OS/390 IBM Communications Server

IP Application Programming Interface Guide

Version 2 Release 10

SC31-8516-05

OS/390 IBM Communications Server

IP Application Programming Interface Guide

Version 2 Release 10

SC31-8516-05

Note:

Before using this information and the product it supports, be sure to read the general information under “Appendix H. Notices” on page 583.

Sixth Edition (December 2000)

This edition applies to OS/390 V2R10 (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
Software Reengineering
Department G7IA/ Bldg 503
Research Triangle Park, NC 27709-9990

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

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

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

World Wide Web:
http://www.ibm.com/s390/os390/

IBMLink:
CIBMORCF at RALVM17

IBM Mail Exchange:
tkinlaw@us.ibm.com

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

Contents

Figures

Tables

About This Book

This book describes the syntax of program source code necessary to write your application programming interface (API) into TCP/IP. You can use this interface as the communication base for writing your own client and server applications. You can also use this book to adapt your existing applications to communicate using TCP/IP over sockets.

To provide flexibility in writing new applications and adapting existing applications, the following programming languages and interfaces are described:

- C sockets
- X/Open Transport Interface (XTI)
- Assembler, PL/1, and COBOL sockets
- REXX sockets
- Pascal language

Please use the Reader Comment Form located at the back of this book for instructions about how to submit your comments by mail, fax, or electronically.

SecureWay® Communications Server for OS/390 V2R10 (CS for OS/390) is an integral part of the OS/390® V2R10 family of products. For an overview of, and mapping of the documentation available for OS/390 V2R10, refer to the *OS/390 Information Roadmap*.

Who Should Use This Book

This book is intended for experienced programmers familiar with MVS®, the IBM® multiple virtual storage operating system, TCP/IP protocols, UNIX® sockets, and data networks.

Before using this book, familiarize yourself with MVS and the IBM time sharing option (TSO).

CS for OS/390 and any required programming products should be installed and customized for your network.

Depending on the design and function of your application, you should be familiar with one or more of the following programming languages:

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

Where to Find More Information

"Bibliography" on page 589, describes the books in the IBM Communications Server for OS/390 library, arranged according to task. The bibliography also lists the titles and order numbers of books related to this book, or cited by name in this book.

Most licensed books were declassified in OS/390 V2R4 and are now included in the OS/390 Online Library Collection, SK2T-6700. The remaining licensed books appear in unencrypted BookManager® softcopy and PDF form on the OS/390 Licensed Product Library, LK2T-2499.

Where to Find Related Information on the Internet

You might find the following information helpful.

You can read more about VTAM, TCP/IP, OS/390, and IBM on these Web pages. For up-to-date information about Web addresses, please refer to informational APAR II11334.

Home Page	**Web address**
IBM Communications Server product	http://www.software.ibm.com/network/commserver/
IBM Communications Server support	http://www.software.ibm.com/network/commserver/support/
OS/390	http://www.ibm.com/s390/os390/
OS/390 Internet Library	http://www.ibm.com/s390/os390/bkserv/
IBM Systems Center publications	http://www.redbooks.ibm.com/
IBM Systems Center flashes	http://www-1.ibm.com/support/techdocs/atsmastr.nsf
VTAM and TCP/IP	http://www.software.ibm.com/network/commserver/about/csos390.html
IBM	http://www.ibm.com

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

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

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

How to Contact IBM Service

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

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

Summary of Changes

Summary of Changes
for SC31-8516-05
IBM Communications Server for OS/390 V2R10

The softcopy version of this book was refreshed for PDF optimization. There were no changes to the information contained in the book.

Summary of Changes
for SC31-8516-04
IBM Communications Server for OS/390 V2R10

This edition contains new and changed information, indicated by vertical lines in the left margin.

New Information

- ARP counter data differs depending on device type (see pages 139 and 265)
- The TCP/IP CTRACE trace option, SOCKAPI (see pages 224 and 332)
- The GETSOCKOPT and SETSOCKOPT calls fully support the SO_SNDBUF and SO_RCVBUF options (see pages 125, 126, 174, 175, 255, 255, 299, and 299)

Changed Information

- The term SecureWay® has been removed from our product name. The new product name is IBM Communications Server for OS/390.

Summary of Changes
for SC31-8516-03
IBM Communications Server for OS/390 V2R8

This edition contains new and changed information, indicated by vertical lines in the left margin.

New Information

- Information on the behavior of multiple concurrent read or write socket calls, which is helpful when considering what type of socket to use
- Description of two main sockets execution environments and available libraries
- Effects of a SHUTDOWN call when there are outstanding socket calls pending
- Environmental restrictions/programming requirements for Macro and CALL instruction APIs
- Information on how to define the address of task storage
- Information on how Task Management and Asychronous function processing work
- Asynchronous exit environmental and programming considerations
- More usage information on INITAPI
- More usage information on ALET
- Two new IOCTL calls
- OS/390 UNIX ERRNO table

Changed Information

- The term eNetwork™ is replaced by SecureWay as part of our product name. The new name is SecureWay Communications Server.
- The Bibliography has been revised to show book number dash levels and delivery format.
- Non-blocking calls and asynchronous calls
- Using a loopback address
- Coding guidelines for macro APIs
- Information on use of ECB/REQAREA in relation to APITYPE=3
- System error return codes, OS/390 unit return codes and sockets extended return codes
- GETSOCKOPT SO_CLUSTERCONNTYPE description

Summary of Changes for SC31-8516-02 eNetwork Communications Server for OS/390 V2R7

This book was updated to document functional enhancements provided in CS for OS/390 V2R7 as well as service enhancements.

The following information was new in this release:

- Coding guidelines added to the Macro API chapter
- New sockets options for GETSOCKOPT
- Coding guidelines for Macro API

The following information changed in this release:

- Updates to reflect that error EIBMUICVERR was removed
- MF=format section was removed
- Updated return code table
- Updates to reflect that High Performance Native Sockets (HPNS) is no longer supported

The following information was deleted in this release:

- SYNC macro
- SO_BULKMODE support

Note: As part of the name change of OpenEdition® to OS/390 UNIX System Services, occurrences of OS/390 OpenEdition have been changed to OS/390 UNIX System Services or its abbreviated name, OS/390 UNIX. OpenEdition may continue to appear in messages, panel text, and other code with OS/390 UNIX System Services.

New and changed information is indicated by a revision bar (|). The updates contained in this manual assume the latest level of maintenance has been applied.

Part 1. Overview

Chapter 1. Introducing TCP/IP Concepts

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

- TCP/IP concepts
- Understanding sockets concepts

TCP/IP Concepts

Conceptually, the TCP/IP protocol stack consists of four layers, each layer consisting of one or more protocols. A protocol is a set of rules or standards that two entities must follow so as to allow each other to receive and interpret messages sent to them. The entities could, for example, be two application programs in an application protocol, or the entities might be two TCP protocol layers in two different IP hosts (the TCP protocol).

Figure 1 illustrates the TCP/IP protocol stack.

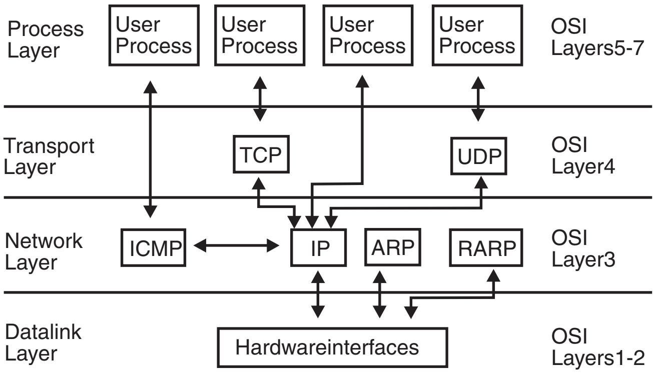


Figure 1. The TCP/IP Protocol Stack

Programs are located at the process layer; here they can interface with the two transport layer protocols (TCP and UDP), or directly with the network layer protocols (ICMP and IP).

TCP Transmission control protocol is a transport protocol providing a reliable, full-duplex byte stream. Most TCP/IP applications use the TCP transport protocol.

UDP User datagram protocol is a connectionless protocol providing datagram services. UDP is less reliable because there is no guarantee that a UDP datagram ever reaches its intended destination, or that it reaches its destination only once and in the same condition as it was passed to the sending UDP layer by a UDP application.

ICMP Internet control message protocol is used to handle error and control information at the IP layer. The ICMP is most often used by network control applications that are part of the TCP/IP software product itself, but ICMP can be used by authorized user processes as well. PING and TRACEROUTE are examples of network control applications that use the ICMP protocol.

IP Internet protocol provides the packet delivery services for TCP, UDP, and ICMP. The IP layer protocol is unreliable (called a best-effort protocol).

There is no guarantee that IP packets arrive, or that they arrive only once and error-free. Such reliability is built into the TCP protocol, but not into the UDP protocol. If you need reliable transport between two UDP applications, you must ensure that reliability is built into the UDP applications.

ARP The networking layer uses ARP to map an IP address into a hardware address. On local area networks (LANs), such an address would be called a media access control (MAC) address.

RARP Reverse address resolution protocol is used to reverse the operation of the ARP protocol: map a hardware address into an IP address. Note that both ARP packets and RARP packets are not forwarded in IP packets, but are themselves media level packets. ARP and RARP are not used on all network types, as some networks do not need these protocols.

Understanding Sockets Concepts

A socket uniquely identifies the endpoint of a communication link between two application ports.

A port represents an application process on a TCP/IP host, but the port number itself does not indicate the protocol being used: TCP, UDP, or IP. The application process might use the same port number for all three protocols. To uniquely identify the destination of an IP packet arriving over the network, you have to extend the port principle with information about the protocol used and the IP address of the network interface; this information is called a socket. A socket has three parts:

{*protocol, local-address, local-port*}

Figure 2 illustrates the concept of a socket.

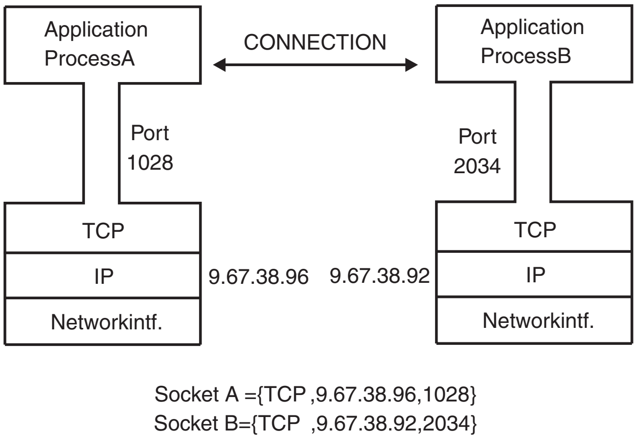


Figure 2. Socket Concept

The term *association* is used to specify completely the two processes that comprise a connection:

{*protocol,local-address,local-port,foreign-address,foreign-port*}

The terms *socket* and *port* are sometimes used as synonyms, but note that the terms *port number* and *socket address* are not like one another. A port number is one of the three parts of a socket address, and can be represented by a single number (for example, 1028) while a socket address can be represented by {tcp,myhostname,1028}.

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

Programming with Sockets

A socket is an endpoint for communication able to be named and addressed in a network. From the perspective of the application program, it is a resource allocated by the address space; it is represented by an integer called the socket descriptor.

The socket interface was designed to provide applications a network interface that hides the details of the physical network. The interface is differentiated by the different services provided: Stream, datagram, and raw sockets. Each interface defines a separate service available to applications.

The MVS socket APIs provide a standard interface using the transport and internetwork layer interfaces of TCP/IP. These APIs support three socket types: stream, datagram, and raw. Stream and datagram socket types interface with the transport layer protocols; raw socket types interface with the network layer protocols. Choose the most appropriate interface for your application.

Selecting Sockets

You can choose among the following types of sockets:

- Stream
- Datagram
- Raw

Stream sockets perform like streams of information. There are no record lengths or character boundaries between data, so communicating processes must agree on their own mechanisms for distinguishing information. Usually, the process sending information sends the length of the data, followed by the data itself. The process receiving information reads the length and then loops, accepting data until all of it has been transferred. Because there are no boundaries in the data, multiple concurrent read or write socket calls of the same type, on the same stream socket, will yield unpredictable results. For example, if two concurrent read socket calls are issued on the same stream socket, there is no guarantee of the order or amount of data that each instance will receive. Stream sockets guarantee to deliver data in the order sent and without duplication. The stream socket defines a reliable connection service. Data is sent without error or duplication and is received in the order sent. Flow control is built in to avoid data overruns. No boundaries are imposed on the data; the data is treated as a stream of bytes.

Stream sockets are most common because the burden of transferring the data reliably is handled by TCP/IP, rather than by the application.

The datagram socket is a connectionless service. Datagrams are sent as independent packets. The service provides no guarantees: data can be lost or duplicated, and datagrams can arrive out of order. The size of a datagram is limited to the size able to be sent in a single transaction. Currently, the default value is 8192 bytes, and the maximum value is 65,535. The maximum size of a datagram is 16384 for TCP and 65535 bytes for raw. No disassembly and reassembly of packets is performed.

The raw socket allows direct access to lower layer protocols, such as IP and the ICMP. This interface is often used to test new protocol implementation because the socket interface can be extended and new socket types defined to provide additional services. For example, the transaction type sockets can be defined for interfacing to the versatile message transfer protocol (VMTP). [1] Transaction-type sockets are not supported by TCP/IP. Because socket interfaces isolate you from the communication function of the different protocol layers, the interfaces are largely independent of the underlying network. In the MVS implementation of sockets, stream sockets interface with TCP, datagram sockets interface with UDP, and raw sockets interface with ICMP and IP.

Note: The TCP and UDP protocols cannot be used with raw sockets.

Note: If you are communicating with an existing application, you must use the same protocols used by the existing application. For example, if you communicate with an application that uses TCP, you must use stream sockets.

You should consider the following factors for these applications:

- Reliability

  Stream sockets provide the most reliable connection. Datagrams and raw sockets are unreliable because packets can be discarded, corrupted, or duplicated during transmission. This characteristic might be acceptable if the application does not require reliability, or if the application implements reliability beyond the socket interface.
- Performance

  Overhead associated with reliability, flow control, packet reassembly, and connection maintenance degrades the performance of stream sockets so that they do not perform as well as datagram sockets.
- Data Transfer

  Datagram sockets limit the amount of data moved in a single transaction. If you send fewer than 2048 bytes of data at one time, use datagram sockets. When the amount of data in a single transaction is greater, use stream sockets.

If you are writing a new protocol to use on top of IP, or if you want to use the ICMP protocol, you must choose raw sockets; but to use raw sockets, you must be authorized by way of RACF® or APF.

Socket Libraries

Figure 3 on page 7 illustrates the TCP/IP networking API relationship on OS/390.

[1] David R. Cheriton and Carey L. Williamson, “MVSTP as the Transport Layer for High-Performance Distributed Systems,” *IEEE Communications* , June 1989, Vol. 27, No. 6.

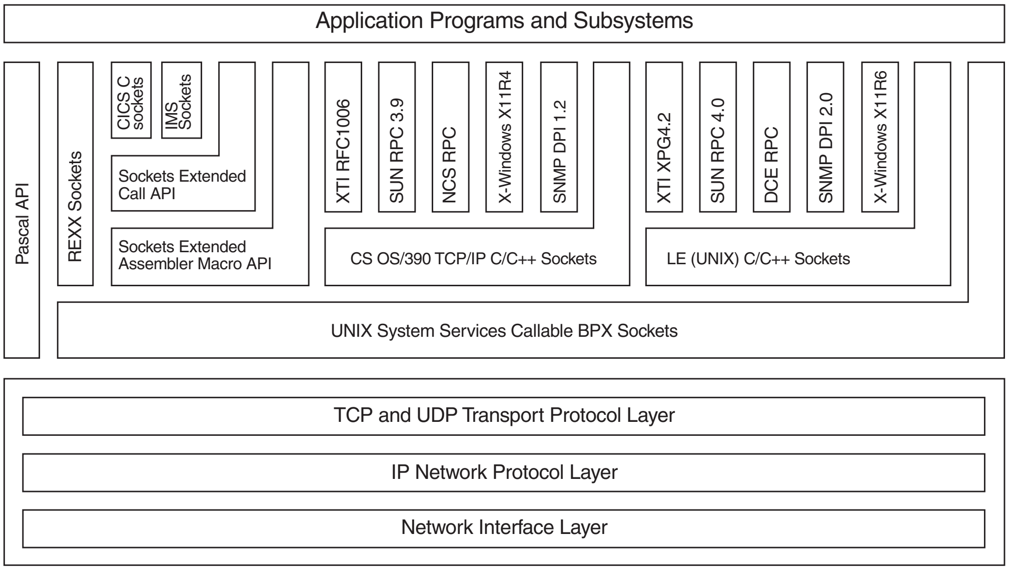

Application Programs and Subsystems
Pascal API
REXX Sockets
CICS C sockets
IMS Sockets
Sockets Extended Call API
Sockets Extended Assembler Macro API
XTI RFC1006
SUN RPC 3.9
NCS RPC
X-Windows X11R4
SNMP DPI 1.2
CS OS/390 TCP/IP C/C++ Sockets
XTI XPG4.2
SUN RPC 4.0
DCE RPC
SNMP DPI 2.0
X-Windows X11R6
LE (UNIX) C/C++ Sockets
UNIX System Services Callable BPX Sockets
TCP and UDP Transport Protocol Layer
IP Network Protocol Layer
Network Interface Layer

Figure 3. TCP/IP Networking API Relationship on OS/390

When we create a sockets program, we use something which generally is called a sockets library. A sockets library consists of both compile-time structures, statically linked support modules, and run-time support modules.

There are two main sockets execution environments in OS/390 with available libraries:

- Native TCP/IP - implemented by TCP/IP in CS for OS/390
- UNIX - implemented by OS/390 UNIX System Services (Language Environment)

Native TCP/IP

A non-UNIX socket program can only use one TCP/IP protocol stack at a time. The native TCP/IP C socket library is not POSIX compliant and it should not be used for new C socket program development. The non-C native TCP/IP socket libraries (sockets extended - call and assembler macro, REXX sockets, CICS® sockets, and IMS™ sockets) are available for development of new socket application programs. The following TCP/IP Services APIs are included in this library:

Pascal API

The Pascal application programming interface enables you to develop TCP/IP applications in Pascal language. Supported environments are normal MVS address spaces. The Pascal programming interface is based on Pascal procedures and functions that implement conceptually the same functions as the C socket interface. The Pascal routines, however, have different names than the C socket calls. Unlike the other APIs, the Pascal API does not interface directly with the LFS. It uses an internal interface to communicate with the TCP/IP protocol stack.

IMS sockets

The Information Management System (IMS) socket interface supports development of client/server applications in which one part of the application executes on a TCP/IP-connected host and the other part executes as an IMS application program. The programming interface used by both application parts is the socket programming interface, and the communication protocols are either TCP, UDP, or IP. For more information, refer to the *OS/390 eNetwork Communications Server: IP IMS Sockets Guide*.

CICS sockets

The CICS socket interface enables you to write CICS applications that act as clients or servers in a TCP/IP-based network. Applications can be written in C language, using the C sockets programming, or they can be written in COBOL, PL/I or assembler, using the Extended Sockets programming interface. For more information, refer to the *OS/390 Secureway Communications Server: IP CICS Sockets Guide*.

CS OS/390 TCP/IP C/C++ Sockets

The C/C++ Sockets interface supports socket function calls that can be invoked from C/C++ programs.

Note: Use of the UNIX C socket library is encouraged.

Sockets Extended macro API

The Sockets Extended macro API is a generalized assembler macro-based interface to sockets programming. It includes extensions to the socket programming interface, such as support for asynchronous processing on most sockets function calls.

Sockets Extended Call Instruction API

The Sockets Extended Call Instruction API is a generalized call-based interface to sockets programming. The functions implemented in this call interface resemble the C-sockets implementation, with some extensions similar to the sockets extended macro interface.

REXX sockets

The REXX sockets programming interface implements facilities for socket communication directly from REXX programs by way of an address rxsocket function. REXX socket programs can execute in TSO, online, or batch.

UNIX

A UNIX socket program can use up to eight TCP/IP protocol stacks at once. The stacks may be a combination of any TCP/IP protocol stack that is supported by OS/390 UNIX System Services. The following APIs are provided by the UNIX element of OS/390 and are not addressed in detail in this publication:

OS/390 UNIX C sockets

OS/390 UNIX C sockets is used in the OS/390 UNIX environment. Programmers use this API to create applications that conform to the POSIX or XPG4 standard (a UNIX specification). Applications built with OS/390 UNIX C sockets can be ported to and from platforms that support these standards. For more information, refer to the *OS/390 C/C++ Run-Time Library Reference*.

OS/390 UNIX Assembler Callable Services

OS/390 UNIX Assembler Callable Services is a generalized call-based interface to OS/390 UNIX sockets programming. The functions implemented in this call interface resemble the OS/390 UNIX C sockets implementation, with some extensions similar to the sockets extended macro interface. For more information, refer to the *OS/390 UNIX System Services Programming: Assembler Callable Services Reference*.

Address Families

Address families define different styles of addressing. All hosts in a given address family understand and use the same scheme for addressing socket endpoints. TCP/IP supports one addressing family: AF_INET. The AF_INET domain defines addressing for the internet domain

Addressing Sockets in an Internet Domain

This section describes how to address sockets in an internet domain.

Internet Addresses

Internet addresses are 32-bit quantities that represent a network interface. Every internet address within an administered AF_INET domain must be unique. It is not the case that every host must have a single unique internet address. In fact, a host has as many internet addresses as it has network interfaces.

Ports

A port is used to differentiate among different applications using the same network interface. It is an additional qualifier used by the system software to get data to the correct application. Physically, a port is a 16-bit integer. Some ports are reserved for particular applications; they are labeled as well known ports.

In the client/server model, the server provides a resource by listening for clients on a particular port. Some applications, such as FTP, SMTP, and Telnet, are standardized protocols and listen on a well known port. Such standardized applications use the same port number on all TCP/IP hosts. For your client/server applications however, you need a way to assign port numbers to represent the services you intend to provide. An easy way to define services and their ports is to enter them into data set *hlq*.ETC.SERVICES. In C, the programmer uses the getservbyname() function to determine the port for a particular service. Should the port number for a particular service change, only the *hlq*.ETC.SERVICES data set needs to be modified.

Note: Note that *hlq* is the high-level qualifier. CS for OS/390 ships with a default hlq of TCPIP. Use this default or override it using the DATASETPREFIX statement in the PROFILE.TCPIP and TCPIP.DATA configuration files. TCP/IP is shipped with data set *hlq*.ETC.SERVICES that contains the well known services of FTP, SMTP, and Telnet. Data set *hlq*.ETC.SERVICES is described in *OS/390 IBM Communications Server: IP Configuration Reference*.

A socket program in an IP host identifies itself to the underlying TCP/IP protocol layers by port number.

A port number is a 16-bit integer ranging from zero to 65535. A port number uniquely identifies this application to the protocol underlying this TCP/IP host (TCP, UDP, or IP). Other applications in the TCP/IP network can contact this application by way of reference to the port number on this specific IP host.

Figure 4 on page 10 shows the port concept.

