter list for a SEND call is incorrect.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the SEND call. You might have incorrect sequencing of socket calls.
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.
10175	The data length for a SENDTO call exceeded the maximum.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the data length in the SENDTO call.
10176	A nonzero return code was received from an IUCV SEVER command.	Return an error to the user.	Call your system programmer.
10177	The parameter list for a SENDTO call is incorrect.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the SENDTO call. You might have incorrect sequencing of socket calls.
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.
10180	The parameter list for a SETSOCKOPT call is incorrect.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the SETSOCKOPT call. You might have incorrect sequencing of socket calls.
10181	The parameter list for a SHUTDOWN call is incorrect.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the SHUTDOWN call. You might have incorrect sequencing of socket calls.
10182	The parameter list for a SOCKET call is incorrect.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the SOCKET call. You might have incorrect sequencing of socket calls.
10183	The parameter list for a TAKESOCKET call is incorrect.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the TAKESOCKET call. You might have incorrect sequencing of socket calls.
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.

Table 5 (Page 7 of 9). Sockets Extended Return Codes

Error Code	Problem Description	System Action	Programmer's Response
10185	The data length for a WRITE call exceeded the maximum length.	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.
10187	The third parameter for IUCV SEVER ALL is incorrect.	End the call.	Call the IBM Software Support Center.
10188	Errors were found in the parameter list for a WRITE call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the WRITE call. You might have incorrect sequencing of socket calls.
10189	Errors were found in the parameter list for a LASTERRNO call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the LASTERRNO call. You might have incorrect sequencing of socket calls.
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.
10191	IUCV returned an error code.	End the call.	Notify operations.
10192	The function is available.	No action.	No action.
10193	The GETSOCKOPT option length is less than the minimum or greater than the maximum length.	End the call.	Correct the length parameter.
10194	The IUCV SEVER ALL user data is not binary zeros.	Bypass the call.	Call the IBM Software Support Center.
10195	An IUCV SEVER path ID error was detected.	Bypass the call.	Call the IBM Software Support Center.
10196	An error occurred in the IUCV CLEAR function.	Bypass the call.	Call the IBM Software Support Center.
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.
10199	EZACICMT cannot be loaded.	Bypass the call.	Correct the SYSLIB concatenation and retry.
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 genral programming .
10201	The seventh parameter, ERRNO address, for an IOCTL call is incorrect.	End the call.	Correct the parameter list.
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.
10204	The requested socket number exceeds 4095.	End the call.	Correct the requested socket number.

Table 5 (Page 8 of 9). Sockets Extended Return Codes

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.
10206	The socket descriptor table is full.	End the call.	Issue an INITAPI call to request more descriptors.
10207	A SYNC call was issued before an ECB was specified.	End the call.	Issue an ECB before the SYNC call.
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.
10216	TCP/IP has terminated.	End the call.	Perform recovery processing for a TCP/IP failure.
10217	The connection to TCP/IP has been severed.	End the call.	Perform recovery processing for the connection. This will include re-establishing communication with TCP/IP.
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.
10220	Trying to close socket that has not been allocated.	End the call.	Correct the S parameter on the CLOSE 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 5 (Page 9 of 9). Sockets Extended Return Codes

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.
10228	The CANCEL call was issued on a non-asynchronous connection.	End the call.	CANCEL is valid only for connections established with the ASYNC parameter on the INITAPI call.
10228	For CICS, the maximum number of sockets specified for an INITAPI exceeds 255.	End the call.	Correct the INITAPI 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.
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.
10333	An error was detected in the asynchronous exit routine.	End the call.	Perform necessary asynchronous exit recovery where applicable.
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:

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

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

Parameters

The following types of parameters are used in syntax diagrams.

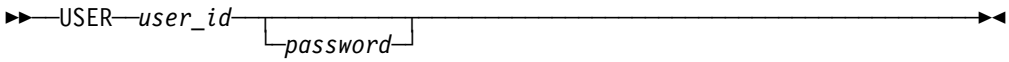
Parameter	Description
Required	Required parameters are displayed on the main path.
Optional	Optional parameters are displayed below the main path.
Default	Default parameters are displayed above the main path.

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.

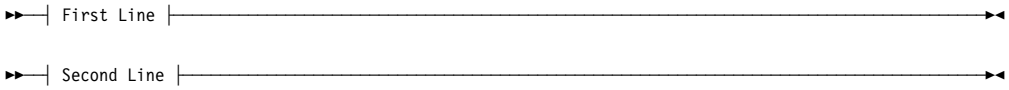
Variables are italicized, appear in lowercase letters, and represent names or values you supply. For example, a data set is a variable.

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.



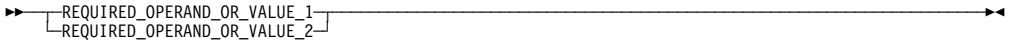
Longer than one line: If a diagram is longer than one line, the first line ends with a single arrowhead and the second line begins with a single arrowhead.



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.



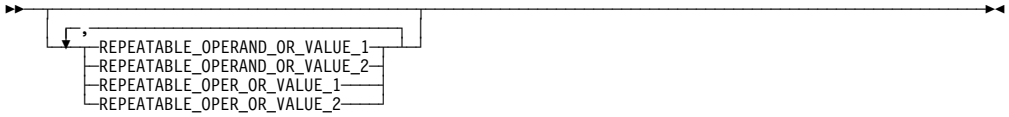
Optional values: Optional operands and values appear below the main path line.

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 commad 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.

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:

Appendix C. Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make them available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter 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
500 Columbus Avenue
Thornwood, NY 10594
USA

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement.

This document is not intended for production use and is furnished as is without any warranty of any kind, and all warranties are hereby disclaimed including the warranties of merchantability and fitness for a particular purpose.

IBM is required to include the following statements in order to distribute portions of this document and the software described herein to which contributions have been made by The University of California.

Portions herein © Copyright 1979, 1980, 1983, 1986, Regents of the University of California. Reproduced by permission. Portions herein were developed at the Electrical Engineering and Computer Sciences Department at the Berkeley campus of the University of California under the auspices of the Regents of the University of California.

Portions of this publication relating to RPC are Copyright © Sun Microsystems, Inc., 1988, 1989.

Some portions of this publication relating to X Window System** are Copyright © 1987, 1988 by Digital Equipment Corporation, Maynard, Massachusetts, and the Massachusetts Institute Of Technology, Cambridge, Massachusetts. All Rights Reserved.

Some portions of this publication relating to X Window System are Copyright © 1986, 1987, 1988 by Hewlett-Packard Corporation.

Permission to use, copy, modify, and distribute the M.I.T., Digital Equipment Corporation, and Hewlett-Packard Corporation portions of this software and its documentation for any purpose without fee is hereby granted, provided that the above copyright notice appears in all copies and that both that copyright notice and this

permission notice appear in supporting documentation, and that the names of M.I.T., Digital, and Hewlett-Packard not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. M.I.T., Digital, and Hewlett-Packard make no representation about the suitability of this software for any purpose. It is provided “as is” without express or implied warranty.

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
AD/Cycle
AIX
AIX/ESA
AnyNet
APPN
AS/400
BookManager
C/370
CICS
DB2
DFSMS
DFSMS/MVS
ESCON
ES/9000
ES/9370
EtherStreamer
Extended Services
FFST
GDDM
Hardware Configuration Definition
IBM
LANStreamer
Library Reader
MVS/ESA
MVS/SP
MVS/XA
NetView
Nways
OpenEdition
OS/2
OS/390
PS/2
RACF
RETAIN
RISC System/6000
RS/6000
SAA
System/360
System/370
System/390
VTAM
3090

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 and other countries licensed exclusively through X/Open Company Limited.