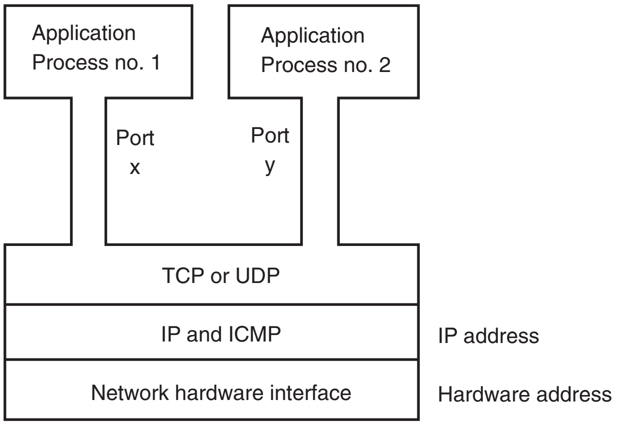


Figure 4. The Port Concept.

Both server applications and client applications have port numbers. A server application uses a specific port number to uniquely identify this server application. The port number can be reserved to a particular server, so no other process ever uses it. In an IBM TCP/IP for MVS environment, you can do this using the PORT statement in the *hlq*.PROFILE.TCP/IP configuration data set. When the server application initializes, it uses the bind() socket call to identify its port number. A client application must know the port number of a server application in order to contact it.

Because advance knowledge of the client's port number is not needed, a client often leaves it to TCP/IP to assign a free port number when the client issues the connect() socket call to connect to a server. Such a port number is called an ephemeral port number; this means it is a port number with a short life. The selected port number is assigned to the client for the duration of the connection, and is then made available to other processes. It is the responsibility of the TCP/IP software to ensure that a port number is assigned to only one process at a time.

Well known official Internet port numbers are in the range of zero to 255. You can find a list of these port numbers in Assigned Numbers, RFC1700. In addition, port numbers in the range of 256 to 1023 are reserved for other well known services. Port numbers in the range of 1024 to 5000 are used by TCP/IP when TCP/IP automatically assigns port numbers to client programs that do not use a specific port number. Your server applications should use port numbers above 5000.

Figure 5 shows port number assignments.

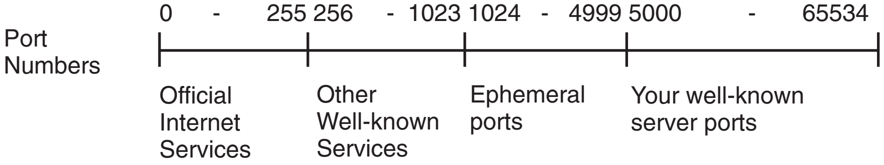


Figure 5. Port Number Assignments

Before you select a port number for your server application, consult the hlq.ETC.SERVICES data set. This data set is used to assign port numbers to server applications. The server application can use socket call getservbyname() to retrieve the port number assigned to a given server name. Add the names of your server applications to this data set and use socket call getservbyname(). With this technique, you avoid hard coding the port number into your server program. The client program must know the port number of the server on the server host. There is no socket call to obtain that information from the server host. To compensate, synchronize the contents of data sets ETC.SERVICES on all TCP/IP hosts in your network. Client application can then use the getservbyname() socket call to query its local ETC.SERVICES data set for the port number of the server. Use this technique to develop your own local well known services.

Network Byte Order

Ports and addresses are usually specified by calls using the network byte ordering convention. Network byte order is also known as big endian byte ordering, where the high order byte defines significance. Network byte ordering allows hosts using different architectures to exchange address information. See “accept()” on page 85, “bind()” on page 87, “htonl()” on page 130, “htons()” on page 131, “ntohl()” on page 143 and “ntohs()” on page 144 for more information about network byte order.

Notes:

1. The socket interface does not handle application data byte ordering differences. Application writers must handle byte order differences themselves, or use higher level interfaces such as remote procedure calls (RPC). For description of the RPC calls, see *TCP/IP for MVS: Programmer’s Reference*.
2. If you use the socket API, your application must handle the issues related to different data representations on different hardware platforms. For character based data, some hosts use ASCII, while other hosts use EBCDIC. Your application must handle translation between the two representations.

Maximum Number of Sockets

For most socket interfaces, the maximum number of sockets allowed is 2000. The exception to this rule is the C sockets interface for CICS, where the maximum allowed is 255.

Socket Addresses in an Internet Domain

A socket address in an internet addressing family comprises four fields:

- The address family (AF_INET)
- The internet address
- A port
- A character array

The structure of an internet socket address is defined by the following *sockaddr_in* structure, which is found in header file IN.H:

```
struct in_addr
{
        u_long s_addr;
};
struct sockaddr_in
{
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

The *sin_family* field is set to AF_INET. The *sin_port* field is the port used by the application, in network byte order. The *sin_addr* field is the internet address of the network interface used by the application; it is also in network byte order. The *sin_zero* field should be set to zeros.

Chapter 2. Organizing a TCP/IP Application Program

This chapter explains how to organize a TCP/IP application program and describes the following concepts:

- Client and server socket programs
- Call sequence in socket programs
- Blocking, nonblocking, and asynchronous socket calls
- Testing a program using a miscellaneous server
- Testing a local machine with a loopback address
- Accessing required data sets

Client and Server Socket Programs

The terms client and server are common within the TCP/IP community, and many definitions exist. In the TCP/IP context, these terms are defined as follows:

Server
A process that waits passively for requests from clients, processes the work specified, and returns the result to the client that originated the request.

Client A process that initiates a service request.

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

No matter which socket programming interface you select, function is identical. The syntax might vary, but the underlying concept is the same.

While clients communicate with one server at a time, servers can serve multiple clients. When you design a server program, plan for multiple concurrent processes. Special socket calls are available for that purpose; they are called concurrent servers, as opposed to the more simple type of iterative server.

To distinguish between these generic socket program categories, the following terms are used:

- **Client program** identifies a socket program that acts as a client.
- **Iterative server program** identifies a socket program that acts as a server, and processes fully one client request before accepting another client request.
- **Concurrent server main program** identifies that part of a concurrent server that manages child processes, accepts client connections, and schedules client connections to child processes.
- **Concurrent server child program** identifies that part of a concurrent server that processes the client requests.

In a concurrent server main program, the child program might be active in many parallel child processes, each processing a client request. In an MVS environment, a process is either an MVS task, a CICS transaction, or an IMS transaction.

Iterative Server Socket Programs

An iterative server processes requests from clients in a serial manner; one connection is served and responded to before the server accepts a new client connection.

Figure 6 shows the iterative server main logic.

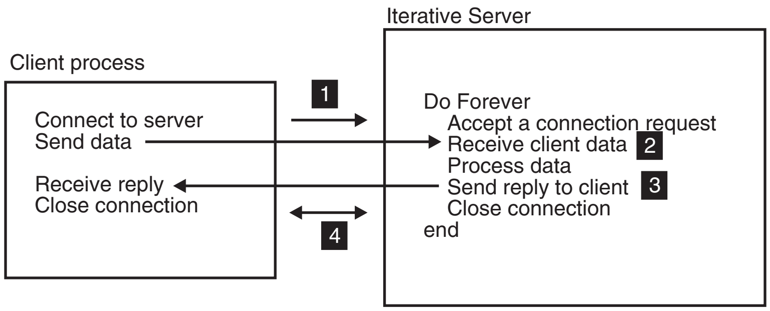


Figure 6. Iterative Server Main Logic

The following list describes the iterative server socket process.

1. When a connection request arrives, it accepts the connection and receives the client data.
2. The iterative server processes the received data and does whatever has to be done to build a reply.
3. The server sends the data back to the client.
4. The iterative server closes the socket and waits for the next connection request from the network.

An MVS iterative server can be implemented as follows:

- As a batch job or MVS task started manually, or by automation software. The job remains active until it is closed by operator intervention.
- As a TSO transaction. For a production implementation, submit a job that executes a batch terminal monitor program (TMP).
- As a long-running CICS task. The task normally begins during CICS start-up, but it can be started by an authorized CICS operator entering the appropriate CICS transaction code.
- As a batch message program (BMP) in IMS.

From a socket programming perspective, there is no difference between an iterative server that runs in a native MVS environment (batch job, started task, or TSO) and one that runs as a CICS task, or as a BMP under IMS.

You can terminate the server process in various ways. For jobs that execute in traditional MVS address spaces (batch job, started task, TSO, IMS BMP), you can implement functions in the server to enable an operator to use the MVS MODIFY command to signal stop; for example F SERVER,STOP. (This technique cannot be used for CICS tasks.) Alternatively, you can include a shutdown message in the application protocol. By doing so, you can develop a shutdown client program that connects to the server and sends a shutdown message. When the server receives a shutdown message from a socket client, it terminates itself.

Concurrent Server Socket Programs

A concurrent server accepts a client connection, delegates the connection to a child process of some kind, and immediately signals its availability to receive the next client connection.

The following list describes the concurrent server process.

1. When a connection request arrives in the main process of a concurrent server, it schedules a child process and forwards the connection to the child process.
2. The child process takes the connection from the main process.
3. The child process receives the client request, processes it, and returns a reply to the client.
4. The connection is closed, and the child process terminates or signals to the main process that it is available for a new connection.

You can implement a concurrent server in the following MVS environments:

- Native MVS (batch job, started task, or TSO). In this environment you implement concurrency by using traditional MVS subtasking facilities. These facilities are available from assembler language programs or from high-level languages that support multitasking or multithreading; for example, C/370.
- CICS. The concurrent main process is started as a long-running CICS task that accepts connection requests from clients, and initiates child processes by way of the EXEC CICS START command. CICS sockets includes a generic concurrent server main program called the CICS LISTENER.
- IMS. The concurrent main process is started as a BMP that accepts connection requests from clients, and initiates child processes by way of the IMS message switch facilities. The child processes execute as IMS message processing programs (MPP). IMS sockets include a generic concurrent server main program called the IMS LISTENER.

In the iterative and concurrent server scenarios described above, client and server processes could have exchanged a series of request and reply sequences before closing the connection.

Call Sequence in Socket Programs

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

Call Sequence in Stream Socket Sessions

This section describes a typical stream socket session.

Use stream sockets for both passive (server) and active (client) processes. While some calls are necessary for both types, others are role specific. See “Sample C Socket Programs” on page 189, for sample socket communication client and server programs. All connections exist until closed by the socket. During the connection, data is delivered, or an error code is returned by TCP/IP.

Figure 7 on page 16 shows the general sequence of calls for most socket routines using stream sockets.

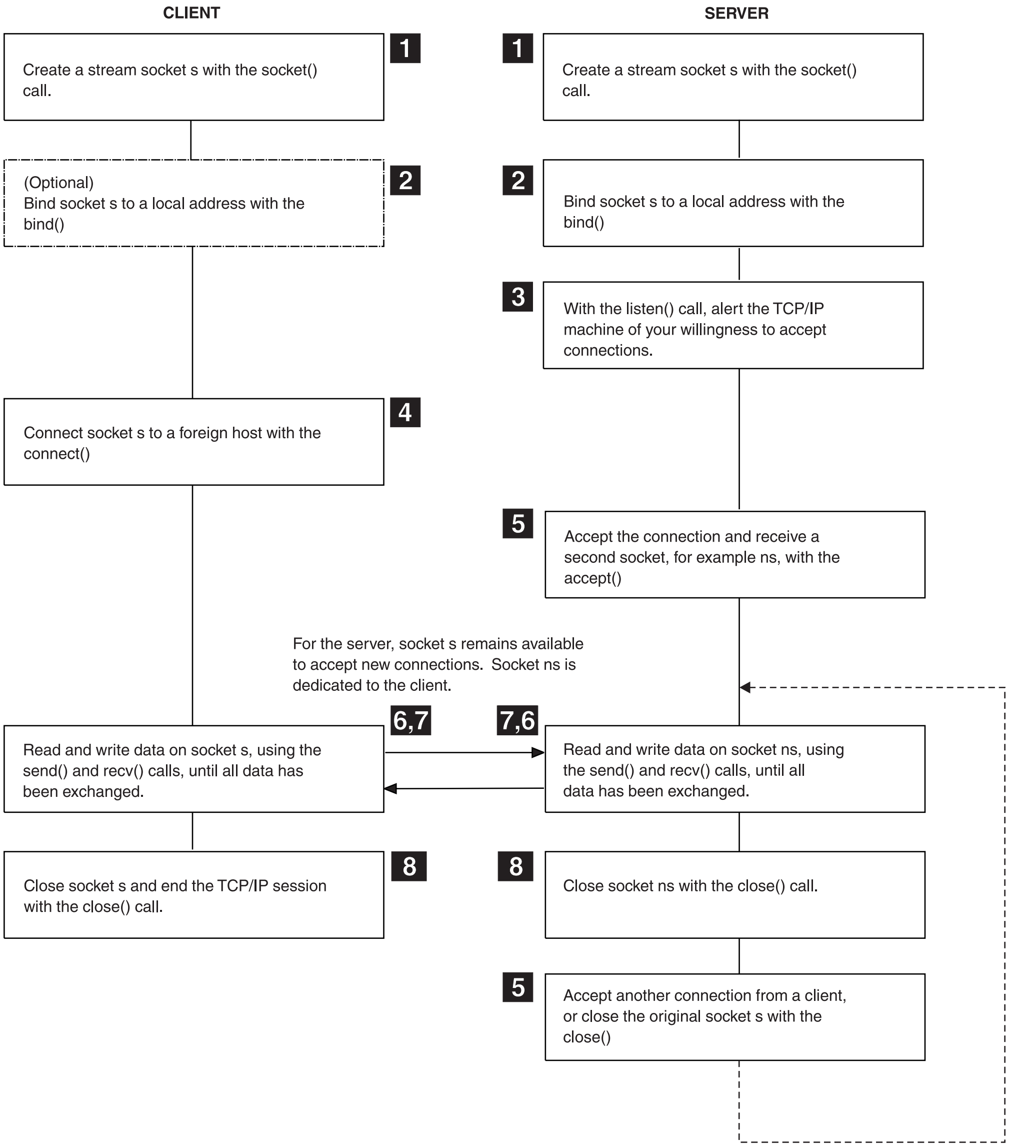

CLIENT
SERVER
1
Create a stream socket s with the socket() call.
1
Create a stream socket s with the socket() call.
2
(Optional)
Bind socket s to a local address with the bind()
2
Bind socket s to a local address with the bind()
3
With the listen() call, alert the TCP/IP machine of your willingness to accept connections.
4
Connect socket s to a foreign host with the connect()
5
Accept the connection and receive a second socket, for example ns, with the accept()
For the server, socket s remains available to accept new connections. Socket ns is dedicated to the client.
6,7
7,6
Read and write data on socket s, using the send() and recv() calls, until all data has been exchanged.
Read and write data on socket ns, using the send() and recv() calls, until all data has been exchanged.
8
Close socket s and end the TCP/IP session with the close() call.
8
Close socket ns with the close() call.
5
Accept another connection from a client, or close the original socket s with the close()

Figure 7. A Typical Stream Socket Session

Call Sequence in Datagram Socket Sessions

Datagram socket processes, unlike stream socket processes, are not clearly distinguished by server and client roles. The distinction lies in connected and unconnected sockets. An unconnected socket can be used to communicate with any host, but a connected socket can send data to and receive data from one host only.

Both connected and unconnected sockets transmit data without verification. After a packet has been accepted by the datagram interface, neither its integrity nor its delivery can be assured.

Figure 8 shows the general sequence of calls for socket routines using datagram sockets.

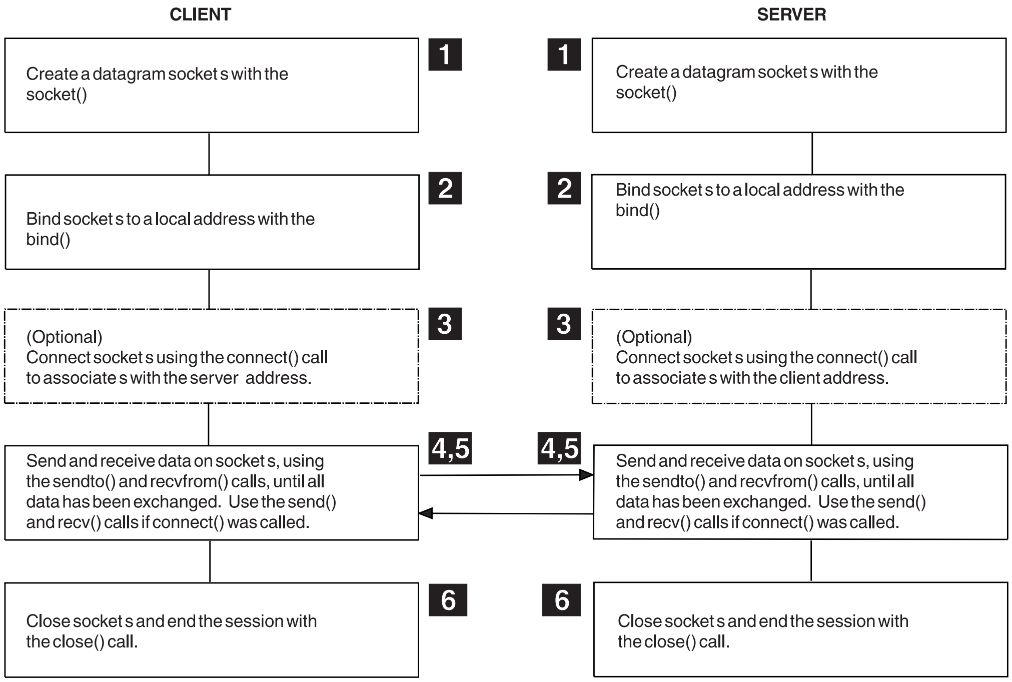

CLIENT
SERVER
1
Create a datagram socket s with the socket()
1
Create a datagram socket s with the socket()
2
Bind socket s to a local address with the bind()
2
Bind socket s to a local address with the bind()
3
(Optional)
Connect socket s using the connect() call to associate s with the server address.
3
(Optional)
Connect socket s using the connect() call to associate s with the client address.
4,5
Send and receive data on socket s, using the sendto() and recvfrom() calls, until all data has been exchanged. Use the send() and recv() calls if connect() was called.
4,5
Send and receive data on socket s, using the sendto() and recvfrom() calls, until all data has been exchanged. Use the send() and recv() calls if connect() was called.
6
Close socket s and end the session with the close() call.
6
Close socket s and end the session with the close() call.

Figure 8. A Typical Datagram Socket Session

Blocking, Nonblocking, and Asynchronous Socket Calls

A socket is in blocking mode when an I/O call waits for an event to complete. If the blocking mode is set for a socket, the calling program is suspended until the expected event completes.

If nonblocking is set by the FCNTL() or IOCTL() calls, the calling program continues even though the I/O call might not have completed. If the I/O call could not be completed, it returns with ERRNO EWOULDBLOCK. (The calling program should use SELECT() to test for completion of any socket call returning an EWOULDBLOCK.)

Note: The default mode is blocking.

If data is not available to the socket, and the socket is in blocking and synchronous modes, the READ call blocks the caller until data arrives.

All IBM TCP/IP for MVS socket APIs support nonblocking socket calls. Some APIs, in addition to nonblocking calls, support asynchronous socket calls.

Blocking

The default mode of socket calls is blocking. A blocking call does not return to your program until the event you requested has been completed. For example, if you issue a blocking recvfrom() call, the call does not return to your program until data is available from the other socket application. A blocking accept() call does not return to your program until a client connects to your socket program.

Nonblocking

Change a socket to nonblocking mode using the ioctl() call that specifies command FIONBIO and a full word (four byte) argument with a non-zero binary value. Any succeeding socket calls against the involved socket descriptor are nonblocking calls.

Alternatively, use the fcntl() call using the F_SETFL command and FNDELAY as an argument.

Nonblocking calls return to your program immediately to reveal whether the requested service was completed. An error number may mean that your call would have blocked had it been a blocking call.

If the call was, for example, a recv() call, your program might have implemented its own wait logic and reissued the nonblocking recv() call at a later time. By using this technique, your program might have implemented its own timeout rules and closed the socket, failing receipt of data from the partner program, within an application-determined period of time.

A new ioctl() call can be used to change the socket from nonblocking to blocking mode using command FIONBIO and a fullword argument of value zero (F'0').

Asynchronous

Like nonblocking calls, asynchronous calls return control to your program immediately. But in this case, there is no need to reissue the call. Asynchronous calls are available with the macro API. For more information, see "Task Management and Asynchronous Function Processing" on page 220 .

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

Table 1. Socket Programming Interface Actions

Call Type	Socket State	Blocking	Nonblocking
types of read() calls	Input is available	Immediate return	Immediate return
	No input is available	Wait for input	Immediate return with EWOULDBLOCK error number (select() exception: READ)
types of write() calls	Output buffers available	Immediate return	Immediate return
	No output buffers available	Wait for output buffers	Immediate return with EWOULDBLOCK error number (select() exception: WRITE)
accept() call	New connection	Immediate return	Immediate return
	No connections queued	Wait for new connection	Immediate return with EWOULDBLOCK error number (select() exception: READ)
connect() call		Wait	Immediate return with EINPROGRESS error number (select() exception: WRITE)

Test pending activity on a number of sockets in a synchronous program by using the select() call. Pass the list of socket descriptors that you want to test for activity to the select() call; specify by socket descriptor the following type of activity you want to test find:

- Pending data to read
- Ready for new write
- Any exception condition

When you use select() call logic, you do not issue any socket call on a given socket until the select() call tells you that something has happened on that socket; for example, data has arrived and is ready to be read by a read() call. By using the select() call, you do not issue a blocking call until you know that the call cannot block.

The select() call can itself be blocking, nonblocking, or, for the macro API, asynchronous. If the call is blocking and none of the socket descriptors included in the list passed to the select() call has had any activity, the call does not return to your program until one of them has activity, or until the timer value passed on the select() call expires.

The select() call and selectex() call are available. The difference between select() and selectex() calls is that selectex() call allows you to include nonsocket related events in the list of events that can trigger the selectex() call to complete. You do so by passing one or more MVS event control blocks (ECBs) on the selectex() call. If there is activity on any of the sockets included in the select list, if the specified timer expires, or if one of the external events completes, the selectex() call returns to your program.

Typically, a server program waits for socket activity or an operator command to shut it down. By using the selectex() call, a shutdown ECB can be included in the list of events to be monitored for activity.

Testing a Program Using a Miscellaneous Server

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

Tool **Server Description**

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

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

Character Generator
Specify Port 19 when you want MISCSERV to return random data regardless of the data it receives. For a stream session, data is returned continuously until you end the session; the received data stream is discarded. For a datagram session, random data is returned for each datagram received; the received datagram is discarded.

Note: The server uses MAXSOC=50. This value limits the sockets available to the server.

For more information, see RFC862, RFC863, RFC864, and *OS/390 IBM Communications Server: IP Configuration Reference* .

Testing a Local Machine Using a Loopback Address

You can use a local loopback address to test your local TCP/IP host without accessing the network. Class A network address 127.0.0.0 is considered a loopback address. You can specify LOOPBACK as the host name; this resolves to 127.0.0.1. Additional loopback addresses can also be configured by your TCP/IP administrator.

You can use the loopback address with any TCP/IP command that accepts IP addresses, although you might find it particularly useful in conjunction with FTP, PING, and TELNET commands. When you issue a command with a loopback address, the command is sent from your local host client to the local TCP/IP host where it is recognized as a loopback address and is sent to your local host server.

Using a loopback address on commands allows you to test client and server functions on the same host for proper operation.

Note: Any command or data that you send using the loopback address never actually leave your local TCP/IP host.

The information you receive reflects the state of your system and tests the client and server code for proper operation. If the client or server code is not operating properly, a command message is returned.

Accessing Required Data Sets

Table 2 lists the data sets and applications to which TCP/IP applications must have access to compile and link-edit.

Table 2. TCP/IP Data Sets and Applications

Data Set	Application
hlq.SEZABPDM	Kerberos B-plus tree database operations
hlq.SEZACMAC	C header files and Pascal include files
hlq.SEZACMTX	Sockets and Pascal API
hlq.SEZADES	Kerberos encryption function, US Kerberos only
hlq.SEZADPIL	SNMP DPI
hlq.SEZAKDB	Kerberos database administration
hlq.SEZAKRB	Kerberos ticket authentications
hlq.SEZALIBN	NCS
hlq.SEZAOLDX	X Release 10 compatibility routines
hlq.SEZARNT1	Sockets, X11, and PEXlib (reentrant)
hlq.SEZARNT2	Athena widget (reentrant)
hlq.SEZARNT3	Motif widget (reentrant)
hlq.SEZARPCL	Remote procedure calls
hlq.SEZAXAWL	Athena widget set
hlq.SEZAXMLB	OSF/Motif-based widget set
hlq.SEZAXTLB	Xt Intrinsics
hlq.SEZAX11L	Xlib, Xmu, Xext, and Xau routines

Part 2. Designing Programs

Chapter 3. Designing an Iterative Server Program

This chapter addresses the following:

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

Allocating Sockets

The server must allocate a socket to provide an endpoint to which clients connect.

A socket is actually an index into a table of connections to the TCP/IP address space, so socket numbers are usually assigned in ascending order. In C, the programmer issues the socket() call to allocate a new socket, as shown in the following example:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

The socket function requires specification of the address family (AF_INET), the type of socket (SOCK_STREAM), and the particular networking protocol to be used. When zero is specified, the TCP/IP address space automatically uses the protocol appropriate to the socket type specified. A new socket is allocated and returned.

An application must first get a socket descriptor using the socket() call, as seen in the following example. For a complete description, see “socket()” on page 181.