Microsoft, Windows, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Glossary

The IBM Networking Software Glossary is now available in HTML format as well as PDF. You can access it directly at the following URL:

`http://www.networking.ibm.com/nsg/nsggls.htm`

This glossary includes terms and definitions from:

- The *American National Standards Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42nd Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.
- The ANSI/EIA Standard—440-A, *Fiber Optic Terminology* Copies may be purchased from the Electronic Industries Association, 2001 Pennsylvania Avenue, N.W., Washington, DC 20006. Definitions are identified by the symbol (E) after the definition.
- The *Information Technology Vocabulary* developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.
- The *IBM Dictionary of Computing*, New York: McGraw-Hill, 1994.
- Internet Request for Comments: 1208, *Glossary of Networking Terms*
- Internet Request for Comments: 1392, *Internet Users' Glossary*
- The *Object-Oriented Interface Design: IBM Common User Access Guidelines*, Carmel, Indiana: Que, 1992.

Bibliography

eNetwork Communications Server for OS/390 V2R5 Publications

Following are descriptions of the books in the eNetwork Communications Server for OS/390 V2R5 library. The books are arranged in the following categories:

- Softcopy Information
- Marketing Information
- Planning
- Installation, Resource Definition, and Tuning
- Operation
- Customization
- Writing Application Programs
- Diagnosis
- Messages and Codes
- APPC Application Suite.
- Multiprotocol Transport Networking (MPTN) Architecture publications

The complete set of unlicensed books in this section can be ordered using a single order number, SBOF-7011.

Softcopy Information

IBM Networking Softcopy Collection Kit CD-ROM(SK2T-6012).

The softcopy library contains softcopy versions of the licensed and unlicensed books for eNetwork Communications Server for OS/390 V2R5.

All of the unlicensed and licensed books described in this section are available in softcopy on this CD-ROM. These softcopy files can be read using any of the IBM BookManager READ programs. They can also be read with the IBM Library Reader program shipped on this CD.

The CD also contains softcopy of the unlicensed books of many other products.

Marketing Information

A Networking Overview and the following IBM Networking Previews are available:

- VTAM
- TCP/IP

Ask your IBM marketing representative for more information.

Planning

OS/390 eNetwork Communications Server: SNA Planning and Migration Guide (SC31-8622). This guide helps you upgrade to eNetwork Communications Server for OS/390 V2R5. It includes:

- Installation procedures
- Planning to upgrade
  - Upward and downward compatibility
  - Software and hardware requirements
  - Storage requirements
  - Impacts of new functions and enhancements performed without changes to user interfaces
  - Changes to installation process
- Upgrading user interfaces
  - Changes to start options
  - Changes to buffer pools
  - Changes to definition statements
  - Changes to IBM-supplied default user-definable tables and modules
  - Changes to user-definable table macroinstructions
  - Changes to commands
  - Changes to messages
  - Changes to SNA application programming interface
  - Changes to installation-wide exit routines
  - Changes to control blocks
- Implementing optional functions and enhancements introduced in eNetwork Communications Server for OS/390 V2R5.
  - Overview of each new function and enhancement introduced since VTAM V4R4
  - Pointers to other books in the library where implementation details can be found.

OS/390 eNetwork Communications Server: IP Planning and Migration Guide (SC31-8512). This book is intended to help you plan for TCP/IP whether you are migrating from a previous version or installing TCP/IP for the first time. This book also identifies the suggested and required modifications needed to enable you to use the enhanced functions provided with TCP/IP.

Installation, Resource Definition, Configuration, and Tuning

Program Directory. These documents are shipped with the product tape and explains the steps for installing VTAM and TCP/IP.

OS/390 eNetwork Communications Server: IP Configuration (SC31-8513). This book is for people who want to configure, customize, administer, and maintain

TCP/IP. Familiarity with MVS operating system, TCP/IP protocols, and IBM Time Sharing Option (TSO) is recommended.

OS/390 eNetwork Communications Server: SNA Network Implementation (SC31-8563). This book presents the major concepts involved in implementing a SNA network, and includes:

- Buffer pools, slowdown, pacing, storage considerations
- Implementation considerations
- Sample major node definitions
- Migration considerations
- Tables and filters
- TSO, VCNS, and other programs that run with VTAM
- Tuning procedures
- VTAM start options.

Use this book in conjunction with the *OS/390 eNetwork Communications Server: SNA Resource Definition Reference*

OS/390 eNetwork Communications Server: SNA Resource Definition Reference (SC31-8565). This book describes each VTAM definition statement, start option, and macroinstruction for user tables. It also describes NCP definition statements that affect VTAM. The information includes:

- IBM-supplied default tables (logon mode and USS)
- Major node definitions
- User-defined tables and filters
- VTAM start options.

If you are unfamiliar with the major concepts involved in implementing a SNA network, use this book in conjunction with the *OS/390 eNetwork Communications Server: SNA Network Implementation*.

OS/390 eNetwork Communications Server: SNA Resource Definition Samples (SC31-8566). This book contains sample definitions to help you implement VTAM functions in your networks, and includes sample major node definitions. Use this book in conjunction with the *OS/390 eNetwork Communications Server: SNA Network Implementation* and *OS/390 eNetwork Communications Server: SNA Resource Definition Reference*

OS/390 eNetwork Communications Server: AnyNet SNA over TCP/IP (SC31-8578). This guide provides information to help you install, configure, use, and diagnose SNA over TCP/IP.

OS/390 eNetwork Communications Server: AnyNet Sockets over SNA (SC31-8577). 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

OS/390 eNetwork Communications Server: IP User's Guide (GC31-8514). This book is for people who want to use TCP/IP for data communication activities such as FTP and Telnet. Familiarity with MVS operating system and IBM Time Sharing Option (TSO) is recommended.

OS/390 eNetwork Communications Server: SNA Operation (SC31-8567). This book serves as a reference for programmers and operators requiring detailed information about specific operator commands. The information includes:

- VTAM commands and start options
- Logon manager commands
- DISPLAY output examples (messages received)
- VSCS commands.

OS/390 eNetwork Communications Server: Operations Quick Reference (SX75-0121). This book contains essential information about VTAM operator commands.

High Speed Access Services User's Guide (GC31-8676).

Customization

OS/390 eNetwork Communications Server: SNA Customization (LY43-0110). This book enables you to customize VTAM, and includes:

- Communication network management (CNM) routing table
- Logon-interpret routine requirements
- Logon manager installation-wide exit routine for the CLU search exit
- TSO/VTAM installation-wide exit routines
- VTAM installation-wide exit routines:
  - Command verification exit (ISTCMMND)
  - Configuration services XID exit (ISTEXCCS) with description of IBM-supplied default exit
  - Directory services management exit (ISTEXCDM)
  - Generic resource resolution exit (ISTEXCGR)
  - Performance monitor exit (ISTEXCPM)
  - SDDLU exit (ISTEXCSD) with description of IBM-supplied default exit
  - Session accounting exit (ISTAUCAG)
  - Session authorization exit (ISTAUCAT)
  - Session management exit (ISTEXCAA) with example
  - TPRINT processing exit (ISTRAEUE)

  - USERVAR exit (ISTEXCUV) with description of IBM-supplied default exit
  - Virtual route pacing window size calculation exit (ISTPUCWC)
  - Virtual route selection exit (ISTEXCVR).

OS/390 eNetwork Communications Server: IP Network Print Facility (SC31-8522). This book is for system programmers and network administrators who need to prepare their network to route VTAM, JES2, or JES3 printer output to remote printers using TCP/IP.

Writing Application Programs

OS/390 eNetwork Communications Server: IP API Guide (SC31-8516). 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.

OS/390 eNetwork Communications Server: IP CICS Sockets Guide (SC31-8521). This book is for people who want to set up, write application programs for, and diagnose problems with the socket interface for CICS using TCP/IP for MVS.

OS/390 eNetwork Communications Server: IP IMS Sockets Guide (SC31-8546). This book is for programmers who want application programs that use the IMS TCP/IP application development services provided by IBM TCP/IP for MVS.