```
int socket(int domain, int type, int protocol);
⋮
int s;
⋮
s = socket(AF_INET, SOCK_STREAM, 0);
```

The code fragment allocates socket descriptor *s* in the internet addressing family. The domain parameter is a constant that specifies the domain in which the communication is taking place. A domain is a collection of applications using a single addressing convention. MVS supports the AF_INET addressing family. The type parameter is a constant that specifies the type of socket; it can be SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW. The protocol parameter is a constant that specifies the protocol to be used. This parameter is ignored, unless type is set to SOCK_RAW. Passing zero chooses the default protocol. If successful, the socket() call returns a positive integer socket descriptor.

The server obtains a socket by way of the socket call. You must specify the domain to which the socket belongs, and the type of socket you want.

Figure 9 on page 26 lists the socket call() variables using the CALL API.

```
*-------------------------------------------------------------*
* Variables used for the SOCKET call                          *
*-------------------------------------------------------------*
 01  afinet                          pic 9(8) binary value 2
 01  soctype-stream                  pic 9(8) binary value 1
 01  proto                           pic 9(8) binary value 0
 01  socket-descriptor               pic 9(4) binary value 0
*-------------------------------------------------------------*
* Get us a socket descriptor                                  *
*-------------------------------------------------------------*
   call 'EZASOKET' using soket-socket
     afinet
     soctype-stream
     proto
     errno
     retcode
   if retcode < 0 then
     - process error -
   else
     Move retcode to socket-descriptor
```

Figure 9. Socket Call Variables

The internet domain has a value of two. A stream socket is requested by passing a ″type″ value of one. The proto field is normally zero, which means that the socket API should choose the protocol to be used for the domain and socket type requested. In this example, the socket uses TCP protocols.

A socket descriptor representing an unnamed socket is returned from the socket() call. An unnamed socket has no port and no IP address information associated with it; only protocol information is available. The socket descriptor is a two-byte binary field and must be passed on subsequent socket calls as such.

A socket is an inconvenient concept for a program because it consists of three different items: a protocol specification, a port number, and an IP address. To represent the socket conveniently, we use the socket descriptor.

The socket descriptor is not in itself a socket, but represents a socket and is used by the socket library routines as an index into the table of sockets owned by a given MVS TCP/IP client. On all socket calls that reference a specific socket, you must pass the socket descriptor that represents the socket with which you want to work.

Figure 10 lists the MVS TCP/IP socket descriptors.

Socket Descriptor	Socket
0	Our listen socket
1	Our connected socket

Figure 10. MVS TCP/IP Socket Descriptor Table

The first socket descriptor assigned to your program is zero (for a sockets extended program). If your program is written in C, socket descriptors zero, one, and two are reserved for std.in, std.out and std.err, and the first socket descriptor assigned for your AF_INET sockets is numeral three or higher.

When a socket is closed, the socket descriptor becomes available; it is returned as a new socket descriptor representing a new socket in response to a succeeding request for a socket.

Note: In reference documentation, the socket descriptor is normally represented by a single letter: `S` or by two letters: `SD`.

When you possess the socket descriptor, you can request the socket address structure from the socket programming interface by way of call getsockname(). A socket does not include both port and IP addresses until after a successful bind(), connect(), or accept() call has been issued.

If your socket program is capable of handling sockets simultaneously, you must keep track of your socket descriptors. Build a socket descriptor table inside of your program to store information related to the socket and the status of the socket. This information is sometimes needed, and can help in debug situations.

Binding Sockets

At this point in the process, an entry in the table of communications has been reserved for your application. However, the socket has no port or IP address associated with it until you use the bind() function. The bind() function requires three parameters:

- The socket just given to the server.
- The number of the port to which the server is to provide service.
- The IP address of the network connection from which the server is to accept connection. If this address is zero, the server accepts connection requests from any address.

Binding with a Known Port Number

In C, the server puts the port number and IP address into structure sockaddr_, *x*, passing it, and the socket, to the bind() function. For example:

```
bind(s, (struct sockaddr *)&x, sizeof(struct sockaddr));
```

After an application possesses a socket descriptor, it can explicitly bind() a unique address to that socket, as in the example listed in Figure 11. For more information about binding, see “bind()” on page 87.

```
int bind(int s, struct sockaddr *name, int namelen);
⋮
int rc;
int s;
struct sockaddr_in myname;

   /* clear the structure to clear the sin_zero field */
   memset(&myname,; 0, sizeof(myname));
   myname.sin_family = AF_INET;
   myname.sin_addr = inet_addr("129.5.24.1"); /* specific interface */
   myname.sin_port = htons(1024);
⋮
   rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
```

Figure 11. An Application Using the bind() Call

This example binds socket descriptor *s* to the address `129.5.24.1`, and port `1024` to the internet domain. Servers must bind to an address and port to be accessible to the network. The example in Figure 11 on page 27 lists two utility routines:

- Socket call inet_addr() takes an internet address in dotted decimal form and returns it in network byte order. For a description, see “inet_addr()” on page 132.
- Socket call htons() takes a port number in host byte order and returns the port number in network byte order. For a description, see “htons()” on page 131.

Binding Using Socket Call gethostbyname

Figure 12 shows another example of socket call bind(). It uses the utility routine gethostbyname() to find the internet address of the host, rather than using socket call inet_addr with a specific address.

```
int bind(int s, struct sockaddr *name, int namelen);
⋮
int rc;
int s;
char *hostname = "myhost";
struct sockaddr_in myname;
struct hostent *hp;

   hp = gethostbyname(hostname);

   /* clear the structure to clear the sin_zero field */
   memset(&myname,0,sizeof(myname));
   myname.sin_family = AF_INET;
   myname.sin_addr.s_addr = *((unsigned long *)hp->h_addr);
   myname.sin_port = htons(1024);
⋮
rc = bind(s,(struct sockaddr *) &myname, sizeof(myname));
```

Figure 12. A bind() Call Using gethostbyname()

Binding a Socket to a Specific Port Number

By binding the socket to a specific port number, you avoid having an ephemeral port number assigned to the socket.

Servers find it inconvenient to have an ephemeral port number assigned, because clients have to connect to a different port number for every instance of the server. By using a predefined port number, clients can be developed to always connect to a given port number.

Client programs can use the socket call bind(), but client programs rarely benefit from using the same port number every time they execute.

Figure 13 on page 29 shows a list of BIND call variables.

```
*----------------------------------------------------------------*
* Variables used for the BIND Call                               *
*----------------------------------------------------------------*
 01  server-socket-address.
     05  server-afinet               pic 9(4) binary value 2
     05  server-port                 pic 9(4) binary value 9998
     05  server-ipaddr               pic 9(8) binary value 0
     05  filler                      pic x(8) value low-value
 01  socket-descriptor               pic 9(4) binary
*----------------------------------------------------------------*
* Bind socket to our server port number                          *
*----------------------------------------------------------------*
   call 'EZASOKET' using soket-bind
      socket-descriptor
      server-socket-address
      errno
      retcode
   if retcode < 0 then
      - process error -
```

Figure 13. Variables Used for the BIND Call

Before you issue this call, you must build a socket address structure for your own socket using the following information:

- Addressing family = two, which means: AF_INET
- Port number for your server application. For a sockets extended program, you have to create a predefined port number; this is either a constant in your program, or a variable passed to your program as an initialization parameter. If you develop your socket program in C, you can issue a getservbyname() call to locate the port number reserved for your server application in data set *hlq*.ETC.SERVICES.
- IP address on which your server application is to accept incoming requests. If your application is executing on a multihomed host, and you want to accept incoming requests over all available network interfaces, you must set this field to binary zeroes, which means: INADDR_ANY

Normally, the IP address is set to INADDR_ANY, but there are situations in which you might want to use a specific IP address. Consider the case of a TCP/IP system address space having been configured with two virtual IP addresses (VIPA). One VIPA address is returned by the named server when clients resolve one host name, and the other VIPA address is returned by the name server when clients resolve the other host name. In fact, both host names represent the same MVS TCP/IP system address space, but the host names can be used to represent two different major socket application on that MVS host. If your Server A and your Server B can generate a very high amount of network traffic, your network administrator might want to implement what is known as session traffic splitting. This means that IP traffic for one server comes in on one network adapter while traffic for the other server comes in on another adapter. To facilitate such a setup, you must be able to bind the server listener socket to one of the two VIPA addresses.

At this point in the process, you have not told TCP/IP anything about the purpose of the socket you obtained. You are free to use it as a client to issue connect requests to servers in the IP network, or use it to become a server yourself. In terms of the socket, it is, and the moment, active; this is the default status for a newly created socket.

Listening for Client Connection Requests

After the bind is issued, the server has been specified a particular IP address and port. It now must notify the TCP/IP address space that it intends to listen for connections on this socket. The listen() function puts the socket into passive open mode and allocates a backlog queue for pending connections. In passive open mode, the socket is open to client contact. For example:

```
listen(s, backlog_number);
```

The server gives to the socket on which it will be listening the number of requests that can be queued (the backlog_number). If a connection request arrives before the server can process it, the request is queued until the server is ready.

When you call listen, you inform TCP/IP that you intend to be a server and accept incoming requests from the IP network. By doing so, socket status is changed from active status to passive.

A passive socket does not initiate a connection; it waits for clients to connect to it.

The listen() call variables are shown in Figure 14.

```
*---------------------------------------------------------------*
* Variables used by the Listen Call                             *
*---------------------------------------------------------------*
 01  backlog-queue                  pic 9(8) binary value 10
 01  socket-descriptor              pic 9(4) binary
*---------------------------------------------------------------*
* Issue passive open via Listen call                            *
*---------------------------------------------------------------*
   call 'EZASOKET' using soket-listen
   socket-descriptor
   backlog-queue
   errno
   retcode
 if retcode < 0 then
    - process error -
```

Figure 14. Variables Used by the Listen Call

The backlog queue value is used by the TCP/IP system address space when a connect request arrives and your server program is busy processing the previous client request. TCP/IP queues new connection requests to the number you specify in the backlog queue parameter. If additional connection requests arrive, they are rejected by TCP/IP, since there is a limit to the size of the backlog queue parameter.

The system-wide limit is set in the TCP/IP system address space PROFILE.TCP/IP configuration data set by parameter SOMAXCONN. The default value of SOMAXCONN is ten, but you can configure it higher as follows:

```
;
; **************************************************************
; *  Set the listen queue to a maximum of 100                 *
; **************************************************************
;
SOMAXCONN 100
```

The value you specify on the listen() call in the backlog parameter cannot exceed the value set for SOMAXCONN in TCPIP.PROFILE. If you specify a backlog parameter of 200 and SOMAXCONN is set to 20, no error is returned, but your backlog queue size will be set to 20 instead of the 200 you requested.

There is a C header file called SOCKET.H (datasetprefix.SEZACMAC member SOCKET) in which there is a variable called SOMAXCONN. The shipped value of this variable is ten as illustrated below:

```
/*
 *Maximum queue length specifiable by listen
/*
#define SOMAXCONN        10
```

The listen () call does not establish connections; it merely changes the socket to a passive state, so it is prepared to receive connection requests coming from the IP network. If a connection request for this server arrives between the time of the listen() call and the succeeding accept() call, it is queued according to the backlog value passed on the listen() call.

Accepting Client Connection Requests

To this point in the process, the server has allocated a socket, bound the socket to an IP address and port, and issued a passive open. The next step is for the server to connect with a client. The accept() call blocks the server until a connection request arrives; if there are connection requests in the backlog queue, a connection is established with the first client in the queue. The following is an example of the accept() call:

```
client_sock = accept(s);
```

The server passes its socket to the accept call. When the connection is established, the accept call returns a new socket representing the connection with the client. When the server wishes to communicate with the client, or to end the connection, it uses this new socket, `client_sock`. The original socket *s* is now ready to accept connection to other clients. The original socket is still allocated, bound, and passively opened. To accept another connection, the server calls accept() again. By repeatedly calling accept(), the server can establish simultaneous sessions with multiple clients.

The accept() call dequeues the first queued connection request or blocks the caller until a connection request arrives over the IP network.

The accept() call uses the variables listed in Figure 15 on page 32.

```
*-------------------------------------------------------------*
* Variables used by the ACCEPT Call                           *
*-------------------------------------------------------------*
   01  client-socket-address.
       05  client-afinet             pic 9(4) binary value 0
       05  client-port               pic 9(4) binary value 0
       05  client-ipaddr             pic 9(8) binary value 0
       05  filler                    pic x(8) value low-value
   01  accepted-socket-descriptor    pic 9(4) binary value 0
   01  socket-descriptor             pic 9(4) binary
*-------------------------------------------------------------*
* Start iterative server loop with a blocking Accept Call     *
*-------------------------------------------------------------*
  call 'EZASOKET' using soket-accept
    socket-descriptor
    client-socket-address
    errno
    retcode
  if retcode < 0 then
     - process error -
  else
     Move retcode to accepted-socket-descriptor
```

Figure 15. Variables Used by the ACCEPT Call

This call works with the following socket descriptors:

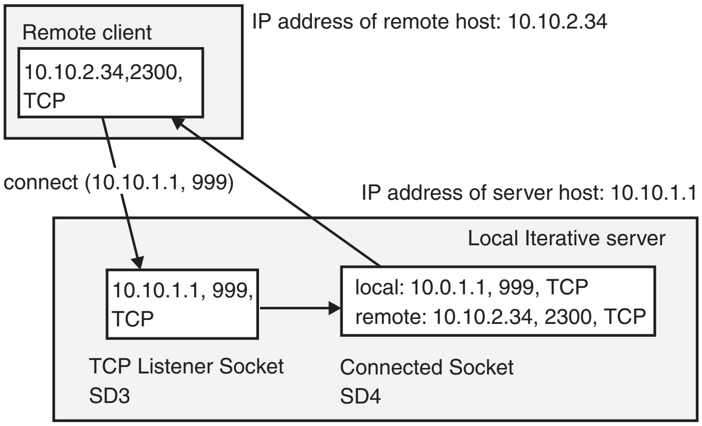
- The first socket descriptor represents the socket that was obtained, bound to the server port and (optionally) the IP address, and changed to the passive state using the listen() call.
- The accept() call returns a new socket descriptor, to represent a complete association:

  ```
  Accepted_socket_descriptor represents:
  {TCP, server IP address, server port, client IP address, client port}
  ```

The original socket, which was passed to the accept() call, is unchanged and is still representing our server half association only:

```
Original_socket_descriptor represents:
{TCP, server IP address, server port}
```

When control returns to your program, the socket address structure passed on the call has been filled with the socket address information of the connecting client. Figure 16 on page 33 illustrates the socket states.



Socket Descriptor Table for the local Iterative server

Descriptor#	Local part (IP addr, port, protocol)	Remote part (IP addr, port, protocol)
SD3	10.0.1.1, 999, TCP	
SD4	10.0.1.1, 999, TCP	10.10.2.34, 2300, TCP

Figure 16. Socket States

When a socket is created, we know the protocol we are going to use with this socket, but nothing else. When a server calls the bind() function, a local address is assigned to the socket, but the socket still only represents a half-association; the remote address is still empty. When the client connects to the listener socket and a new socket is created, this new socket represents a fully bound socket possessing both a local address (that of the listener socket) and a remote address (that of the client socket). Figure 16 illustrates a fully bound socket.

Subsequent socket calls for the exchange of data between the client and the server use the new socket descriptor. The original socket descriptor remains unused until the iterative server has finished processing the client request and closed the new socket. The iterative server then reissues the accept() call using the original socket descriptor and waits for a new connection.

Transferring Data between Sockets

See “Chapter 7. Transferring Data between Sockets” on page 61.

Closing a Connection

Closing a socket imposes some problems because the TCP protocol layer must ensure that all data has been successfully transmitted and received before the socket resources can be safely freed at both ends.

The following sections describe various ways to close a connection.

Active and Passive Closing

The program that initiates the close-down process by issuing the first close() call, is said to initiate an active close. The program that closes in response to the initiation is said to initiate a passive close.

Figure 17 illustrates socket closing.

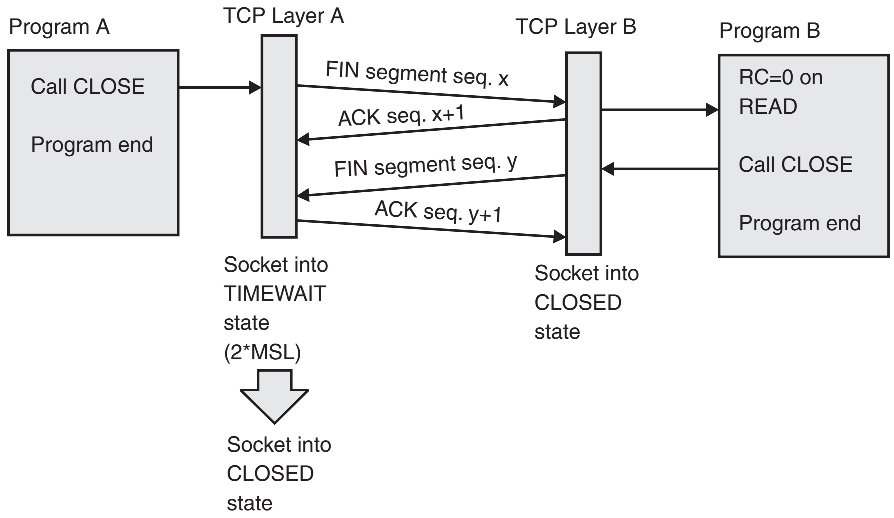


Figure 17. Closing Sockets

In “close()” on page 91, Program A initaties the active close, while Program B initiates the passive close. When a program calls the close socket function, the TCP protocol layer sends a segment known as FIN (FINish). When Program B receives the final acknowledgment segment, it knows that all data has been successfully transferred and that Program A has received and processed the FIN segment. The TCP protocol layer for Program B can then safely remove the resources that were occupied by the Program socket. The TCP protocol layer for Program A sends an acknowledgment to the FIN segment it received from Program B; but the Program A TCP protocol layer does not know whether that ACK segment arrived at the Program B TCP protocol layer. It must wait a reasonable amount of time to see whether the FIN segment from Program B is retransmitted, indicating that Program B never received the final ACK segment from Program A. In that case, Program A must be able to retransmit the final ACK segment. The Program A socket cannot be freed until this time period has elapsed. The time period is defined as twice the maximum segment life time; normally between one and four minutes, depending on the TCP implementation.

If Program A is the client in a TCP connection, this TIMEWAIT state does not create any major problems. A client normally uses an ephemeral port number; if the client restarts before the TIMEWAIT period has elapsed, it is merely assigned another ephemeral port number. If Program A, on the other hand, is the server in a TCP connection, this TIMEWAIT state does create a problem. A server binds its socket to a predefined port number; if the server tries to restart and bind the same port number before the TIMEWAIT period has elapsed, it receives an EADDRINUSE error code on the bind() call. This situation could arise when a server crashes and you try to restart it before the TIMEWAIT period has elapsed; you must wait to restart your server.

If the server cannot wait for one to four minutes, you can use the setsockopt() call in the server to specify SO_REUSEADDR before it issues the bind() call. In that case, the server will be able to bind its socket to the same port number it was using before, even if the TIMEWAIT period has not elapsed. However, the TCP protocol layer still prevents it from establishing a connection to the same partner socket address. As clients normally initiate connections and clients use ephemeral port numbers, the likelihood of this is low.

Shutdown Call

If you want to close the stream in one direction only, use the shutdown socket call instead of the close() call. On the shutdown() call, you can specify the direction in which the stream is to be closed.

See Table 3 for a list of the effect on read and write calls when the stream is shut down in one or both directions.

Table 3. Effect of Shutdown Socket Call

Socket calls in local program	Local Program		Remote Program	
	Shutdown SEND	Shutdown RECEIVE	Shutdown RECEIVE	Shutdown SEND
Write calls	Error number EPIPE on first call		Error number EPIPE on second call*	
Read calls		Zero length return code		Zero length return code
* If you issue two write calls immediately, both might be successful, and an EPIPE error number might not be returned until a third write call is issued.				

Linger Option

By default, a close socket call returns control to your program immediately, even where there is unsent data on the socket. This data will be transmitted by the TCP protocol layer, but your program is not notified of any error. This is true of both blocking and nonblocking sockets.

You can request that no control be returned to your program before unsent data has been transmitted and acknowledged by the receiver. To do so, issue the SO_LINGER option on call setsockopt. Before you issue the actual close() call, pass the following option value fields:

ONOFF
This full word is used to enable or disable the SO_LINGER option. Any nonzero value enables the option; a zero value disables it.

LINGER
This is the linger time, in seconds; this is the maximum delay the close call observes. If data is successfully transmitted before this time expires, control is returned to your program. If this time interval expires before data has been successfully transmitted, control is returned to your program also. You cannot distinguish between the two return events.

Note: If you set a zero linger time, the connection cannot close in an orderly manner, but stops, resulting in a RESET segment being sent to the connection partner. Also, if the aborting socket is in nonblocking mode, the close call is treated as though no linger option had been set.

Chapter 4. Designing a Concurrent Server Program

This chapter describes concurrent server programs and includes:

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

Overview

A server handling more than one client simultaneously acts like a dispatcher. The server receives client requests, and then creates and dispatches tasks to handle each client.

In the UNIX operating system, a new process is dispatched using the fork() system call after the server has established the connection; this new process automatically inherits the socket attached to the client. In MVS, an independent task is started using the supervisor call instruction ATTACH. A server can complete the call after each connection is established (similar to the UNIX operating system), or it can repeatedly request an ATTACH when it begins execution, and pass clients to tasks that already exist. In either case, the server must manually give the new socket to the subtask. Because each task has its own socket table, it is not sufficient to pass only the socket number to the subtask; for example, socket Number 4 for Task A is not the same as socket Number 4 for Task B. You must specify the task as well as the socket number.

Concurrent Servers in Native MVS Environment

The concurrent server is complicated to implement. Logic must be split into a main program and a child program. In addition, you have to manage all processes within your application.

In the UNIX-based environment, you implement such logic by means of the UNIX fork() call. Because this call is not available in a traditional MVS environment, you must improvise.

In the UNIX environment, the fork function is implemented using APPC/MVS to schedule and initiate a child process in an MVS address space other than the address space of the original process.

For the MVS address space examples presented in this chapter, the more traditional MVS subtasking facilities are used; the main process and the child process operate as tasks within the same address space.

You can implement your concurrent server in both an IMS, a CICS, or a traditional MVS address space environment, but unlike the implementation of an iterative server, the implementation of a concurrent server is a unique to its environment. In this chapter, we discuss implementation of a concurrent server in an MVS address space.

Note: For simplicity, the scope of our applications is limited to the AF_INET addressing family and stream sockets.

If you want to implement a high-performance server application that creates or accesses MVS resource of various kinds (especially MVS data sets), you will probably implement your server as a concurrent server in an MVS address space. This address space can be TSO, batch, or started task.

To implement concurrence in an MVS address space, use MVS multitasking facilities. This limits available programming interfaces to the Sockets Extended assembler macro programming interface or to C sockets.

For the sockets extended assembler macro interface, use standard MVS subtasking facilities: ATTACH and DETACH assembler macros.

For C sockets, use the subtasking facilities that are part of the IBM implementation of C in an MVS environment.

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

MVS Subtasking Considerations

Using multiple tasks in a single address space brings unique challenges which apply equally to assembler programming and to high-level languages that support subtasking.

For example, tasks might be concurrently dispatched on different processors; for example, you run your application on an *n*-way system. Two or more tasks might execute in parallel, one perhaps passing the other.

Access to Shared Storage Areas

If two tasks access the same storage area, you need full control over the use of the storage area unless the storage is read-only. If the storage area is used to pass parameters between the tasks, you must serialize access to the shared resource (the storage area).

In an MVS environment, you can do this using MVS latching services or traditional enqueue and dequeue system calls. In assembler, use the ENQ and DEQ macros.

Figure 18 on page 39 illustrates access to a shared storage area.

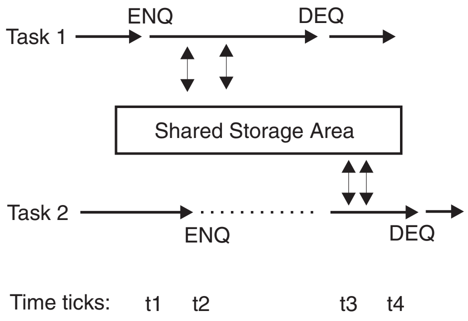


Figure 18. Serialized Access to a Shared Storage Area

The following steps describe this process.

1. At time t1, Task 1 issues a serialize request by means of an enqueue call. On the enqueue() call it passes two character fields to uniquely identify the resource in question. The literal value of these two fields does not matter; the other tasks must use these same values when they access this storage area. As no other task has issued an enqueue for the resource in question, Task 1 gets access to it and continues to modify the storage area.
2. At time t2, Task 2 needs to access the same storage area, and issues an enqueue() call using the same resource names used by Task 1. Because Task 1 has enqueued, Task 2 is placed in a wait and stays there until Task 1 releases the resource.
3. At time t3, Task 1 releases the resource with a dequeue system() call, and Task 2 is immediately taken out of its wait and begins to modify the shared storage area.
4. At time t4, Task 2 has finished updating the shared storage area and releases the resource with a dequeue system() call. (In this example, we assumed we need serialized access only when the tasks need to update information in the shared storage area.)

There are situations in which this assumption does not suffice. If you use a storage area to pass parameters to some kind of service task inside your address space, you must ensure that the service task has read the information and acted before another task in your address space tries to pass information to the service task using the same storage area; for example, a service task, like log or trace. This is illustrated in Figure 19 on page 40.

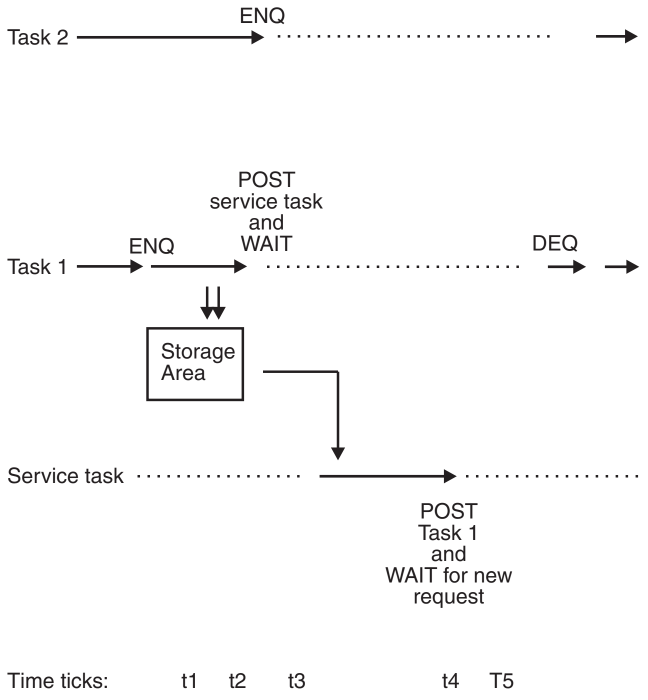


Figure 19. Synchronized Use of a Common Service Task

Follow these steps to synchronize a common service task:

1. At time t1, Task 1 gains access to the common storage area to implicitly use the service task in question.
2. At time t2, Task 2 also needs to use the service task services, but it is placed into a wait, because Task 1 already has the resource.
3. At time t3, Task 1 has finished placing values into the common storage area, and signals the service task to start processing it. This is done with a POST system call. Immediately following this call, Task 1 enters a wait, where it stays until the service task has completed its processing. The service task starts, processes the data in the common storage, and prints.
4. At time t4, the service task has finished its work and signals to Task 1 that Task 1 can continue, while the service task enters a new wait and waits for a new work request.
5. At time t5, Task 1 releases the lock it obtained at time t1, and Task 2 is immediately taken out of its wait and starts filling its values into the common storage area before posting the same service task to process a new request.

This technique is relatively simple. It can be made more complicated, and more efficient, by using internal request queues so the requesting task does not need to wait for the service task to complete the active request.

When you use the enqueue system call, you have the option to test whether a resource is available. In some situations, you might choose this to avoid the wait at a particular point in your processing, so you can divert to some other actions when the resource is not available.

Data Set Access

When you access MVS data sets in a multitasking environment, observe these general rules:

- A given DD-name can be used by only one open data control block (DCB) at a time. If you need to have more DCBs open for the same data set, you must use different DD names. This strategy works best for read access only.
- Only the task that opens a DCB can issue read and write requests using that DCB. You cannot let your main task open a DCB, and then have your subtasks issue read or write requests to that DCB. You can deal with this by using the technique described, but include a special services task that opens a DCB to a particular data set. Other tasks then issue requests to this service task for access to the data set. Such a service task is generally called a data services task (DST). One very common implementation of a DST is the example we used above: print log and trace information to a sysout file.
- Authorization checking for access to a data set is done when the data set is opened, not for every read or write request. If you develop a multitasking server, where you establish task level security environments for each transaction entering your server, you must plan to authorize access to the information in a data set owned by a DST. You can, of course, open and close the data set for each transaction, but that might degrade performance.

Task and Workload Management

When a program is started by MVS, it is executing as the main task of the address space in which it was started. In the examples in this chapter, the main task is used as the main process of our concurrent server implementation. The child processes are then started as subtasks to the main.

Generally, there are two ways to manage your processes:

- Each time a connection request arrives, a new subtask is started; the subtask makes one connection and then terminates.
- During initialization, the main task starts a number of subtasks. Each subtask initializes and enters wait-for-work status. When a connection request arrives, the main process selects the first subtask waiting for work, and schedules the connection to that subtask. The subtask processes the connection and, when complete, re-enters wait-for-work status.

The second process is most efficient because it limits the overhead of creating new tasks to one time during server startup. But, it is also more complicated to implement than the other process because of the following:

- You must decide on the number of server subtasks to be started during initialization. If more connection requests arrive than you have server subtasks available, you must include code to deal with that situation. (Reject the connection or dynamically change the number of subtasks in your concurrent server address space. This is called workload management.)
- The subtasks must be reusable and include logic to enter wait-for-work status; they must be able to process connection requests serially.
- The main process must be able to manage situations in which a server subtask abends or terminates.
- To achieve a graceful shutdown, you must implement a technique to terminate subtasks in an orderly manner. A simple technique is to post the subtask from the main process with a return code; a zero return code for work and some other value for termination.

In this chapter's concurrent MVS server example, the technique using a pool of subtasks that waited for work was presented. We did not implement a dynamic increase of subtasks, but sent a negative reply back to the requester when no server subtasks were available.

Security Considerations

When you start your server address space in MVS, a security environment is established for that address space. This environment is based on the user ID of your batch job or TSO user, or based on the started task user ID associated with the started task procedure named in the RACF started task table (ICHRIN03).

Unless you specify otherwise, all tasks in your address space execute under the security environment of the address space. MVS resources access authorization is based on the MVS address space security environment.

If this setup does not meet your needs, MVS allows you to build and delete task-level security environments using the RACROUTE REQUEST=VERIFY function in MVS; the task must run in an authorized state.

Reentrant Code

Reentrant code is not required, but is efficient. Non-reentrant code is loaded into virtual storage as many times as subtasks requiring it are started. Reentrant code is loaded once.

High level languages usually make re-entrancy a compile option. In assembler language, it might be more complicated; however, good use of macros for program initiation and termination can simplify the process.

Understanding the Structure of a Concurrent Server Program

Figure 20 on page 43 shows the basic logic in a multitasking concurrent server.

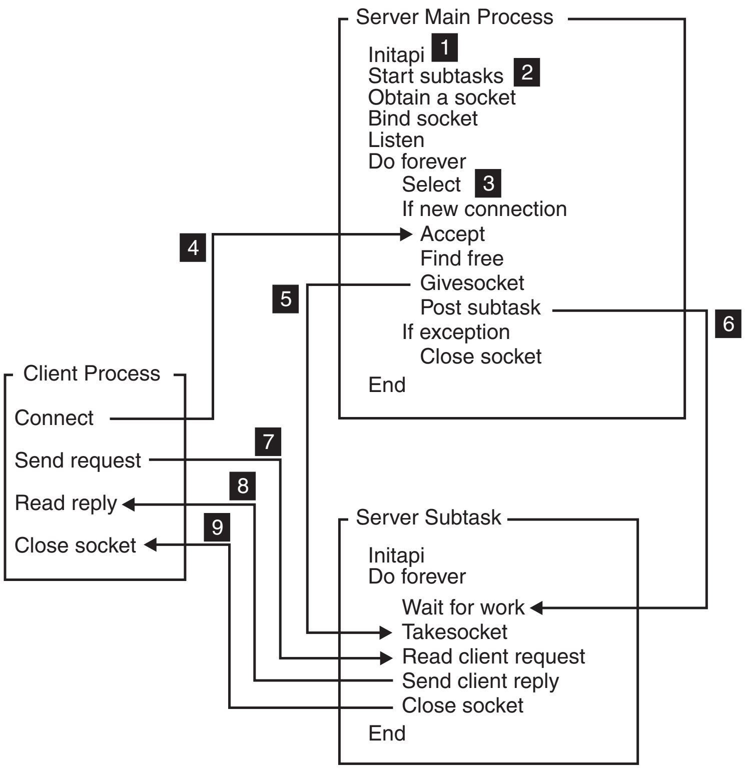


Figure 20. Concurrent Server in an MVS Address Space

Selecting Requests

At this point in the process, the server is ready to handle requests on this port from any client on a network from which the server is accepting connections. Until this point however, it had been assumed that the server was handling one socket only. Now, an application is not limited to one socket. Typically, a server listens for clients on a particular socket, but it allocates a new socket for each client it handles. For maximum performance, a server should operate only on those sockets ready for communication. The select() call allows an application to test for activity on a group of sockets.

To test any number of sockets with one call to select(), place the sockets to test into a bit set, passing the bit set to the select() call. A bit set is a string of bits where each member of the set is represented by zero or one. If the members bit is zero, the member is not in the set; if the members bit is one, the member is in the set. For example, if socket three is a member of a bit set, then bit three is set; otherwise, bit three is cleared.

In C language, the following functions are used to manipulate the bit sets:

FD_SET
Sets the bit corresponding to a socket

FD_ISSET
Tests whether the bit corresponding to a socket is set or cleared

FD_ZERO
Clears the entire bit set

If a socket is active, it is ready for read or write data. If the socket is not active, an exception condition might have occurred. Therefore, the server specifies three bit sets of sockets in its call to the select() call as follows:

- One bit set for sockets on which to receive data
- One bit set for sockets on which to write data
- Any sockets with exception conditions

The select() call tests each socket in each bit set for activity and returns only those sockets that are active.

A server that processes many clients at once can be written to process only those clients that are ready for activity.

When all initialization is complete, and the server main process is ready to enter normal work, it builds a bit mask for a select() call. The select() call is used to test pending activity on a list of socket descriptors owned by this process. Before issuing the select() call, construct three bit strings representing the sockets you want to test, as follows:

- Pending read activity
- Pending write activity
- Pending exceptional activity

The length of a bit string must be expressed as a number of fullwords. If the highest socket descriptor you want to test is socket descriptor number three, you must pass a four-byte bit string, because this is the minimum length. If the highest number is 32, you must pass eight bytes (two fullwords).

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

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

Table 4 shows the first fullword passed using a bit string.

Table 4. First Fullword Passed in a Bit String Select()

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

Using standard assembler numbering notation, the left-most bit or byte is relative to zero.

If you want to test socket descriptor number five for pending read activity, you raise bit two in byte three of the first fullword (X'00000020'). To test both socket descriptors four and five, raise both bit two and bit three in byte three of the first fullword (X'00000030').

To test socket descriptor Number 32, pass two fullwords, where the numbering scheme for the second fullword resembles that of the first. Socket descriptor

Number 32 is bit seven in byte three of the second fullword. To test socket descriptors Number 5 and Number 32, pass two fullwords with the following content: X'0000002000000001'

The bits in the second fullword represent the socket descriptor numbers shown in Table 5.

Table 5. Second Fullword Passed in a Bit String Using Select()

Socket Descriptor Numbers Represented by Byte	Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
Byte 4	63	62	61	60	59	58	57	56
Byte 5	55	54	53	52	51	50	49	48
Byte 6	47	46	45	44	43	42	41	40
Byte 7	39	38	37	36	35	34	33	32

To set and test these bits in an another way, use the following assembler macro, found in file *hlq*.SEZACMAC:

```
.*********************************************************************
.*                                                                   *
.* Part Name:          TPIMASK                                       *
.*                                                                   *
.* SMP/E Distribution Name:  EZABCTPI                                *
.*                                                                   *
.* Component Name:     SOK                                           *
.*                                                                   *
.* Copyright:    Licensed Materials - Property of IBM                *
.*               This product contains "Restricted Materials of IBM"*
.*               5645-001 5655-HAL (C) Copyright IBM Corp. 1996.     *
.*               All rights reserved.                                *
.*               US Government Users Restricted Rights -             *
.*               Use, duplication or disclosure restricted by        *
.*               GSA ADP Schedule Contract with IBM Corp.            *
.*               See IBM Copyright Instructions.                     *
.*                                                                   *
.* Status:             TCP/IP for MVS                                *
.*                                                                   *
.* Function:           Macro used to set or test bits in the         *
.*                     read, write and exception masks used          *
.*                     in the SELECT/SELECTEX macro or calls.        *
.*                                                                   *
.* Part Type:          MACRO - assembler                             *
.*                                                                   *
.* Usage:                                                            *
.*                     TPIMASK SET,MASK=READMASK,SD=SOCKDESC         *
.*                          or TEST, or WRITEMASK,                   *
.*                                   or EXCEPTMASK,                  *
.*                                                                   *
.*                     SET  - Set the SD bit on in MASK              *
.*                     TEST - Test SD bit in MASK for on/off         *
.*                            Follow the macro invocation with:      *
.*                            BE (Branch Equal) - Bit was on         *
.*                            BNE (Branch Not Equal) - Bit was off   *
.*                                                                   *
.* Change Activity:                                                  *
.* CFD List:                                                         *
.*                                                                   *
.* $xn= workitem  release  date  pgmr:  description                  *
.*                                                                   *
.* End CFD List:                                                    *
.*                                                                  *
.********************************************************************
         MACRO
         TPIMASK &TYPE,        SET or TEST bit setting                 X
               &MASK=,         Read, Write or Except array             X
               &SD=            Socket descriptor TOR PARAMETER
         SR    14,14           Clear Reg14
         AIF   ('&SD'(1,1) EQ '(').SDREG
         LH    15,&SD          Get Socket Descriptor
         AGO   .SDOK
.SDREG ANOP
         LR    15,&SD          Get Socket Descriptor
.SDOK ANOP
         D     14,=A(32)       Divide by 32, R15 = word bit is in
         SLL   15,2            Multiply word by word length: 4
         AIF   ('&MASK'(1,1) EQ '(').MASKREG
         LA    1,&MASK         Mask starts here
         AGO   .MASKOK
.MASKREG ANOP
         LR    1,&MASK         Mask starts here
.MASKOK  ANOP
         AR    15,1            Increment to word bit is in
         LA    1,1             Set rightmost bit on
         SLL   1,0(14)         Shift left remainder from division
         O     1,0(15)         Or with word from mask
         AIF   ('&TYPE' EQ 'SET').DOSET
         C     1,0(15)         If equal, bit was set on
         MEXIT
.DOSET ANOP
         ST    1,0(15)         Update new mask after SET
         MEND
```

If you develop your program using another programming language, you might be able to benefit from the EZACIC06 routine, which is provided as part of TCP/IP for MVS. This routine translates between a character string mask (one byte per flag) and a bit string mask (one bit per flag). If you use the select() call in COBOL, EZACIC06 can be very useful.

Build the three bit strings for the socket descriptors you want to test, and the select() call passes back three corresponding bit strings with bits raised for those of the tested socket descriptors with activity pending. Test the socket descriptors using the following sample.

```
*---------------------------------------------------------------------*
* Test for socket descriptor activity with the SELECT call            *
*---------------------------------------------------------------------*
         EZASMI TYPE=SELECT,       *Select call                        C
               MAXSOC=TPIMMAXD,    *Max. this many descr. to test      C
               TIMEOUT=SELTIMEO,   *One hour timeout value             C
               RSNDMSK=RSNDMASK,   *Read mask                          C
               RRETMSK=RRETMASK,   *Returned read mask                 C
               WSNDMSK=WSNDMASK,   *Write mask                         C
               WRETMSK=WRETMASK,   *Returned write mask                C
               ESNDMSK=ESNDMASK,   *Exception mask                     C
               ERETMSK=ERETMASK,   *Returned exception mask            C
               ECB=ECBSELE,        *Post this ECB when activity occurs C
               ERRNO=ERRNO,        *- ECB points to an ECB plus 100    C
               RETCODE=RETCODE,    *- bytes of workarea for socket     C
               ERROR=EZAERROR      *- interface to use.
         ICM   R2,15,RETCODE       *If Retcode < zero it is
         BM    EZAERROR            *- an error
*
SELMASKS DS    0F
RSNDMASK DC    XL8'00000000'       *Read mask
RRETMASK DC    XL8'00000000'       *Returned read mask
```

```
WSNDMASK DC    XL8'00000000'       *Write mask
WRETMASK DC    XL8'00000000'       *Returned write mask
ESNDMASK DC    XL8'00000000'       *Exception mask
ERETMASK DC    XL8'00000000'       *Returned exception mask
*
NOSELCD  DC    A(0)                *Keep track of selected sd's
SELTIMEO DC    A(3600,0)           *One hour timeout
ECBSELE  DC    A(0)                *Select ECB
         DC    100X'00'            *Required by EZASMI
*
TPIMMAXD DC    AL4(50)             *Maximum descriptor number
*
ERRNO    DC    A(0)                *Error number from EZASMI
RETCODE  DC    A(0)                *Returncode from EZASMI
```

In the above select() call, the asynchronous facilities of the socket assembler macro interface. By placing an ECB parameter on the EZASMI macro call, the select() call does not block the process; we receive control immediately, even if none of the specified socket descriptors had activity. Use this technique to enter a wait, which waits for a series of events of which the completion of a select() call is just one. In the sample application, the main process was placed into a wait from which it would return when any of the following events occurred:

- Socket descriptor activity occurred, and the select() call was posted.
- One of our subtasks terminated unexpectedly.
- The MVS operator issued a MODIFY command to stop the server.

The number of socket descriptors with pending activity is returned in the RETCODE field. You must process all selected socket descriptors before you issue a new select() call. A selected socket descriptor is selected only once.

When a connection request is pending on the socket for which the main process issued the listen() call, it is reported as a pending read.

When the main process has given a socket, and the subtask has taken the socket, the main process socket descriptor is selected with an exception condition. The main process is expected to close the socket descriptor when this happens.

Applications can handle multiple sockets. In such situations, use the select() call to determine the sockets that have data to be read, those that are ready for data to be written, and the sockets that have pending exceptional conditions. An example of how the select() call is used is shown in Figure 21 on page 48.

```
fd_set readsocks;
fd_set writesocks;
fd_set exceptsocks;
struct timeval timeout;
int number_of_sockets;
int number_found;
⋮
/* set bits in read write except bit masks.
* To set mask for a descriptor's use
* FD_SET(s, &readsocks)
* FD_SET(s, &writesocks)
* FD_SET(s, &exceptsocks)
*
* set number of sockets to be checked (plus 1)
* number_of_sockets = x;
*/
⋮
number_found = select(number_of_sockets,
          &readsocks, &writesocks, &exceptsocks, &timeout)
```

Figure 21. An Application Using the select() Call

In this example, the application uses bit sets to indicate that the sockets are being tested for certain conditions, and also indicates a time-out. If the time-out parameter is NULL, the call does not wait for any socket to become ready. If the time-out parameter is nonzero, the select() call waits for the amount of time required for at least one socket to become ready under the indicated condition. This process is useful for applications servicing multiple connections that cannot afford to block, waiting for data on one connection. For a description, see “select()” on page 154.

Client Connection Requests

As shown in Figure 20 on page 43, the listener socket is selected with a pending read, then a new connection request arrived, and the following socket() call must accept.

Figure 22 on page 49 illustrates this type of connection request.