OS/390 eNetwork Communications Server: IP Programmer's Reference (SC31-8515). 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 MVS operating system, TCP/IP protocols, and IBM Time Sharing Option (TSO) is recommended.

OS/390 eNetwork Communications Server: SNA Programming (SC31-8573). This book describes how to use VTAM 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. The information includes:

- API concepts
  - Cryptography
  - RUs and exchanges
  - Session establishment and termination
- BIND area format
- Communication Network Management Interface
- Dictionary of VTAM macroinstructions
- OPEN or CLOSE errors
- Operating system differences
- Program Operator Coding requirements
- RAPI DSECTs and control block mappings (ACB, ADSP, BLENT, CV29, EXLST, MTS, NIB, NIB DEVCHAR, NIB PROC, RH, RPL, RPL RTNCD=FDB2=FDBK=DSECT)
- RAPI global variables
- Vector lists
- RPL-based macroinstructions
- RPL RTNCD,FDB2 codes
- User exit routines.

OS/390 eNetwork Communications Server: SNA Programmers LU 6.2 Guide (SC31-8581). This book describes how to use the VTAM 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.) The information includes:

- VTAM's implementation of the LU 6.2 architecture
- Design considerations for LU 6.2 application programs
- Negotiating session limits with partner LUs
- BIND image and response
- Allocating and deallocating conversations
- FMH-5 and PIP data
- Conversation states
- Sending and receiving data
- Using high performance data transfer (HPDT)
- Session- and conversation-level security and data encryption
- Register usage
- Sync point services
- LU 6.2 global variables
- Vector lists
- Sense codes for FMH-7 and UNBIND
- RCPRI,RCSEC codes
- User exit routines.

OS/390 eNetwork Communications Server: SNA Programmers LU 6.2 Reference (SC31-8568). This book provides reference material for the VTAM LU 6.2 programming interface for host application programs. The information includes:

- APPCCMD macroinstructions
- Primary and secondary return codes (RCPRI, RCSEC)
- DSECTs
- Examples of using VTAM's LU 6.2 API
- Register usage

OS/390 eNetwork Communications Server: CSM Guide (SC31-8575). This book describes how applications use

the communications storage manager. The information includes:

- Creating and deleting buffer pools
- Obtaining and freeing buffers
- Return codes and reason codes
- DSECTs

OS/390 eNetwork Communications Server: CMIP Services and Topology Agent Guide (SC31-8576). 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 VTAM topology agent and includes the following topics:

- Management information base (MIB) API functions
- CMIP message strings
- Special CMIP message strings
- Read queue exit routine
- Sample CMIP application program
- VTAM resources as CMIP objects
- Naming conventions for objects
- VTAM resources and OSI states
- Attributes to object cross-reference
- ASN.1 syntax for CMIP messages
- GDMO table format
- ACYAPHDH header file.

Diagnosis

OS/390 eNetwork Communications Server: IP Diagnosis (SC31-8521). 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.

OS/390 eNetwork Communications Server: SNA Diagnosis (LY43-0079). This book helps you identify a VTAM 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. The information includes:

- Command syntax for running traces and collecting and analyzing dumps
- VIT entries
- Procedures for collecting documentation (VTAM, TSO)
- VTAM internal trace and VIT analysis tool
- FFST Probes
- Channel programs
- Flow diagrams
- Procedures for locating buffer pools
- CPCB operation codes
- Storage and control block ID codes
- Offset names and locations for VTAM buffer pools.

OS/390 eNetwork Communications Server: Data Areas Volume 1 (LY43-0111). This book describes VTAM data areas and can be used to read a VTAM dump. It is intended for IBM programming service representatives and customer personnel who are diagnosing problems with VTAM.

OS/390 eNetwork Communications Server: Data Areas Volume 2 (LY43-0112). This book describes VTAM data areas and can be used to read a VTAM dump. It is intended for IBM programming service representatives and customer personnel who are diagnosing problems with VTAM.