```
*---------------------------------------------------------------------*
* ACCEPT the connection from a client                                  *
*---------------------------------------------------------------------*
    EZASMI   TYPE=ACCEPT,   *Accept new connection                 C
             S=TPIMSNO,     *On listener socket descriptor         C
             NAME=SOCSTRUC, *Returned client socket structure      C
             ERRNO=ERRNO,                                          C
             RETCODE=RETCOD                                        C
             ERROR=EZAERROR
    ICM      R15,15,RETCODE  *OK ?
    BM       EZAERROR        *- No, error indicated
    STH      R15,NEWSOC      *Returned new socket descriptor
*
SOCSTRUC DS  0F              *ACCEPT Socket address structure
SSTRFAM  DC  AL2(2)          *TCP/IP Addressing family
SSTRPORT DC  AL2(0)          *Port number
SSTRADDR DC  AL4(0)          *IP Address
SSTRRESV DC  8X'00'          *Reserved
*
TPIMSNO  DC  AL2(0)          *Listen socket descriptor
*
NEWSOC   DC  AL2(0)          *Returned socket descriptor
*
ERRNO    DC  A(0)            *Error number from EZASMI
RETCODE  DC  A(0)            *Returncode from EZASMI
```

Figure 22. Accepting a Client Connection

The accept call returns a new socket descriptor representing the connection with the client. The original listen socket descriptor is available to a new select() call.

Passing Sockets

This section contains information about passing sockets, and includes concepts and tasks.

Common Interface Concepts

To help you better understand socket passing, the following sections explain common interface concepts.

- **Blocking versus Nonblocking**

  A socket is in blocking mode when an I/O() call waits for an event to complete. If blocking mode is set for a socket, the calling program is suspended until the expected event completes.

  If nonblocking is set by calls FCNTL() or IOCTL(), the calling program continues even though the I/O() call might not have completed. If the I/O() call could not be completed, it returns with ERRNO 35 (EWOULDBLOCK). The calling program should use select() to test for completion of any socket call returning an ERRNO 35.

  The default mode is blocking.
- If data is not available for the socket, and the socket is in blocking and synchronous modes, the read() call blocks the caller until data arrives.
- **Concurrent Servers versus Iterative Servers**

  An iterative server handles one client at a time. A concurrent server receives connection requests from multiple clients and creates subtasks to process those client requests.

  When a subtask is created, the concurrent server gets a new socket, passes the new socket to the subtask, and disassociates itself from the connection. (The CICS listener program is an example of a concurrent server.)

  - To pass a socket, the concurrent server first calls givesocket(). If the subtask address space name and subtask ID are specified in the givesocket() call, only a subtask having a matching address space and subtask ID can take the socket. If this field is set to blanks, any MVS address space requesting a socket can take this socket.
  - The concurrent server starts the subtask and passes to it the socket descriptor and concurrent server ID obtained from earlier socket() and getclientid() calls.
  - The subtask calls takesocket() using the concurrent server ID and socket descriptor.
  - The concurrent server issues the select() call to test the socket for the takesocket-completion exception condition.
  - When takesocket() has successfully completed, the concurrent server issues the close() call to free the socket.
- If the queue has no pending connection requests, accept() blocks the socket when blocking mode is set on. You can set the socket to nonblocking by calling FCNTL or IOCTL.
- Issuing a select() call before the accept() call ensures that a connection request is pending. Using the select() call in this way prevents the accept() call from blocking.
- TCP/IP does not screen clients, but you can control the connection requests accepted by closing a connection immediately after you determine the identity of the client.
- A given TCP/IP host can have multiple aliases and multiple host internet addresses.

For more information about sockets, see *UNIX Programmer's Reference Manual*.

A server handling more than one client simultaneously acts like a dispatcher at a messenger service. A messenger dispatcher gets telephone calls from people who want items delivered, and the dispatcher sends out messengers to do the work. In a similar manner, the server receives client requests, and then spawns tasks to handle each client.

Tasks can pass sockets with the givesocket() and takesocket() calls. The task passing the socket uses givesocket(), and the task receiving the socket uses takesocket(). The following sections describe these processes.

givesocket and takesocket

In the UNIX operating system, a new process is dispatched with the fork() system call after the server has established the connection; the new process automatically inherits the socket attached to the client. In MVS, an independent task is started using the attach() supervisor call instruction. A server can perform an attach() call for of a subtask after each connection is established in a way similar to the UNIX operating system, or it can request an attach() several times when it begins execution and pass clients to tasks that already exist. In either case, the server must manually give the new socket to the subtask. Because each task has its own socket table, it is not sufficient to pass only the socket number to the subtask. Socket Number 4 for Task A is not the same as socket Number 4 for Task B.

For C programs using TCP/IP for MVS, each task is given a unique eight-byte name. The task uses the getclientid() call to determine its unique name. The main server task passes the following arguments to the givesocket() call:

- The socket number it wants to give
- Its own name

- The name of the task to which it wants to give the socket

If the server does not know the name of the subtask to receive the socket, it blanks out the name of the subtask. The first subtask calling takesocket() using the server unique name receives the socket. However, the subtask must know the main task unique name, and the number of the socket it is to receive. This information can be passed in a common work area that you define.

When takesocket() acquires the socket, it assigns a new socket number for the subtask to use, but the new socket number represents the same line of communication as the parent socket. The transferred socket can be referred to as socket Number 4 by the parent task, and as socket Number 3 by the subtask. However, both sockets represent the same connection to the TCPIP address space.

After the socket has successfully been transferred, the TCPIP address space posts an exception condition on the parent socket. The parent uses the select() call to test for this condition. After the notification, the parent task must issue close() call on its socket to deallocate the socket.

“Appendix A. Multitasking C Socket Sample Program” on page 535 contains examples of a server, a subtask, and a client. Three examples are written in C, and one example is written in System/370 assembler language.

The C sample programs are included as members of the file *hlq*.SEZAINST partitioned data set; their member names are:

- MTCSRVR
- MTCCSUB
- MTCCLNT

For information about the JCL needed to use the multitasking facility (MTF), see *IBM C/370 User’s Guide*.

Giving a Socket to a Subtask

The socket represented by the new socket descriptor has to be passed to an available subtask. Which technique the main process uses to find an available subtask is not important. Assume that the main process has located an available subtask to which it gives the socket by way of a givesocket() call as shown in Figure 23 on page 52:

```
*----------------------------------------------------------------------*
*Give socket to subtask                                                *
*----------------------------------------------------------------------*
      MVC   CLNNAME,TPIMCNAM    *Our Client ID Address Space Name
      MVC   CLNTASK,TPISTCBE    *Give to this subtask
      EZASMI TYPE=GIVESOCKET,   *Givesocket                         C
            S=NEWSOC,           *Give this socket descriptor        C
            CLIENT=CLNSTRUC,    *- to a specific child process      C
            ERRNO=ERRNO,                                            C
            RETCODE=RETCODE,                                        C
            ERROR=EZAERROR
            ICM   R15,15,RETCODE *OK ?
            BM    EZAERROR      *- No, tell about it.
*
*      CLNSTRUC DS    0F         *GIVESOCKET: Client structure
CLNFAM   DC    A(2)              *TCP/IP Addressing family
CLNNAME  DC    CL8' '            *Address space name of target
CLNTASK  DC    CL8' '            *Task ID of child process subtask
CLNRESV  DC    XL20'00'          *Reserved
*
NEWSOC   DC    AL2(0)            *Socket descriptor from Accept
*
ERRNO    DC    A(0)              *Error number from EZASMI
RETCODE  DC    A(0)              *Returncode from EZASMI
```

Figure 23. Giving a Socket to a Subtask

If you are programming in C, you might not be able to determine the full client ID of the subtask. In that case, you can pass the task ID field as eight blanks on the givesocket() call, which means that any task within your own address space can take the socket, but only the task to which you pass the socket descriptor number will actually take it.

After you have issued the givesocket() call, you must include the given socket descriptor in the exception select mask on the next select() call.

Your main process is now ready to wake up the selected subtask by way of a post system call.

If no other sockets were selected on the previous select() call, your main process can build a new set of select masks, and issue a new select() call.

Taking Sockets from the Main Process

As shown in Figure 20 on page 43, the subtask is reactivated by the post() call issued from the main process; it immediately issues a takesocket() call to receive the socket passed from the main process. Figure 24 on page 53 illustrates this process.

```
*-------------------------------------------------------------------*
* Take socket from main process                                     *
*-------------------------------------------------------------------*
   EZASMI TYPE=TAKESOCKET,    *Takesocket                           C
          CLIENT=TPIMCLNI,    *Main task client id structure        C
          SOCRECV=TPISSOD,    *Main task socket descriptor          C
          ERRNO=ERRNO,                                              C
          RET CODE=RETCODE,                                         C
          ERROR=EZAERROR
   ICM    R15,15,RETCODE      *Did we do well ?
   BM     EZAERROR            *- No, deal with it.
   STH    R15,TPISNSOD        *Server subtask socket descr.no
*
TPIMCLNI DS    0C             *Main task client id
TPIMCDOM DC    A(0)           *Domain: AF-INET
TPIMCNAM DC    CL8' '         *Our address space name
TPIMCTSK DC    CL8' '         *Main task TCB address in EBCDIC
         DC    20X'00'        *Reserved (part of clientid)
*
TPISSOD  DC    AL2(0)         *Parent socket descr. no.
TPISNSOD DC    AL2(0)         *Subtask socket descr. no.
```

Figure 24. Taking Sockets from the Main Process

In order to take a socket, the subtask must know the client ID of the task that gave the socket, and the socket descriptor used by that task. These values must be passed to the subtask from the main process before a takesocket() call can be issued.

On the takesocket() call, you specify the full client ID of the process that gave the socket, and you specify the socket descriptor number used by the process that gave the socket.

A new socket descriptor number to be used by the subtask is returned in the RETCODE when the takesocket() call is successful. As soon as your subtask has taken the socket, the main process is posted in its pending select with a pending exception activity; this means that the main process must close its socket descriptor.

In Figure 24, the client sends its request to the subtask, which processes it and sends back a reply.

Finally, the client process and the server subtask close their sockets, and the server subtask re-enters wait-for-work status.

Transferring Data between Sockets

See “Chapter 7. Transferring Data between Sockets” on page 61.

Closing a Concurrent Server Program

See “Chapter 3. Designing an Iterative Server Program” on page 25.

Chapter 5. Designing a Client Program

This chapter explains how to design a client program. The following topics are included:

- Allocating a socket
- Connecting to a server
- Transferring data between sockets
- Closing a client program

Allocating a Socket

From their own perspective, clients must first issue the socket() call to allocate a socket from which to communicate as follows:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

For more information, see “Allocating Sockets” on page 25.

Connecting to a Server

To connect to a server, the client must know the server name. This section describes how to determine a server name, and connect to that server.

Note: Examples are written in C language and REXX.

To connect to the server, the client places the port number and the IP address of the server into a sockaddr_in structure like the bind() call. If the client does not know the server IP address, but it does know the server host name, the gethostbyname() call is called to translate the host name into its IP address.

The client then calls connect() as shown in the following C language example of the connect() call:

```
connect(s, name, namelen);
```

When the connection is established, the client uses its socket to communicate with the server.

If you need to determine a server name while writing in REXX and you know only the host name, you must resolve the host name into one or more IP addresses using the gethostbyname() call as shown in Figure 25.

```
 /*--------------------------------------------------------------*/
 /* Find IP addresses of server host                             */
 /*--------------------------------------------------------------*/
 servipaddr = DoSocket('Gethostbyname', tpiserver)
 if sockrc <> 0 then do
    say 'Gethostbyname failed, rc='sockrc
    say sockval
    x=Doclean
    exit(sockrc)
end
```

Figure 25. Finding the IP Address of a Server Host Using gethostbyname()

The REXX gethostbyname() call returns a list of IP addresses if the host is a multihomed host. You can parse the REXX string and place the IP addresses into a REXX stem variable using the following piece of REXX code:

```
/*---------------------------------------------------------------*/
/* Parse returned IP address list                                */
/*---------------------------------------------------------------*/
numips = words(servipaddr)
do i = 1 to numips
  sipaddr.i = word(servipaddr, i)
end
sipaddr.0 = numips
```

When you issue a connect call to an IP address currently not available, your connect call eventually times out, with an error number of 60 (ETIMEDOUT). The socket you used on such a failed connect call cannot be reused for another connect() call. If you try to do so, you receive error number 22 (EINVAL). You have to close the existing socket and get a new socket before you reissue the connect call using the next IP address in the list of IP addresses returned by the gethostbyname() call.

The connect call can be placed in a loop that terminates when a connect is successful, or the list of IP addresses is exhausted. The following sample illustrates this process.

```
 /*---------------------------------------------------------------*/
 /*                                                               */
 /* Get a socket and try to connect to the server                 */
 /*                                                               */
 /* If connect fails (ETIMEDOUT), we must close the socket,       */
 /* get a new one and try to connect to the next IP address       */
 /* in the list, we received on the gethostbyname call.           */
 /*                                                               */
 /*---------------------------------------------------------------*/
i=1
connected = 0
do until (i > sipaddr.0 | connected)
   sockdescr = DoSocket('Socket')
   if sockrc <> 0 then do
      say 'Socket failed, rc='sockrc
      exit(sockrc)
   end
   name = 'AF_INET '||tpiport||' '||sipaddr.i
   sockval = DoSocket('Connect', sockdescr, name)
   if sockrc = 0 then do
      connected = 1
   end
   else do
      sockval = DoSocket('Close', sockdescr)
      if sockrc <> 0 then do
         say 'Close failed, rc='sockrc
         exit(sockrc)
      end
   end
   i = i + 1
end
if connected then do
   say 'Connect failed, rc='sockrc
   exit(sockrc)
end
```

Transferring Data between Sockets

See “Chapter 7. Transferring Data between Sockets” on page 61.

Closing a Client Program

See “Chapter 3. Designing an Iterative Server Program” on page 25.

Chapter 6. Designing a Program to Use Datagram Sockets

This chapter explains how to design a program to use datagram sockets. Topics include:

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

Datagram Socket Characteristics

The most significant characteristics of datagram sockets follow:

- Datagram sockets are connectionless.

  There is no connection setup effected by the UDP protocol layer. No data is exchanged between sending and receiving UDP protocol layers until your application issues its first send call.

  If your UDP server program has not been started, or it resides on a host that cannot be reached from your client host, your client UDP application can wait forever to receive a reply to the datagram it sent to a UDP server. You have to implement time-out logic in your client UDP program to recognize this situation.
- The UDP protocol layer does not implement reliability functions.

  The implicit significance of this fact is that a datagram sent from one UDP program to another might never arrive. Neither the sending program nor the target program ever learns from the UDP protocol layer that such a condition exists.

  If your UDP application must be reliable, you must add reliability code to your UDP client and server programs. Such code must include the ability to detect missing datagrams, datagrams arriving out of sequence, duplicate datagrams, and corrupt datagrams.

  Because implementation of such function is complicated, it is recommend that you use TCP protocols instead of UDP protocols if your application must be reliable.
- Unlike a TCP socket, where there is no one-to-one relationship between send() and recv() calls, UDP socket send corresponds exactly to a UDP socket recv() call.

Understanding Datagram Socket Program Structure

The datagram socket program terms client and server can be misleading. Two socket programs that have each bound a socket to a local address can send any number of datagrams to each other in any sequence. The program that sends the first data will act as a client. Any datagram sent to a destination address for which no program has bound a socket is lost. Care must be taken so that the program you intend to be the client does not begin sending datagrams until the server program has bound its socket to the destination address expected.

Typically, the structure for a datagram socket resembles the iterative server discussed in “Chapter 3. Designing an Iterative Server Program” on page 25.

Allocating a Socket

See “Allocating Sockets” on page 25.

Binding Sockets to Port Numbers

The server program must bind its socket to a predefined server port number, so the clients know the port to which they should send their datagrams. In the socket address structure that the server passes on the bind() call, it can specify if it will accept datagrams from the available network interfaces, or whether only from a specific network interface. This is done by setting the IP address field of the socket address structure to either INADDR_ANY, or a specific IP address.

The client program needs to bind its socket to a local address if it wants the server program to be able to return a datagram to it. In contrast to the server, the client does not need to specify a specific port number on the bind() call; an ephemeral port number chosen by the UDP protocol layer is sufficient; this is called a dynamic bind.

Streamline Data Transfer Using Connect Call

While you can use the connect() call on a datagram socket, it does not act for a datagram socket as it acts for a stream socket.

On a connect() call, you specify the remote socket address with which you want to exchange datagrams. This serves the following purposes:

- On succeeding calls to send datagrams, you can use the send() call without specifying a destination socket address; the datagram is sent to the socket address you specified on the connnect() call.
- On succeeding calls to receive datagrams, only datagrams that originate from the socket address specified on the connect() call are passed to your program from the UDP protocol layer.

Note: A connect() call for a datagram socket does not establish a connection. No data is exchanged over the IP network as the result of the connect() call. The functions performed are local, and control is returned immediately to your application.

Transferring Data between Sockets

See “Chapter 7. Transferring Data between Sockets” on page 61.

Chapter 7. Transferring Data between Sockets

This chapter contains information about transferring data between sockets, and includes the following topics:

- Overview
- Streams and messages
- Data representation
- Transferring data between connected sockets
- Using sendto() and recvfrom() calls

Overview

Transferring data over a datagram socket is similar to working with MVS records. You send and receive data records. One send() call results in exactly one recv() call.

If your sending program sends a datagram of 8192 bytes, and your receiving program issues a recv() call, in which it specifies a buffer size of, for example, 4096 bytes, it will receive the 4096 bytes it requested; the remaining 4096 bytes in the datagram are discarded by the UDP protocol layer without further notification to either sender or receiver.

IBM Communications Server for OS/390 V2R10 includes a performance enhancement that when both the source and destination of a packet are known to and managed by a single TCP/IP stack, the IP layer can be bypassed. This provides an overall pathlength savings when processing such packets and the decrease in pathlength through the stack results in an overall throughput improvement for applications that reside on the same MVS systems and communicate with each other through the same TCP/IP stack. Socket application programmers can take advantage of this performance enhancement by using a non-loopback home address when sending data between applications that reside on the same MVS system and communicate with each other through the same TCP/IP stack. See the *OS/390 IBM Communications Server: IP Migration* for additional information.

Streams and Messages

This section describes how to design an application protocol so that the partner program can divide the receive stream into individual messages.

Some socket applications are simple, and the receiver can continue to receive data until the sender closes the socket; for example, a simple file transfer application. Most applications are not that simple, and usually require that the stream can be divided into a number of distinct messages.

A message exchanged between two socket programs must imbed information so that the receiver can decide how many bytes to expect from the sender and (optionally) what to do with the received message.

A few common techniques are used to imbed information about the length of a message into the stream, as follows:

- The message type identifier technique

If your messages are fixed length, you can implement a message ID per message type worked with. Each message type has a predefined length that is known by your client and server programs. If you place the message ID at the start of each message, the receiving program can determine how long the message is if it knows the content of the first few bytes in the message. This is illustrated in Figure 26.

```
*---------------------------------------------------------------*
* Layout of a message between TPI client and TPI server         *
*---------------------------------------------------------------*
 01  tpi-message.
     05  tpi-message-id             pic x
     88  tpi-request-add        value '1'
     88  tpi-request-update     value '2'
     88  tpi-request-update     value '2'
     88  tpi-request-query      value '3'
     88  tpi-request-query      value '3'
     88  tpi-request-delete     value '4'
     88  tpi-query-reply        value 'A'
     88  tpi-response           value 'B'
     05  tpi-constant           pic x(4)
     88  tpi-identifier         value 'TPI '
```

Figure 26. Layout of a Message between a TPI Client and a TPI Server

Each message ID is associated with a fixed length known to your application.

- The record descriptor word (RDW) technique

  If your messages are variable length, you can implement a length field in the beginning of each message. Normally, you would implement the length in a binary half-word with the value encoded in network byte order, but you can implement it as a text field, as shown in Figure 27.

```
*---------------------------------------------------------------*
* Transaction Request Message segment                           *
*---------------------------------------------------------------*
01  TRM-message.
    05  TRM-message-length   pic 9(4) Binary Value 20
    05  filler                     pic x(2) Value low-value
    05  TRM-identifier             pic x(8) Value '*TRNREQ*'
    05  TRM-trancode               pic x(8) Value '?????'.
```

Figure 27. Transaction Request Message Segment

- The end-of-message marker technique

  A third technique most often seen in C programs is to send a null-terminated string. A null-terminated string is a string of bytes terminated by a byte of binary zero. The receiving program reads whatever data is on the stream and then loops through the received buffer separating each record at the point where a null-byte is found. When the received records have been processed, the program issues a new read for the next block of data on the stream.

  If your messages contain only character data, you can designate any non-display byte value as your end-of-message marker. Although this technique is most often seen in C programs, it can be used with any programming language.

- The TCP/IP buffer flushing technique

  This technique is based on the observed behavior of the TCP protocol, where a send() call followed by a recv() call forces the sending TCP protocol layer to flush its buffers, and forward whatever data might exist on the stream to the receiving TCP protocol layer. You can use this method to implement a half-duplex, flip-flop application protocol, where your two partner programs acknowledge the receipt of each message with, for example, a one-byte application acknowledgment message.

Figure 28 shows the TCP buffer flush technique.

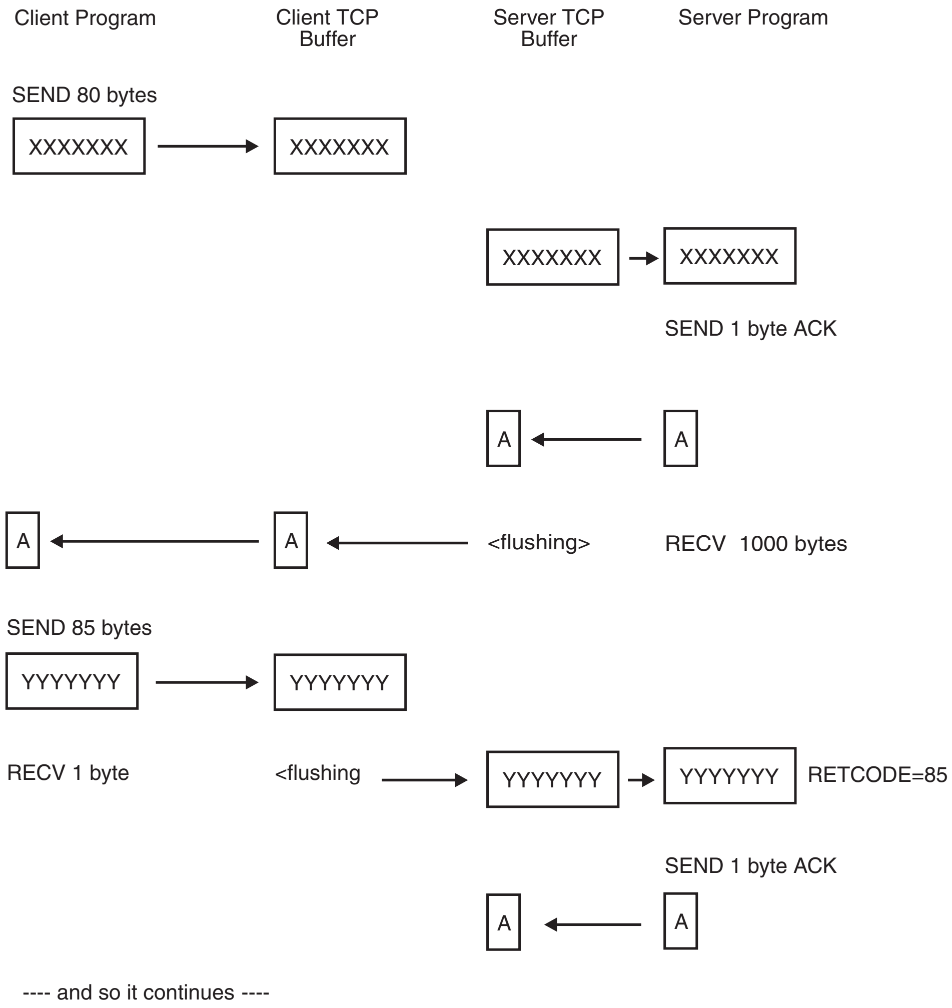


Figure 28. The TCP Buffer Flush Technique

In Figure 28, the client sends an 80 byte message. The server has issued a recv() call for 1000 bytes, but receives only the 80 bytes (RETCODE=80). This presents a problem because there is no guarantee the server will receive the full 80-byte message on its receive call. It might only receive 30 bytes; but, with this technique, it has no way of knowing that it is missing another 50 bytes. The smaller the messages are, the less likely the server will receive only a part of the full message.

Note: This technique is widely used, but you should use it only in controlled environments, or in programs where you use non-blocking socket calls to implement your own time-out logic.

The message type identifier and the record descriptor word techniques require that the receiving program be able to learn the content of the first bytes in the message before it reads the entire message.

If this is a problem for your application, use the peek flag on a recv socket() call.

A recv() call with the peek flag on does not remove the data from the TCP buffers, but copies the number of bytes you requested into the application buffer you specified on the recv() call.

For example, if your message length field or message ID field is located within the first five bytes of each message, issue the following recv() call:

```
*---------------------------------------------------------------*
* Peek buffer and length fields for RECV call                   *
*---------------------------------------------------------------*
01  soket-recv                     pic x(16) value 'RECV            '
01  recv-flag-peek                 pic 9(8) binary value 2
01  recv-peek-len                  pic 9(8) binary value 5
01  recv-peek-buffer.
    05  message-id                 pic x value space
        88  tpi-query-reply        value 'A'
        88  tpi-response           value 'B'
    05  message-constant           pic x(4)
        88  tpi-identifier         value 'TPI '
01  socket-descriptor              pic 9(4) binary value  0
01  errno                          pic 9(8) binary value  0
01  retcode                        pic s9(8) binary value 0
*---------------------------------------------------------------*
* Peek at first 5 bytes of client data                          *
*---------------------------------------------------------------*
   call 'EZASOKET' using soket-recv
       socket-descriptor
       recv-flag-peek
       recv-peek-len
       recv-peek-buffer
       errno
       retcode
   if retcode < 0 then
      - process error -
   if retcode = 0 then
      - process client closed socket -
   if not TPI-identifier then
      - translate recv-peek-buffer from ASCII to EBCDIC -
```

The recv() call blocks until some bytes have been received, or the sender closes its socket. The above example is not complete since you cannot be sure that you actually received the five bytes requested. Your call might come back to you with only one byte received. In order to manage the situation, you need to repeat your recv() call until all five bytes have been received, and recognized as such.

If the other half of the connection closes the socket, the recv() call returns zero in the *retcode* field.

The data is copied into your application program buffer only; it is still available to a recv() call, in which you can specify the full length of the message you now know to be available.

Data Representation

If you use the socket API, your application must handle the issues related to different data representations occurring on different hardware platforms. For character based data, some hosts use ASCII, while other hosts use EBCDIC. Translation between the two representations must be handled by your application.

For integers, some hardware platforms use big endian byte order (S/370/390, Motorola style), while others use little endian byte order (Intel style). An example of the difference between big and little endian byte orders is shown in Figure 29.

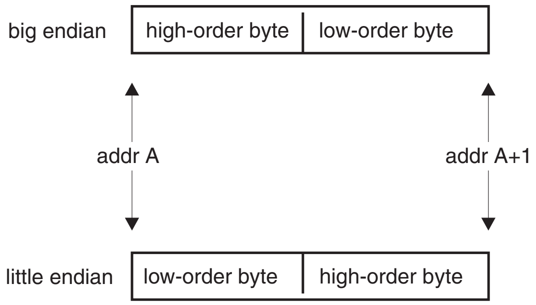


Figure 29. Big or Little Endian Byte Order for a 2-Byte Integer

IBM S/370- and IBM S/390-based computers all use big endian byte order, while the IBM PS/2 uses the little endian byte order. TCP/IP has defined a network byte order standard to be used for all 16-bit and 32-bit integers that appear in protocol headers. This network byte order is based on the big endian byte order. This is the reason, in the C-socket interface, you find the following:

htons Translates a short integer (two bytes) from host byte order to network byte order

ntohs Translates a short integer from network byte order to host byte order

htonl Translates a long integer (four bytes) from host byte order to network byte order

ntohl Translates a long integer from network byte order to host byte order

The socket-based application should manage the application data portion of a message. If you develop a server that serves clients on different hardware platforms, define your own standard and implement it as part of your application protocol.

In some cases, it is easier to base your messages on text data. If you, as part of your message design, define a fixed text string in the beginning of each message, your application can test the contents of this string and decide whether the data is in EBCDIC or ASCII. If the data is in ASCII, you can translate the full message from ASCII to EBCDIC on input, and translate from EBCDIC to ASCII on output from MVS. An example of this design is the transaction request message (TRM) format used by the IMS Listener program. Bytes four to 11 have a fixed value of *TRNREQ*, which is used both to distinguish this message from other messages and to find out whether the client is transmitting data in ASCII or EBCDIC.

If you mix text data and binary data in your messages, be sure to only apply translation between ASCII and EBCDIC to the text fields in your message.

If you use binary integer fields in your messages, it is recommended that you use the network byte order standard, which TCP/IP uses for all integers in protocol headers. If you design your messages according to the network byte order standard, your MVS programs do not need to translate or rearrange the bytes in binary integer fields. Your programs executing on little endian hosts must use the integer conversion routines to convert integers between local format and the format used in the messages they exchange with your MVS programs.

Text data and binary two and four byte integers are easy to handle in a heterogeneous computer environment. In more complex data types like floating point numbers or packed decimal, it becomes much more complicated because there is no generally accepted standard, and there is no easy support for transformation between the formats. If you include these data types in your messages, be sure that the partner program knows how to interpret them. If the two computer systems use the same architecture, this is valid. If you exchange messages by way of socket programs between two MVS systems, you do not need to be concerned about conversion.

Using send() and recv() Calls

This section provides information about sending and receiving calls.

The Conversation

Client and server communicate using send() and recv() as shown below:

```
num = send(s, addr_of_data, len_of_data, 0);
num = recv(s, addr_of_buffer, len_of_buffer, 0);
```

The send() and recv() calls specify:

- The socket *s* on which to communicate
- The address in storage of the buffer that contains, or will contain, the data (addr_of_data, addr_of_buffer)
- The size of this buffer (len_of_data, len_of_buffer)
- A flag that tells how the data is to be sent

Flag 0 tells TCP/IP to transfer the data normally. The server uses the socket that is returned from the accept() call.

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

When the conversation is over, both the client and the server call close() to end the connection. Close() also deallocates the socket, freeing its space in the table of connections. To end a connection with a specific client, the server closes the socket returned by accept(). If the server closes its original socket, it can no longer accept new connections, but it can still converse with the clients to which it is connected. The close() call is represented as follows:

```
close(s);
```

If you are writing a client application, you might want to verify the processes the server will use. Both client applications, and the servers with which they communicate, must be aware of the sequence of events each will follow.

Using Socket Calls in a Network Application

You can use the following example to write a socket network application. The example is written using C socket syntax conventions, but the principles illustrated here apply to all the APIs in this book.

Clients and servers wanting to transfer data have many calls from which to choose. The read() and write(), readv() and writev(), and the send() and recv() calls can be used only on sockets that are connected. The sendto() and recvfrom(), and sendmsg() and recvmsg() calls can be used at any time. The example listed in Figure 30 illustrates the use of send() and recv() calls.

```
int send(int socket, char *buf, int buflen, int flags);
int recv(int socket, char *buf, int buflen, int flags);
⋮
int bytes_sent;
int bytes_received;
char data_sent[256];
char data_received[256];
int s;
⋮
bytes_sent = send(s, data_sent, sizeof(data_sent), 0);
⋮
bytes_received = recv(s, data_received, sizeof(data_received), 0);
```

Figure 30. An Application Using the send() and recv() Calls

The example in Figure 30 shows an application sending data to a connected, socket and receiving data in response. The flags field can be used to specify additional options to send() or recv(), such as sending out-of-band data. For more information about these routines, see the following:

- “read()” on page 145
- “readv()” on page 146
- “recv()” on page 148
- “send()” on page 160
- “write()” on page 186
- “writev()” on page 187

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

read and write
These calls can only be used with connected sockets. No processing flags can be passed on these calls.

recv and send
These calls also work with connected sockets only. You can pass processing flags on these calls:

- NOFLAG — read or write data as a read call or a write call would.
- OOB — read or write Out Of Band data (expedited data).
- PEEK — peek at data, but do not remove data from the buffers.

recvfrom and sendto
These calls work with both connected and non-connected sockets. You can pass addressing information directly (as parameters) on these calls. The available flags are the same as those for recv and send.

A connected socket is either a stream socket for which a connection has been established, or it is a datagram socket, for which you have issued a connect() call to specify the remote datagram socket address.

Reading and Writing Data from and to a Socket

Stream sockets during read and write calls might behave in a way that you would expect to be an error. The read() call might return fewer bytes, and the write() call may write fewer bytes, than requested. This is not an error, but a normal situation that your programs must deal with when they read or write data over a socket.

You might need to use a series of read calls to read a given number of bytes from a stream socket. Each successful read() call returns in the retcode field the number of bytes actually read. If you know you have to read, for example, 4000 bytes and the read call returns 2500, you have to reissue the read call with a new requested length of 4000 minus the 2500 already received (1500).

If you develop your program in COBOL, the following example shows an implementation of such logic. In this example, the message to be read has a fixed size of 8192 bytes.

```
 *---------------------------------------------------------------*
 * Variables used by the READ call                               *
 *---------------------------------------------------------------*
 01  read-request-read               pic 9(8) binary value 0
 01  read-request-remaining          pic 9(8) binary value 0
 01  read-buffer.
     05  read-buffer-total           pic x(8192) value space.
     05  read-buffer-byte redefines read-buffer-total
                                    pic x occurs 8192 times.
*---------------------------------------------------------------*
* Read 8K block from server                                     *
*---------------------------------------------------------------*
    move zero to read-request-read
    move 8192 to read-request-remaining
    Perform until read-request-remaining = 0
       call 'EZASOKET' using soket-read
          socket-descriptor
          read-request-remaining
          read-buffer-byte(read-request-read + 1)
          errno
          retcode
       if retcode < 0 then
          - process error and exit -
       end-if
       add retcode to read-request-read
       subtract retcode from read-request-remaining
       if retcode = 0 then
          Move zero to read-request-remaining
       end-if
    end-perform
```

An actual execution of the program, following the above logic, used four read calls to retrieve 8K of data. The first call returned 1960 bytes, the second call 3920 bytes, the third call returned 1960 bytes and the final call 352 bytes. It is not possible to predict how many calls will be needed to retrieve the message. That depends on the internal buffer utilization of a TCP/IP. In some cases, only two calls were needed to retrieve 8K of data.

It is good programming practice, whenever you know the number of bytes to read, to issue read calls imbedded in logic, which is similar to the method described above.

If you work with short messages, you usually receive the full message on the first read() call, but there is no guarantee.

The behavior of a write() call is similar to that of a read() call. You might need to repeat more write() calls to write out all the data you want written. The following example illustrates this technique.

```
*---------------------------------------------------------------*
* Buffer and length fields for write operation                  *
*---------------------------------------------------------------*
01  send-request-sent              pic 9(8) binary value 0
01  send-request-remaining         pic 9(8) binary value 0
01  send-buffer.
    05  send-buffer-total          pic x(8192) value space.
    05  send-buffer-byte redefines send-buffer-total
                                   pic x occurs 8192 times.
*---------------------------------------------------------------*
* Send 8K data block                                            *
*---------------------------------------------------------------*
    move 8192 to send-request-remaining
    move 0 to send-request-sent
    Perform until send-request-remaining = 0
       call 'EZASOKET' using soket-write
            socket-descriptor
            send-request-remaining
            send-buffer-byte(send-request-sent + 1)
            errno
            retcode
       if retcode < 0 then
             - process error and exit -
       end-if
       add retcode to send-request-sent
       subtract retcode from send-request-remaining
    if retcode = 0 then
       Move zero to send-request-remaining
    end-if
 end-perform
```

Using sendto() and recvfrom() Calls

If the socket is not in a connected state, additional address information must be passed to sendto(), and can be (optionally) returned from recvfrom(). An example of the sendto() and recvfrom() calls is listed in Figure 31 on page 70.

```
int sendto(int socket, char *buf, int buflen, int flags,
           struct sockaddr *addr, int addrlen);
int recvfrom(int socket, char *buf, int buflen, int flags,
           struct sockaddr *addr, int *addrlen);
⋮
int bytes_sent;
int bytes_received;
char data_sent[256];
char data_received[256];
struct sockaddr_in to;
struct sockaddr from;
int addrlen;
int s;
⋮
memset(&to, 0, sizeof(to));
to.sin_family = AF_INET;
to.sin_addr   = inet_addr("129.5.24.1");
to.sin_port   = htons(1024);
⋮
bytes_sent = sendto(s, data_sent, sizeof(data_sent), 0,
           (struct sockaddr*)&to, sizeof(to));
⋮
addrlen = sizeof(from); /* must be initialized */
bytes_received = recvfrom(s, data_received,
   sizeof(data_received), 0, &from, &addrlen)
```

Figure 31. An Application Using the sendto() and recvfrom() Calls

The sendto() and recvfrom() calls take additional parameters to allow the caller to specify the recipient of the data, or to be notified of the sender of the data. See “recvfrom()” on page 150, “sendmsg()” on page 162, and “sendto()” on page 164 for more information about these additional parameters. Usually, sendto() and recvfrom() are used for datagram sockets, and send() and recv() are used for stream sockets.

Part 3. Application Program Interfaces

Chapter 8. C Socket Application Programming Interface (API)

Note: Use of the UNIX C socket library is encouraged. For more information, refer to the *OS/390 C/C++ Run-Time Library Reference*.

This chapter describes the C socket application program interface (API) provided with TCP/IP. Use the socket routines to interface with the TCP, UDP, and IP protocols. The socket routines allow you to communicate with other programs across networks. You can, for example, use socket routines when you write a client program that must communicate with a server program running on another computer.

Topics include:

- Compiler restrictions
- Compiling and linking C applications
- Compiler messages
- Program abends
- C socket implementation
- C socket header files
- C structures
- Error messages and return codes
- C socket calls
- Sample C socket programs

To use the C socket API, you must know C language programming. For more information about C language programming, see *OS/390 C/C++ Programming Guide*.

Compiler Restrictions

This section tells you how to move your application to the CS for OS/390 system.

- When you need to recompile, use the compiler shipped with this product.
- All applications linked to the TCP/IP C sockets library must run on the LE run-time library shipped with CS for OS/390.
- To access system return values, you need only use include statement errno.h supplied with the compiler. To access network return values, you must add the following include statement:

  ```
  #include <tcperrno.h>
  ```
- To print system errors only, use perror(), a procedure available from the C compiler run-time library. To print both system and network errors, use tcperror(), a procedure provided by IBM and included with CS for OS/390.

  Note to CICS users:

  Do not use tcperror(); rather, add statement #include <ezacichd.h>, and compile the statement as non-reentrant. For more information, see the section on C Language Programming in the *TCP/IP for MVS: CICS TCP/IP Socket Interface Guide and Reference*.
- If your C language statements contain information, such as sequence numbers, that are not part of the input for the C/C^{++} compiler, you must exclude that information during compilation. The C/C^{++} compiler provides several ways to do this, one of which is:

```
#pragma margins (1,72)
```

In this example, we are presuming you have sequence numbers in columns 73 through 80.

- By default, prototype C socket functions and their parameters for the current release are defined. If you need to access the TCP/IP V3R1 definitions, specify the following during a compile:

  ```
  #define_TCP31_PROTOS
  ```

Compiling and Linking C Applications

There are several ways to compile, link-edit, and execute CS for OS/390 C source program in MVS. To run a C source program under MVS batch using IBM supplied cataloged procedures, you must include data sets. This section contains information about the data sets that you must include.

The following data set name is used as an example in the sample JCL statements.

USER.MYPROG.H
Contains user #include files

Non-Reentrant Modules

You must make additions to the compile step of your catalogued procedure to compile a non-reentrant module. The following lines describe these additions. Catalogued procedures are included in the IBM-supplied samples for your MVS system.

Note: Compile all C code source using the def(MVS) preprocessor symbol.

- Add the following line as the first //SYSLIB DD statement:

  ```
  //SYSLIB DD  DSN=hlq.SEZACMAC,DISP=SHR
  ```

  Note that *hlq* is the high-level qualifier of your TCP/IP system libraries.
- Add the following //USERLIB DD statement:

  ```
  //USERLIB DD  DSN=USER.MYPROG.H,DISP=SHR
  ```

The following lines describe the additions that you must make to the link-edit step of your catalogued procedure to link-edit a non-reentrant module.

- To link-edit programs that use C sockets library functions, add the following statement as the first //SYSLIB DD statement:

  ```
  //SYSLIB DD  DSN=hlq.SEZACMTX,DISP=SHR
  ```

Figure 32 on page 75 shows a sample JCL to be used when compiling non-reentrant modules. Modify the lines to conform to the naming conventions of your site.

```
//COMPIT   JOB ,COMPILE,MSGLEVEL=(1,1)
//********************************************************************
//*                                                                  *
//*  SAMPLE JCL THAT COMPILES A TEST PROGRAM  AS NORENT              *
//*       USING THE C/C++ COMPILER C/MVS IN NON-OE ENVIRONMENT       *
//*  INPUT  : USER71.TEST.SRC(&INFILE)                               *
//*  OUTPUT : USER71.TEST.OBJ(&OUTFILE)                              *
//*                                                                  *
//********************************************************************
//*
//CPPC PROC CREGSIZ='4M',
//   INFILE=CTEST,
//   OUTFILE=CTEST,
//   CPARM1=NORENT,
//   CPARM2='LIS,SO,EXP,OPT,DEF(MVS)',
//   DCB80='(RECFM=FB,LRECL=80,BLKSIZE=3200)',
//   DCB3200='(RECFM=FB,LRECL=3200,BLKSIZE=12800)',
//   LIBPRFX1='CEEL.OSV2R7',
//   LIBPRFX2='CEE.OSV2R7',
//   COMPRFX='CBC.OSV2R7'
//*
//*----------------------------------------------------
//*  COMPILE STEP:
//*----------------------------------------------------
//COMPILE EXEC PGM=CBCDRVR,PARM=('&CPARM1','&CPARM2'),
//   REGION=&CREGSIZ
//STEPLIB  DD  DSNAME=&LIBPRFX1..SCEERUN,DISP=SHR
//         DD  DSNAME=&COMPRFX..SCBCCMP,DISP=SHR
//SYSMSGS  DD  DUMMY,DSNAME=&COMPRFX..SCBC3MSG(EDCMSGE),DISP=SHR
//SYSIN    DD  DSNAME=USER71.TEST.SRC(&INFILE),DISP=SHR
//USERLIB  DD  DSN=USER.MYPROG.H,DISP=SHR
//SYSLIB   DD  DSN=TCP.SEZACMAC,DISP=SHR
//         DD DSN=&LIBPRFX2..SCEEH.H,DISP=SHR
//SYSLIN   DD  DSNAME=USER16.TEST.OBJ(&OUTFILE),DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSCPRT  DD  SYSOUT=*
//SYSUT1   DD  UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB80
//SYSUT5   DD  UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT6   DD  UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT7   DD  UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT8   DD  UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT9   DD  UNIT=VIO,SPACE=(32000,(30,30)),
//             DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SYSUT10  DD  SYSOUT=*
//*
//    PEND
//    EXEC PROC=CPPC
```

Figure 32. Sample JCL for Compiling Non-Reentrant Modules.

Figure 33 on page 76 shows a sample JCL to be used when linking non-reentrant modules. Modify the lines to conform to the naming conventions of your site.

```
//LINKIT   JOB ,LINK,MSGLEVEL=(1,1)
//********************************************************************
//*                                                                  *
//*  SAMPLE JCL THAT LINKS A NON_REENTRANT TEST PROGRAM              *
//*       USING THE C/C++ COMPILER C/MVS                             *
//*  INPUT  LIBRARY: USER71.TEST.OBJ(&MEM)                           *
//*  OUTPUT LIBRARY: USER71.TEST.LMOD(&MEM)                          *
//*                                                                  *
//********************************************************************
//EDCL PROC USER=USER71
//TCPIP     EXEC PGM=IEWL,
//     PARM=',MAP,RMODE(ANY),SIZE=(320K,64K)'
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSLMOD  DD DSN=&USER..TEST.LMOD(&MEM),DISP=SHR
//SYSLIN   DD DSN=&USER..TEST.OBJ(&MEM),DISP=SHR
//SYSLIB   DD DSN=TCP.SEZACMTX,DISP=SHR
//         DD DSN=CEE.OSV2R7.SCEELKED,DISP=SHR
//     PEND
//   EXEC EDCL,MEM=CTEST
```

Figure 33. Sample JCL for Linking Non-Reentrant Modules.

Figure 34 shows JCL to be used when running non-reentrant modules. Modify the lines to conform to the naming conventions of your site.

```
//RUNTST  JOB ,RUN,MSGLEVEL=(1,1),CLASS=A,REGION=4096K
//********************************************************************
//*                                                                  *
//*  SAMPLE JCL THAT RUNS A TEST PROGRAM, CTEST                      *
//*                                                                  *
//********************************************************************
//S1  EXEC PGM=CTEST
//STEPLIB DD DSN=CEEL.OSV2R7.SCEERUN,DISP=SHR
//        DD DSN=USER71.TEST.LMOD,DISP=SHR
//SYSPRINT DD  SYSOUT=*
```

Figure 34. Sample JCL for Running Non-Reentrant Modules.

Note: For more information about compiling and linking, see *OS/390 C/C++ Compiler and Run-Time Migration Guide*.

Reentrant Modules

The following lines describe the additions that you must make to the compile step of your catalogued procedure to compile a reentrant module. Catalogued procedures are included in the IBM-supplied samples for your MVS system.

Note: Compile all C source code using the def(MVS) preprocessor symbol.

Be sure to use the RENT compiler option if your code is not usually reentrant.

- Add the following line as the first //SYSLIB DD statement:

  ```
  //SYSLIB DD  DSN=hlq.SEZACMAC,DISP=SHR
  ```

- Add the following //USERLIB DD statement:

  ```
  //USERLIB DD  DSN=USER.MYPROG.H,DISP=SHR
  ```

The following lines describe the additions that you must make to the prelink-edit and link-edit steps of your catalogued procedure to create a reentrant module.

To prelink programs that use the C sockets library function, put the following statement first in the SYSLIB concatenation:

```
//SYSLIB DD DSN=hlq.SEZARNT1,DISP=SHR
```

To link-edit programs that have the C sockets library function, the following statement must be first in the SYSLIB concatenation:

```
//SYSLIB DD DSN=hlq.SEZACMTX,DISP=SHR
```

Notes:

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

Figure 35 on page 78 shows sample JCL to be used when compiling a test program with reentrancy. Modify the lines to conform to the naming conventions of your site.