Messages and Codes

OS/390 eNetwork Communications Server: SNA Messages (SC31-8569). This book describes the following types of messages and other associated information:

- Messages:
  - ELM messages for logon manager
  - IKT messages for TSO/VTAM
  - IST messages for VTAM network operators
  - ISU messages for sockets-over-SNA
  - IVT messages for the communications storage manager
  - IUT messages
  - USS messages
- Other information that displays in VTAM messages:
  - Command and RU types in VTAM messages
  - Node and ID types in VTAM messages
- Supplemental message-related information:
  - Message additions, deletions, and changes
  - Message flooding prevention
  - Message groups and subgroups
  - Message routing and suppression including descriptor codes, routing codes, and suppression levels for ELM, IKT, IST, and ISU messages
  - Message text and description formats
  - Message text of MSGLVL option messages including general information on the MSGLVL option
  - Message text of all VTAM network operator messages including variable field lengths

OS/390 eNetwork Communications Server: IP Messages Volume 1 (SC31-8517). This volume contains TCP/IP messages beginning with EZA.

OS/390 eNetwork Communications Server: IP Messages Volume 2 (SC31-8570). This volume contains TCP/IP messages beginning with EZB.

OS/390 eNetwork Communications Server: IP Messages Volume 3 (SC31-8674). This volume contains TCP/IP messages beginning with EZY, EZZ, and SNM.

OS/390 eNetwork Communications Server: IP and SNA Codes (SC31-8571). This book describes codes and other information that display in CS/390 messages:

- Sense codes including VTAM sense code hints, SNA sense field values for RPL-based macroinstructions, and 3270 SNA and non-SNA device sense fields
- Return codes for macroinstructions including ACB OPEN and CLOSE macroinstruction error fields, RTNCD-FDB2 return code combinations, and LU 6.2 RCPRI-RCSEC return codes
- Data link control (DLC) status codes
- Status codes including resource status and session state codes
- Wait state event codes and IDs
- Abend codes
- ATM network-generated cause and diagnostic codes

APPC Application Suite

OS/390 eNetwork Communications Server: APPC Application Suite User's Guide (GC31-8619). 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.

OS/390 eNetwork Communications Server: APPC Application Suite Administration (SC31-8620). This book contains the information that administrators need to configure the APPC application suite and to manage the APING, ANAME, AFTP, and A3270 servers.

OS/390 eNetwork Communications Server: APPC Application Suite Programming (SC31-8621). This book provides the information application programmers need to add the functions of the AFTP and ANAME APIs to their application programs.

Multiprotocol Transport Networking (MPTN) Architecture Publications

Following are selected publications for MPTN:

Networking Blueprint Executive Overview (GC31-7057)

Multiprotocol Transport Networking: Technical Overview (GC31-7073)

Multiprotocol Transport Networking: Formats (GC31-7074)

OS/390 Publications

For information on OS/390 and other products, refer to *OS/390 Information Roadmap* (GC28-1727-03).

Index

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Communicating Your Comments to IBM

OS/390 eNetwork Communications Server
IP IMS Sockets Guide
Version 2 Release 5

Publication No. SC31-8519-00

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply.

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

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

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give the RCF to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:

  1-800-227-5088(US and Canada)
- If you prefer to send comments electronically, use this network ID:
  - USIB2HPD@VNET.IBM.COM
  - USIB2HPD at IBMMAIL

Make sure to include the following in your note:

- Title and publication number of this book
- Page number or topic to which your comment applies.

Readers' Comments — We'd Like to Hear from You

OS/390 eNetwork Communications Server
IP IMS Sockets Guide
Version 2 Release 5

Publication No. SC31-8519-00

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
SC31-8519-00

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
Information Development
Department CGMD / Bldg 500
P.O. Box 12195
Research Triangle Park, NC 27709-9990

Fold and Tape | **Please do not staple** | Fold and Tape

SC31-8519-00

Cut or Fold
Along Line

File Number: S390-50
Program Number: 5647-A01

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC31-8519-00

Spine information:

OS/390 eNetwork Communications Server IP IMS Sockets Guide *Version 2 Release 5*