```
//********************************************************************
//*                                                                  *
//*  SAMPLE JCL THAT COMPILES A TEST PROGRAM, CTEST, AS 'RENT',      *
//*       USING THE C/C++ COMPILER C/MVS                             *
//*                                                                  *
//********************************************************************
//*
//CPPC PROC CREGSIZ='4M',
//   INFILE=CTEST,
//   CPARM1=RENT,
//   CPARM2='LIS,SO,EXP,OPT,DEF(MVS),SHOWINC',
//   DCB80='(RECFM=FB,LRECL=80,BLKSIZE=3200)',
//   DCB3200='(RECFM=FB,LRECL=3200,BLKSIZE=12800)',
//   LIBPRFX1='CEEL.OSV2R7',
//   LIBPRFX2='CEE.OSV2R7',
//   COMPRFX='CBC.OSV2R7'
//*
//*----------------------------------------------------
//*  COMPILE STEP:
//*----------------------------------------------------
//COMPILE EXEC PGM=CBCDRVR,PARM=(,
//   '&CPARM1','&CPARM2'),REGION=&CREGSIZ
//STEPLIB  DD  DSNAME=&LIBPRFX1..SCEERUN,DISP=SHR
//         DD  DSNAME=&COMPRFX..SCBCCMP,DISP=SHR
//SYSMSGS  DD  DUMMY,DSNAME=&COMPRFX..SCBC3MSG(EDCMSGE),DISP=SHR
//SYSIN    DD  DSNAME=USER71.TEST.SRC(&INFILE),DISP=SHR
//USERLIB  DD  DSN=USER.MYPROG.H,DISP=SHR
//SYSLIB   DD  DSN=TCP.SEZACMAC,DISP=SHR
//         DD DSN=&LIBPRFX2..SCEEH.H,DISP=SHR
//SYSLIN   DD  DSNAME=USER71.TEST.RENTDS,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSCPRT  DD  SYSOUT=*
//SYSUT1   DD  UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB80
//SYSUT5   DD  UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT6   DD  UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT7   DD  UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT8   DD  UNIT=VIO,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT9   DD  UNIT=VIO,SPACE=(32000,(30,30)),
//             DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SYSUT10  DD  SYSOUT=*
//*
//    PEND
//    EXEC PROC=CPPC
```

Figure 35. Sample JCL for Compiling Reentrant Modules

Figure 36 on page 79 shows sample JCL to be used when prelinking and linking a reentrant program using the C socket library. Modify the lines to conform to the naming conventions of your site.

```
//********************************************************************
//*
//*   PRE-LINK AND LINK FOR REENTRANCY WITH C/C++ COMPILER C/MVS,
//*        OS390 RUNTIME LIBRARY.
//*   NOTES:
//*    - SPECIFY 'RENT' ON LINK STEP
//*    - RENTDS WAS PREVIOUSLY COMPILED WITH 'RENT'
//*    - THE MEMBER @@DC370$ CAN BE USED TO BRING IN ALL C SOCKET
//*      MEMBERS. THIS IS EASIER THAN SPECIFYING ALL THE INDIVIDUAL
//*      MEMBER INCLUDES.
//*    - TCPV34.SEZARNT1 IS THE REENTRANT C SOCKET LIBRARY.
//*      IT IS USED ON THE PRE-LINK STEP.
//*    - TCPV34.SEZACMTX IS THE GENERIC SOCKET LIBRARY.
//*      IT IS USED ON THE LINK STEP TO RESOLVE OTHER C SOCKET
//*      MODULES THAT DO NOT EXIST IN TCPV34.SEZARNT1.
//*    - THE PRE-LINK REQUIRES THE 'UPCASE' PARM SO THAT THE
//*      OTHER MODULES FROM SEZACMTX (WHICH ARE KNOWN BY
//*      THEIR UPPERCASE NAMES) CAN BE FOUND.
//********************************************************************
//*
//*---------------------------------------------------------
//*     MODIFY THE FOLLOWING LINES TO CONFORM TO THE
//*     NAMING CONVENTIONS AT YOUR SITE.
//*---------------------------------------------------------
//RENTTEST PROC MYHLQ='USER71.TEST',
//   LIBPRFX1='CEEL.OSV2R7',
//   LIBPRFX2='CEE.OSV2R7'
//*-------------------------------------------------------------
//* PRE-LINKEDIT STEP:
//*-------------------------------------------------------------
//PLKED   EXEC PGM=EDCPRLK,
//    REGION=2048K,PARM='UPCASE'
//STEPLIB  DD  DSNAME=&LIBPRFX1..SCEERUN,DISP=SHR
//SYSMSGS  DD  DSNAME=&LIBPRFX2..SCEEMSGP(EDCPMSGE),DISP=SHR
//SYSLIB   DD  DSN=TCP.SEZARNT1,DISP=SHR
//SYSMOD   DD  DSNAME=&&PLKSET,UNIT=SYSDA,DISP=(MOD,PASS),
//             SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//OBJLIB   DD  DSN=&MYHLQ..OBJ,DISP=SHR
//MYRENT   DD  DSN=&MYHLQ..RENTDS,DISP=SHR
//SYSOUT   DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//*
//*-------------------------------------------------------------------
//* LINKEDIT STEP:
//*-------------------------------------------------------------------
//LKED   EXEC PGM=IEWL,COND=(4,LT,PLKED),
//    REGION=2048K,PARM='RENT,AMODE=31,MAP'
//SYSLIB   DD  DSNAME=&LIBPRFX2..SCEELKED,DISP=SHR
//         DD  DSN=TCP.SEZACMTX,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=*.PLKED.SYSMOD,DISP=(OLD,DELETE)
//SYSLMOD  DD  DSNAME=&MYHLQ..LMOD(CTESTRNT),DISP=SHR
//SYSUT1   DD  UNIT=SYSDA,SPACE=(32000,(30,30))
//   PEND
//S1  EXEC PROC=RENTTEST
//PLKED.SYSIN DD *
  INCLUDE MYRENT
  INCLUDE SYSLIB(@@DC370$)
/*
```

Figure 36. Sample JCL for Prelinking and Linking Reentrant Modules

Figure 37 shows sample JCL to be used when running the reentrant program prelinked and linked in the previous JCL sample. Modify the lines to conform to the naming conventions of your site.

```
//RUNTST  JOB ,RUN,MSGLEVEL=(1,1),CLASS=A,REGION=4096K
//*********************************************************************
//*                                                                   *
//*  SAMPLE JCL THAT RUNS A TEST PROGRAM, CTESTRNT                    *
//*                                                                   *
//*********************************************************************
//S1  EXEC PGM=CTESTRNT
//STEPLIB DD DSN=CEEL.OSV2R7.SCEERUN,DISP=SHR
//        DD DSN=USER71.TEST.LMOD,DISP=SHR
//SYSPRINT DD  SYSOUT=*
```

Figure 37. Sample JCL for Running the Reentrant Program

Compiler Messages

CS for OS/390 uses the C/C^{++}/390 compiler. For C programs, migrating from AD/Cycle® to the C/C^{++}/390 compiler can pose a few minor problems. Table 6 identifies some of the errors you might find.

Table 6. C/C^{++}/390 R4 Compiler Messages

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

Program Abends

A C program may compile and link correctly, but at run-time it might abend or behave peculiarly. The following lists some reasons for unexpected behavior, and suggests some fixes.

Errno values

Code depends on specific errno values. This may be a problem, as errno values can change from release to release. Do the following:

1. Check for any error conditions.
2. Make sure your logic has a default section that can be used if the specific errno has changed, or is no longer available.

Printing errno values: The tcpserror() function converts errno values to strings, which you can then print using printf() or a similar command. This procedure is provided by IBM and included with CS for OS/390, similar to the strerror() function in the standard C library.

Return values

Code depends on a specific return value. Some RTL functions, such as remove(), specify that the return code be nonzero on failure. In earlier releases, checking for negative one was sufficient; with this release, the correct check is for nonzero.

Unfortunately, there is no checklist of functions that might generate this problem. If you get an abend, work backwards from the failure and examine prior RTL function return-code checking.

Built-in RTL functions

If RTL functions were built-in during your compile, ensure that they perform the same way as the non-inbuilt functions from the RTL.

Functions that might have this problem include abs, cds, cs, decabs, decchk, decfix, fabs, fortrc, memchr, memcpy, memcmp, memset, strcat, strchr, strcmp, strcpy, strlen, strncat, strncmp, strncpy, strrchr, and tsched.

SCEERUN missing

Ensure that SCEERUN is the first library in STEPLIB encountered by your compile procedure.

Uninitialized storage

Check for uninitialized storage. Storage for automatic variables is guaranteed to be garbage.

C Socket Implementation

The IBM socket implementation differs from the Berkeley socket implementation. The following list summarizes the differences in the two methods:

- Under IBM implementation, you must make reference to the additional header file, TCPERRNO.H, if you want to refer to the networking errors other than those described in the compiler-supplied ERRNO.H file.
- Under IBM implementation, you must use the tcperror() routine to print the networking errno messages. tcperror() should be used only after socket calls; perror() should be used only after C library calls.
- Under IBM implementation, you must include MANIFEST.H to remap the socket function long names to eight-character names.
- The IBM ioctl() call implementation might differ from the current Berkeley ioctl() call implementation. See “ioctl()” on page 138 for a description of the functions supported by the IBM implementation.
- The IBM getsockopt() and setsockopt() calls support only a subset of the options available. See “getsockopt()” on page 123 and “setsockopt()” on page 174 for details about the supported options.

- The IBM fcntl() call supports only a subset of the options available. See “fcntl()” on page 99 for details about the supported commands.
- The IBM implementation supports an increased maximum number (2000) of simultaneous sockets through the use of the maxdesc() call. (Only 1997 simultaneous sockets can be used, however.) The default maximum number of sockets is 47, any or all of which can be AF_INET sockets.

Keep the following in mind while creating your C socket application:

- Compile all C source using the def(MVS) preprocessor symbol.
- During debugging, use sock_do_teststor (1) set to on to validate all storage addresses. After debugging, use sock_do_teststor (0) set to off.

C Socket Header Files

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

- bsdtime.h
- bsdtocms.h
- bsdtypes.h
- fcntl.h
- if.h
- in.h
- inet.h
- ioctl.h
- manifest.h
- netdb.h
- rtrouteh.h
- socket.h
- tcperrno.h
- types.h
- uio.h

Note: The C socket header files have been enhanced to allow the user to specify the coded character set to be used. When including the header files in an application, the bsdtypes.h file must preceed the socket.h file.

Manifest.h

Under IBM implementation, MANIFEST.H is used to remap socket function long names to eight-character names. To refer to the names, you must include the following statement as the first #include at the beginning of each program:

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

Prototyping

Under TCP/IP CS for OS/390, the prototyping of C socket functions and their parameters is the default. If you are migrating your applications, you can bypass the new prototyping by specifying `#define_TCP31_PROTOS` during a C compile.

C Structures

The parameter lists for some C language socket calls include a pointer to a data structure defined by a C structure. Table 7 shows the C structures used, and the corresponding assembler language syntax.

Table 7. C Structures in Assembler Language Format

C Structure	Assembler Language Equivalent

struct sockaddr_in
{
    short   sin_family;
    ushort sin_port;
    struct in_addr sin_addr;
    char   sin_zero[8];
};
	FAMILY DS H
	PORT DS H
	ADDR DS F
	ZERO DC XL8'00'

struct timeval
{
    long   tv_sec;
    long   tv_usec;
};
	TVSEC DS F
	TVUSEC DS F

struct linger {
    int   l_onoff;
    int   l_linger;
};
	ONOFF DS F
	LINGER DS F

struct   ifreq {
#define IFNAMSIZ 16
    char   ifr_name[IFNAMSIZ];
    union {
    struct sockaddr ifru_addr;
    struct sockaddr ifru_dstaddr;
    struct sockaddr ifru_broadaddr;
    short   ifru_flags;
    int   ifru_metric;
    caddr_t ifru_data;
    } ifr_ifru;
};
	NAME DS CL16
	ADDR.FAMILY DS H
	ADDR.PORT DS H
	ADDR.ADDR DS F
	ADDR.ZERO DC XL8'00'
	         ORG ADDR.FAMILY
	DST.FAMILY DS H
	DST.PORT DS H
	DST.ADDR DS F
	DST.ZERO DC XL8'00'
	         ORG ADDR.FAMILY
	BRD.FAMILY DS H
	BRD.PORT DS H
	BRD.ADDR DS F
	BRD.ZERO DC XL8'00'
	         ORG ADDR.FAMILY
	FLAGS DS H
	         ORG ADDR.FAMILY
	METRIC DS F

struct   ifconf {
    int ifc_len;
    union {
    caddr_t   ifcu_buf;
    struct   ifreq *ifcu_req;
    } ifc_ifcu;
};
	IFCLEN DS F
	IGNORED DS F

struct clientid {
    int   domain;
    char   name[8];
    char   subtaskname[8];
    char   reserved[20];
};
	DOMAIN DS F
	NAME DS CL8
	SUBTASK DS CL8
	RESERVED DC XL20'00'

Error Messages and Return Codes

For information about error messages, see *TCP/IP for MVS: Messages and Codes*.

The most common return codes (ERRNOs) returned by TCP/IP are listed following each socket call.

For information about all return codes see Appendix B. Return Codes.

C Socket Calls

This section lists the syntax, parameters, and other information appropriate to each C socket call supported by TCP/IP.

accept()

The accept() call is used by a server to accept a connection request from a client. The call accepts the first connection on its queue of pending connections. The accept() call creates a new socket descriptor with the same properties as *s* and returns it to the caller. If the queue has no pending connection requests, accept() blocks the caller unless *s* is in nonblocking mode. If no connection requests are queued and *s* is in nonblocking mode, accept() returns negative one and sets errno to EWOULDBLOCK. The new socket descriptor cannot be used to accept new connections. The original socket, *s*, remains available to accept additional connection requests.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>
int accept(int s, struct sockaddr *addr, int *addrlen)
```

Parameter	**Description**
s	The socket descriptor.
addr	The socket address of the connecting client that is filled by accept() before it returns. The format of *addr* is determined by the domain in which the client resides. *addr* is only filled in by accept() when both *addr* and *addrlen* are nonzero values.
addrlen	Must initially point to an integer that contains the size in bytes of the storage pointed to by *addr*. If *addr* is NULL, then *addrlen* is ignored and can be NULL.

The *s* parameter is a stream socket descriptor created using the socket() call. It is usually bound to an address using the bind() call. The listen() call marks the socket as one that accepts connections and allocates a queue to hold pending connection requests. The listen() call allows the caller to place an upper boundary on the size of the queue.

The *addr* parameter pointers to a buffer into which the connection requester address is placed. The *addr* parameter is optional and can be set to NULL. If *addr* or *addrlen* is null or zero, *addr* is not filled in. The exact format of *addr* depends on the addressing domain from which the communication request originated. For example, if the connection request originated in the AF_INET domain, *addr* points to a *sockaddr_in* structure as defined in the header file IN.H. The *addrlen* parameter is used only when *name* is not NULL. Before calling accept(), you must set the integer pointed to by *addrlen* to the size of the buffer, in bytes, pointed to by *addr*. If the buffer is not large enough to hold the address, only the *addrlen* number of bytes of the requester address are copied.

Note: This call is used only with SOCK_STREAM sockets. There is no way to screen requesters without calling accept(). The application cannot determine which system from which requesters it will accept connections. However, the caller can choose to close a connection immediately after discovering the identity of the requester.

A socket can be checked for incoming connection requests using the select() call.

Return Values

A nonnegative socket descriptor indicates success; the value negative one indicates an error. Errno identifies the specific error.

Errno **Description**

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

ENOBUFS Indicates insufficient buffer space available to create the new socket.

EINVAL The *s* parameter is not of type SOCK_STREAM.

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

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

Example

Following are two examples of the accept() call. In the first, the caller wishes to have the requester's address returned. In the second, the caller does not want the requester address returned.

```
int clientsocket;
int s;
struct sockaddr clientaddress;
int addrlen;
int accept(int s, struct sockaddr *addr, int *addrlen);
/* socket(), bind(), and listen() have been called */
/* EXAMPLE 1: I want the address now */
addrlen = sizeof(clientaddress);
clientsocket = accept(s, &clientaddress, &addrlen)
/* EXAMPLE 2: I can get the address later using getpeername() */
addrlen = 0;
clientsocket = accept(s, (struct sockaddr *) 0, (int *) 0);
```

Related Calls

bind(), connect(), getpeername(), listen(), socket()

bind()

The bind() call binds a unique local name to the socket using descriptor *s*. After calling socket(), the descriptor does not have a name associated with it. However, it does belong to a particular addressing family, as specified when socket() is called. The exact format of a name depends on the addressing family. The bind() call also allows servers to specify the network interfaces from which they want to receive UDP packets and TCP connection requests.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>
int bind(int s, struct sockaddr *name, int namelen)
```

Parameter	Description
s	Socket descriptor returned by a previous socket() call
name	Points to a *sockaddr* structure containing the name to be bound to *s*
namelen	Size of *name* in bytes

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

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

Related Information

Socket Descriptor Created in the AF_INET DOMAIN

If the socket descriptor *s* was created in the AF_INET domain, then the format of the name buffer is expected to be *sockaddr_in*, as defined in the header file IN.H.

```
struct in_addr
{
        u_long s_addr;
};
struct sockaddr_in
{
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

The *sin_family* field must be set to AF_INET.

The *sin_port* field identifies the port to which the application must bind. It must be specified in network byte order. If *sin_port* is set to zero, the caller expects the system to assign an available port. The application can call getsockname() to discover the port number assigned.

The *sin_addr.s_addr* field is set to the internet address and must be specified in network byte order. On hosts with more than one network interface (called multihomed hosts), a caller can select the interface to which it should bind. Subsequently, only UDP packets and TCP connection requests from this interface (the one value matching the bound name) are routed to the application. If this field is set to the constant INADDR_ANY, as defined in IN.H, the caller is requesting that the socket be bound to all network interfaces on the host. Subsequently, UDP packets and TCP connections from all interfaces matching the bound name are routed to the application. This becomes important when a server offers a service to multiple networks. By leaving the address unspecified, the server can accept all UDP packets and TCP connection requests made of its port, regardless of the network interface on which the requests arrived.

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

Socket Descriptor Created in the AF_IUCV DOMAIN

If the socket descriptor *s* is created in the AF_IUCV domain, the format of the name buffer is expected to be *sockaddr_iucv*, as defined in the header file SAIUCV.H.

```
struct sockaddr_iucv
{
    short           siucv_family;    /* addressing family */
    unsigned short  siucv_port;      /* port number */
    unsigned long   siucv_addr;      /* address */
    unsigned char   siucv_nodeid[8]; /* nodeid to connect to */
    unsigned char   siucv_userid[8]; /* userid to connect to */
    unsigned char   siucv_name[8];   /* iucvname for connect */
};
```

- The *siucv_family* field must be set to AF_IUCV.
- The *siucv_port*, *siucv_addr*, and *siucv_nodeid* fields are reserved for future use.
- The *siucv_port* and *siucv_addr* fields must be zeroed.
- The *siucv_nodeid* field must be set to exactly eight blank characters.
- The *siucv_userid* field is set to the MVS user ID of the application making the bind call. This field must be eight characters long, padded with blanks to the right. It cannot contain the NULL character.
- The *siucv_name* field is set to the application name by which the socket is to be known. It must be unique, because only one socket can be bound to a given name. The recommended form of the name contains eight characters, padded with blanks to the right. The eight characters for a connect() call executed by a client must exactly match the eight characters passed in the bind() call executed by the server.

Note: Internally, dynamic names are built using hexadecimal character strings representing the internal storage address of the socket. You should choose names that contain at least one non-hexadecimal character to prevent potential conflict. Hexadecimal characters include 0–9, and a–f. Uppercase A–F are considered non-hexadecimal and can be used by the user to build dynamic names.

Return Values

The value zero indicates success; the value negative one indicates an error. Errno identifies the specific error.

Errno Description

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

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

EFAULT

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

EAFNOSUPPORT

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

EADDRINUSE

The address is already in use. See the SO_REUSEADDR option described under “getsockopt()” on page 123 and the SO_REUSEADDR option described under the “setsockopt()” on page 174 for more information.

EINVAL

The socket is already bound to an address. For example, an attempt to bind a name to a socket that is in the connected state.

Example

Following are examples of the bind() call. The internet address and port must be in network byte order. To put the port into network byte order, the htons() utility routine is called to convert a short integer from host byte order to network byte order. The *address* field is set using another utility routine, inet_addr(), which takes a character string representing the dotted decimal address of an interface and returns the binary internet address representation in network byte order. Finally, it is a good idea to zero the structure before using it to ensure that the name requested does not set any reserved fields. See “connect()” on page 92 for examples how a client might connect to servers.

This example illustrates the bind() call binding to interfaces in the AF_INET domain.

```
int rc;
int s;
struct sockaddr_in myname;
struct sockaddr_iucv mymvsname;
int bind(int s, struct sockaddr *name, int namelen);
/* Bind to a specific interface in the internet domain */
/* make sure the sin_zero field is cleared */
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr = inet_addr("129.5.24.1"); /* specific interface */
myname.sin_port = htons(1024);
⋮
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
/* Bind to all network interfaces in the internet domain */
/* make sure the sin_zero field is cleared */
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr.s_addr = INADDR_ANY; /* specific interface */
myname.sin_port = htons(1024);
⋮
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
/* Bind to a specific interface in the internet domain.
   Let the system choose a port                       */
/* make sure the sin_zero field is cleared */
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr = inet_addr("129.5.24.1"); /* specific interface */
myname.sin_port = 0;
⋮
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
```

This example illustrates the bind() call binding to interfaces in the AF_IUCV domain.

```
/* Bind to a name in the IUCV domain */
/* make sure the siucv_addr, siucv_port fields are zeroed and the
   siucv_nodeid fields is set to blanks */
memset(&mymvsname, 0, sizeof(mymvsname));
strncpy(mymvsname.siucv_nodeid, "        ", 8);
strncpy(mymvsname.siucv_userid, "        ", 8);
strncpy(mymvsname.siucv_name,   "        ", 8);
mymvsname.siucv_family = AF_IUCV;
strncpy(mymvsname.siucv_userid, "MVSUSER1", 8);
strncpy(mymvsname.siucv_name, "APPL", 4);
⋮
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
```

The binding of a stream socket is not complete until a successful call to bind(), listen(), or connect() is made. Applications using stream sockets should check the return values of bind(), listen(), and connect() before using any function that requires a bound stream socket.

Related Calls

gethostbyname(), getsockname(), htons(), inet_addr(), listen(), socket()

close()

The close() call shuts down the socket associated with the socket descriptor *s*, and frees resources allocated to the socket. If *s* refers to an open TCP connection, the connection is closed. If a stream socket is closed when there is input data queued, the TCP connection is reset, not cleanly closed.

If you specify zero on SO_LINGER on the setsockopt() call, the data is cancelled and the CLOSE is immediately returned. If you do not specify a value for SO_LINGER on the setsockopt() call, the CLOSE returns and TCP/IP tries to immediately resend the data.

Note: Issue a shutdown() call before issuing a close() call for any socket.

```
#include <manifest.h>
#include <socket.h>
int close(int s)
```

Parameter	Description
s	Descriptor of the socket to be closed

Return Values

The value zero indicates success; the value negative one indicates an error. Errno identifies the specific error.

Errno	Description
EBADF	The *s* parameter is not a valid socket descriptor.

Related Calls

accept(), getsockopt(), setsockopt(), socket()

connect()

For stream sockets, the connect() call attempts to establish a connection between two sockets. For UDP sockets, the connect() call specifies the peer for a socket. The *s* parameter is the socket used to originate the connection request. The connect() call performs two tasks when called for a stream socket. First, it completes the binding necessary for a stream socket (in case it has not been previously bound using the bind() call). Second, it attempts to connect to another socket.

The connect() call on a stream socket is used by the client application to connect to a server. The server must have a passive open pending. If the server is using sockets, this means the server must successfully call bind() and listen() before a connection can be accepted by the server using accept(). Otherwise, connect() returns negative one and errno is set to ECONNREFUSED.

If *s* is in blocking mode, the connect() call blocks the caller until the connection is set up, or until an error is received. If the socket is in nonblocking mode, then connect() returns negative one with errno set to EINPROGRESS if the connection can be initiated (no other errors occurred). The caller can test completion of the connection setup by calling select() and testing ability to write to the socket.

When called for a datagram or raw socket, connect() specifies the peer with which this socket is associated. This gives the application the ability to use data transfer calls reserved for sockets that are in the connected state. In this case, read(), write(), readv(), writev(), send(), and recv() calls are available in addition to sendto(), recvfrom(), sendmsg(), and recvmsg() calls. Stream sockets can call connect() only once, but datagram sockets can call connect() multiple times to change their association. Datagram sockets can dissolve their association by connecting to an incorrect address, such as a null address (all fields zeroed).

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>
int connect(int s, struct sockaddr *name, int namelen)
```

Parameter	**Description**
s	Socket descriptor
name	Points to a *socket address* structure containing the address of the socket to which connection will be attempted
namelen	Size of the *socket address*, in bytes, pointed to by *name*

The *name* parameter points to a buffer containing the name of the peer to which the application needs to connect. The *namelen* parameter is the size, in bytes, of the buffer pointed to by *name*.

Related Information

Servers in the AF_INET Domain

If the server is in the AF_INET domain, the format of the name buffer is expected to be *sockaddr_in* as defined in the header file IN.H.

```
struct in_addr
{
        u_long s_addr;
};
struct sockaddr_in
{
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

The *sin_family* field must be set to AF_INET. The *sin_port* field identifies the port to which the server is bound; it must be specified in network byte order. The *sin_zero* field is not used, and must be set to all zeros.

Servers in the AF_IUCV Domain

If the server is in the AF_IUCV domain, the format of the name buffer is expected to be *sockaddr_iucv* as defined in the header file SAIUCV.H.

```
struct sockaddr_iucv
{
    short          siucv_family;    /* addressing family */
    unsigned short siucv_port;      /* port number */
    unsigned long  siucv_addr;      /* address */
    unsigned char  siucv_nodeid[8]; /* nodeid to connect to */
    unsigned char  siucv_userid[8]; /* userid to connect to */
    unsigned char  siucv_name[8];   /* iucvname for connect */
};
```

The *siucv_family* field must be set to AF_IUCV.

Note: The *siucv_port*, *siucv_addr*, and *siucv_nodeid* fields are reserved for future use.

The *siucv_port* and *siucv_addr* fields must be set to zero. Set the *siucv_nodeid* field to exactly eight blank characters. The *siucv_userid* field is set to the MVS user ID of the application to which the application is requesting a connection. This field must be eight characters long, padded with blanks to the right. It cannot contain the null character. The *siucv_name* field is set to the application name by which the server socket is known. The name should exactly match the eight characters passed in the bind() call executed by the server.

Return Values

The value zero indicates success; the value negative one indicates an error. Errno identifies the specific error.

Errno **Description**

EADDRNOTAVAIL
Calling host cannot reach the specified destination

EAFNOSUPPORT
Address family not supported

EALREADY
Socket descriptor *s* is marked nonblocking, and a previous connection attempt is incomplete

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

ECONNREFUSED
The connection request was rejected by the destination host.

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

EINPROGRESS
The socket descriptor *s* is marked nonblocking, and the connection cannot be completed immediately. The EINPROGRESS value does not indicate an error.

EISCONN Socket descriptor *s* is already connected

ENETUNREACH
Network cannot be reached from this host

ETIMEDOUT Connection attempt timed out before the connection was made.

Example

Following is a connect() call example. The internet address and port must be in network byte order. To put the port into network byte order, the htons() utility is called to convert a short integer from host byte order to network byte order. The *address* field is set using another utility, inet_addr(), which takes a character string representing the dotted decimal address of an interface, and returns the binary internet address in network byte order. Zero the structure before using it to ensure that the name requested does not set any reserved fields.

These examples could be used to connect to the servers shown in the examples listed with the call, “bind()” on page 87.

```
int s;
struct sockaddr_in servername;
struct sockaddr_iucv mvsservername;
int rc;
int connect(int s, struct sockaddr *name, int namelen);
/* Connect to server bound to a specific interface in the internet domain */
/* make sure the sin_zero field is cleared */
memset(&servername, 0, sizeof(servername));
servername.sin_family = AF_INET;
servername.sin_addr = inet_addr("129.5.24.1"); /* specific interface */
servername.sin_port = htons(1024);
⋮
rc = connect(s, (struct sockaddr *) &servername, sizeof(servername));
/* Connect to a server bound to a name in the IUCV domain */
/* make sure the siucv_addr, siucv_port, siucv_nodeid fields are cleared
*/
memset(&mvsservername, 0, sizeof(mvsservername));
mvsservername.siucv_family = AF_IUCV;
strncpy(mvsservername.siucv_nodeid, "        ",8);
/* The field is 8 positions padded to the right with blanks */
strncpy(mvsservername.siucv_userid, "MVSUSER1 ", 8);
strncpy(mvsservername.siucv_name, "APPL    ", 8);
⋮
rc = connect(s, (struct sockaddr *) &mvsservername, sizeof(mvsservername));
```

Related Calls

bind(), htons(), inet_addr(), listen(), select(), selectex(), socket()

endhostent()

The endhostent() call closes the *hlq*.HOSTS.SITEINFO and *hlq*.HOSTS.ADDRINFO data sets. *hlq*.HOSTS.SITEINFO and *hlq*.HOSTS.ADDRINFO data sets contain information about known hosts. The endhostent() call is available only where RESOLVE_VIA_LOOKUP is defined before MANIFEST.H is included.

```
#include <manifest.h>
#include <socket.h>
void endhostent()
```

Related Calls

gethostbyname(), gethostent(), sethostent()

endnetent()

The endnetent() call closes the *hlq*.HOSTS.SITEINFO data set. The *hlq*.HOSTS.SITEINFO data set contains information about known networks.

```
#include <manifest.h>
#include <socket.h>
void endnetent()
```

Related Calls

getnetbyname(), getnetent(), setnetent()

endprotoent()

The endprotoent() call closes the *hlq*.ETC.PROTO data set.

The *hlq*.ETC.PROTO data set contains information about networking protocols IP, ICMP, TCP, and UDP.

```
#include <manifest.h>
#include <socket.h>
 void endprotoent()
```

Related Calls

getprotoent(), setprotoent()

endservent()

The endservent() call closes the *hlq*.ETC.SERVICES data set.

The *hlq*.ETC.SERVICES data set contains information about the networking services running on the host. Example services are domain name server, FTP, and Telnet.

```
#include <manifest.h>
#include <socket.h>
void endservent()
```

Related Calls

getservbyport(), getservent(), setservent()

fcntl()

The operating characteristics of sockets can be controlled with the fcntl() call.

Note: COMMAND values that are supported by the UNIX Systems Services fcntl() callable service are also supported.

```
#include <manifest.h>
#include <socket.h>
#include <bsdtypes.h>
#include <fcntl.h>
int fcntl(int s, int cmd, int arg)
```

Parameter	**Description**
s	Socket descriptor
cmd	Command to perform
arg	Data associated with *cmd*

The operations to be controlled are determined by *cmd*. The *arg* parameter is a variable the meaning of which depends on the value of the *cmd* parameter. The following are valid fcntl() commands:

Command **Description**

F_SETFL Sets the status flags of socket descriptor *s*. One flag, FNDELAY, can be set.

F_GETFL Returns the status flags of socket descriptor *s*. One flag, FNDELAY, can be queried.

The FNDELAY flag marks *s* as being in nonblocking mode. If data is not present on calls that can block (read(), readv(), and recv()) the call returns with negative one, and errno is set to EWOULDBLOCK.

Note: This function does not reject other values that might be rejected downstream.

Return Values

For the **F_GETFL** command, the return value is the flags, set as a bit mask. For the **F_SETFL** command, the value zero indicates success; the value negative one indicates an error. Errno identifies the specific error.

Errno	**Description**
EBADF	The *s* parameter is not a valid socket descriptor.
EINVAL	The *arg* parameter is not a valid flag, or the command is not a valid command.

Example

```
int s;
int rc;
int flags;
⋮
/* Place the socket into nonblocking mode */
rc = fcntl(s, F_SETFL, FNDELAY);
/* See if asynchronous notification is set */
flags = fcntl(s, F_GETFL, 0);
if (flags & FNDELAY)
   /* it is set */
else
   /* it is not */
```

Related Calls

ioctl(), getsockopt(), setsockopt(), socket()

getclientid()

The getclientid() call returns the identifier by which the calling application is known to the TCP/IP address space. The *clientid* is used in givesocket() and takesocket() calls.

```
#include <manifest.h>
#include <socket.h>
#include <bsdtypes.h>
int  getclientid(int domain, struct clientid *clientid)
```

Parameter	Description
domain	The value in *domain* must be AF_INET.
clientid	Points to a *clientid* structure to be provided.

Return Values

The value zero indicates success. The value negative one indicates an error. Errno identifies the specific error.

Errno **Description**

EFAULT
The *clientid* parameter as specified would result in an attempt to access storage outside the caller address space, or storage that cannot be modified by the caller.

EAFNOSUPPORT
The domain is not AF_INET.

Related Calls

takesocket()

getdtablesize()

The TCPIP address space reserves a fixed-size table for each address space using sockets. The size of this table equals the number of sockets an address space can allocate simultaneously. The getdtablesize() call returns the maximum number of sockets that can fit in the table.

To increase the table size, use maxdesc(). After calling maxdesc(), always use getdtablesize() to verify the change.

```
#include <manifest.h>
#include <socket.h>
int getdtablesize()
```

Related Calls

maxdesc()

gethostbyaddr()

The gethostbyaddr() call tries to resolve the host address through a name server, if one is present. If a name server is not present, gethostbyaddr() searches the *hlq*.HOSTS.ADDRINFO data set until a matching host address is found, or an EOF marker is reached. If the symbol RESOLVE_VIA_LOOKUP is defined before including MANIFEST.H, gethostbyaddr() uses the *hlq*.HOSTS.ADDRINFO file for resolution, and the named server is not used.

Note: Data sets *hlq*.HOSTS.LOCAL, *hlq*.HOSTS.ADDRINFO, and *hlq*.HOSTS.SITEINFO are described in *OS/390 IBM Communications Server: IP Configuration Reference*.

```
#include <manifest.h>
#include <netdb.h>
struct hostent *gethostbyaddr(char *addr, int addrlen, int domain)
```

Parameter	**Description**
addr	Points to an unsigned long value containing the address of the host.
addrlen	Size of *addr* in bytes.
domain	Address domain supported (AF_INET).

The gethostbyaddr() call points to *hostent* structure for the host address specified on the call.

The NETDB.H header file defines the *hostent* structure, and contains the following elements:

Element	**Description**
h_name	The address of the official name of the host.
h_aliases	A pointer to a zero-terminated list of addresses pointing to alternate names for the host.
h_addrtype	The type of host address returned; currently, always set to AF_INET.
h_length	The length of the host address, in bytes.
h_addr	A pointer to a zero-terminated list of addresses pointing to the internet host addresses returned by the call.

Return Values

The return value points to static data that is overwritten by subsequent calls. A pointer to a *hostent* structure indicates success. A NULL pointer indicates an error or end-of-file. When a NULL pointer or zero is returned, the value of errno is indeterminate: the output from a tcperror() call cannot be validated. The global variable h_errno identifies the specific error.

h_errno	**Description**
HOST_NOT_FOUND	The name specified is unknown, the address domain specified is not supported, or the address length specified is not valid.
TRY_AGAIN	Temporary error; information not currently accessible.
NO_RECOVERY	Fatal error occurred.

Related Calls

gethostent(), sethostent(), endhostent()

gethostbyname()

The gethostbyname() call tries to resolve the host address through a name server, if one is present. If a named server is not present, gethostbyname() searches the *hlq*.HOSTS.SITEINFO data set until a matching host name is found, or an EOF marker is reached.

```
#include <manifest.h>
#include <netdb.h>
struct hostent *gethostbyname(char *name)
```

Parameter	**Description**
name	The name of the host being queried.

The gethostbyname() call returns a pointer to a *hostent* structure for the host name specified on the call. If the symbol RESOLVE_VIA_LOOKUP is defined before including MANIFEST.H, gethostbyname() uses the *hlq*.HOSTS.ADDRINFO file for resolution, and the name server is not used.

Note: Data sets *hlq*.HOSTS.LOCAL, *hlq*.HOSTS.ADDRINFO, and *hlq*.HOSTS.SITEINFO are described in *OS/390 IBM Communications Server: IP Configuration Reference*.

The NETDB.H header file defines the *hostent* structure and contains the following elements:

Element	**Description**
h_name	The address of the official name of the host.
h_aliases	A pointer to a zero-terminated list of addresses pointing to alternate names for the host.
h_addrtype	The type of host address returned; currently, set to AF_INET.
h_length	The length of the host address in bytes.
h_addr	A pointer to the network address of the host.

Return Values

The return value points to static data that is overwritten by subsequent calls. A pointer to a *hostent* structure indicates success. A NULL pointer indicates an error or EOF. When a NULL pointer or zero is returned, the value of errno is indeterminate, and therefore, the output from a tcperror() call is also not valid. Global variable h_errno identifies the specific error.

h_errno Value	**Description**
HOST_NOT_FOUND	The name specified is unknown.
TRY_AGAIN	Temporary error; information not currently accessible.
NO_RECOVERY	Fatal error occurred.

Related Calls

gethostent(), sethostent(), endhostent()

gethostent()

The gethostent() call reads the next line of the *hlq*.HOSTS.SITEINFO data set.

The gethostent() call points to the next entry in the *hlq*.HOSTS.SITEINFO data set. The gethostent() call uses *hlq*.HOSTS.ADDRINFO to get aliases. The gethostent() call is available only when RESOLVE_VIA_LOOKUP is defined before MANIFEST.H is included.

Note: Data sets *hlq*.HOSTS.LOCAL, *hlq*.HOSTS.ADDRINFO, and *hlq*.HOSTS.SITEINFO data sets are described in *OS/390 IBM Communications Server: IP Configuration Reference*.

```
#include <manifest.h>
#include <netdb.h>
struct hostent *gethostent()
```

The NETDB.H header file defines the *hostent* structure, and contains the following elements:

Element	**Description**
h_name	The address of the official name of the host.
h_aliases	A pointer to a zero-terminated list of addresses pointing to alternate names for the host.
h_addrtype	The type of host address returned; currently set to AF_INET.
h_length	The length of the host address, in bytes.
h_addr	A pointer to the network address of the host.

Return Values

The return value points to static data that is overwritten by subsequent calls. A pointer to a *hostent* structure indicates success. A NULL pointer indicates an error or EOF. When a NULL pointer or zero is returned, the value of errno is indeterminate, and therefore, the output from a tcperror() call is also not valid.

Related Calls

gethostbyname(), sethostent()

gethostid()

The gethostid() call returns the 32-bit identifier unique to the current host. This value is the default home internet address.

This call can be used only in the AF_INET domain.

```
#include <manifest.h>
#include <socket.h>
unsigned long gethostid()
```

Return Values

The gethostid() call returns the 32-bit identifier of the current host, which should be unique across all hosts. When a NULL pointer or zero is returned, the value of errno is indeterminate, and therefore, the output from a tcperror() call is also not valid.

Related Calls

gethostname()

gethostname()

The gethostname() call returns the name of the host processor on which the program is running. Characters to the limit of *namelen* are copied into the name array. The value returned for hostname is limited to 24 characters. The returned name is NULL-terminated unless truncated to the size of the name array.

This call can be used only in the AF_INET domain.

Errno EINVAL is returned when *namelen* is zero, or greater than 255 characters.

```
#include <manifest.h>
#include <socket.h>
int gethostname(char *name, int namelen)
```

Parameter	**Description**
name	Character array to be filled with the host name
namelen	Length of *name*; restricted to 255 characters

Return Values

The value zero indicates success; the value negative one indicates an error. Errno identifies the specific error.

Errno	**Description**
EFAULT	The *name* parameter specified an address outside the caller address space.

Related Calls

gethostbyname(), gethostid()

getibmopt()

The getibmopt() call returns the number of TCP/IP images installed on a given MVS system, and their status, version, and name.

Note: Images from pre-V3R2 releases of TCP/IP for MVS are excluded. The getibmopt() call is not meaningful to pre-V3R2 releases.

Using this information, the caller can dynamically choose the TCP/IP image with which to connect through the setibmopt() call. The getibmopt() call is optional. If it is not used, determine the connecting TCP/IP image as follows:

- Connect to the TCP/IP specified TCPIPJOBNAME in the active TCPIP.DATA file.
- Locate TCPIP.DATA using one of the following:
  SYSTCPD DD card
  jobname/userid.TCPIP.DATA
  zapname.TCPIP.DATA

For detailed information about this method, see *TCP/IP for MVS: Planning and Migration Guide*.

```
#include <manifest.h>
#include <socket.h>
int getibmopt(int cmd, struct ibm_gettcpinfo *buf)
struct ibm_tcpimage {
    unsigned short status;
    unsigned short version;
    char name[8];
}
struct ibm_gettcpinfo {
    int tcpcnt;
    struct ibm_tcpimage image[8];
}
```

Parameter	Description
cmd	The command to perform. For TCP/IP V3R2 for MVS, IBMTCP_IMAGE is supported.
buf	Points to the structure filled in by the call.

The *buf* parameter is a pointer to the (*struct ibm_gettcpinfo*) buffer into which the TCP/IP image status, version, and name are placed.

On successful return, the *struct ibm_tcpimage* buffer contains the status, version, and name of up to eight active TCP/IP images.

The status field can contain the following information:

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

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

When this field returns with a combination of Down and Stopped, TCP/IP was abended. Value stopped, when returned alone, indicates that TCP/IP has been stopped only.

The version field for OS/390 Release 8 is X'0308'.

The TCP/IP character name field is the PROC name, left-justified, and padded with blanks.

The *tcpcnt* field of *struct ibm_gettcpinfo* is a count field into which the TCP/IP image count is placed. The caller uses this value to determine how many entries in the *ibm_tcpimage* structure of *buf* have been filled. If the *tcpcnt* returned is zero, there are no TCP/IP images present.

Return Values

Zero indicates success; the value negative one indicates an error. Errno identifies the specific error.

Errno	**Description**
EOPNOTSUPP	This is returned for command that is not valid.
EFAULT	The *name* parameter specified an address outside of the caller address space.

getibmsockopt()

Like getsockopt() call, the getibmsockopt() call gets the options associated with a socket in the AF_INET domain. This call is for options specific to the IBM implementation of sockets. Currently, only the SOL_SOCKET level and socket option SO_SO_NONBLOCKLOCAL are supported.

This call can be used only in the AF_INET domain.

```
#include <manifest.h>
#include <socket.h>
int getibmsockopt(int s, int level, int optname, char *optval, int *optlen)
```

Parameter	Description
s	The socket descriptor.
level	The level for which the option is set.
optname	The name of a specified socket option.
optval	Points to option data.
optlen	Points to the length of the option data.

For SO_NONBLOCKLOCAL, optval points to an integer; the call returns zero when the option has not been set, and returns one when the option has been set.

Return Values

The value zero indicates success; the value negative one indicates an error. Errno identifies the specific error.

Errno	Description
EBADF	The *s* parameter is not a valid socket descriptor.
EFAULT	Using optval and optlen parameters would result in an attempt to access storage outside the caller address space.
EINVAL	This is returned when optlen points to a length of zero.

Example

```
#include <manifest.h>
#include <socket.h>
#include <tcperror.h>
 { struct ibm_bulkmode_struct bulkstr;
   int optlen, rc;
   optlen = sizeof(bulkstr);
   rc = getibmsockopt(s, SOL_SOCKET, SO_BULKMODE, (char *), &bulkstr, &optlen);
   if (rc < 0)
     { tcperror("on getibmsockopt()");
       exit(1);
     }
   fprintf(stream,"%d byte buffer available for outbound queue.\n",
           bulkstr.b_max_send_queue_size_avail);
 }
```

Related Calls

ibmsflush(), setibmsockopt(), getockopt()

getnetbyaddr()

The getnetbyaddr() call searches the *hlq*.HOSTS.ADDRINFO data set for the specified network address.

This call can be used only in the AF_INET domain.

Note: Data sets *hlq*.HOSTS.LOCAL, *hlq*.HOSTS.ADDRINFO, and *hlq*.HOSTS.SITEINFO are described in *OS/390 IBM Communications Server: IP Configuration Reference*.

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

Parameter	**Description**
net	The network address.
type	The address domain supported (AF_INET).

The *netent* structure is defined in the NETDB.H header file and contains the following elements:

Element	**Description**
n_name	The official name of the network.
n_aliases	An array, terminated with a NULL pointer, of alternative names for the network.
n_addrtype	The type of network address returned. The call always sets this value to AF_INET.
n_net	The network number, returned in host byte order.

Return Values

The return value points to static data that is overwritten by subsequent calls. A pointer to a *netent* structure indicates success. A NULL pointer indicates an error or end-of-file. When a NULL pointer or zero is returned, the value of errno is indeterminate, and therefore, the output from a tcperror() call cannot be validated.

Related Calls

endnetent(), getnetbyname(), getnetent(), setnetent(), endhostent()

getnetbyname()

The getnetbyname() call searches the *hlq*.HOSTS.ADDRINFO data set for the specified network name.

This call can be used only in the AF_INET domain.

Note: Data sets*hlq*.HOSTS.LOCAL, *hlq*.HOSTS.ADDRINFO, and *hlq*.HOSTS.SITEINFO are described in *OS/390 IBM Communications Server: IP Configuration Reference*.

```
#include <manifest.h>
#include <netdb.h>
struct netent *getnetbyname(char *name)
```

Parameter	**Description**
name	Points to a network name

The getnetbyname() call returns a pointer to a *netent* structure for the network name specified on the call.

The *netent* structure is defined in the NETDB.H header file and contains the following elements:

Element	**Description**
n_name	The official name of the network.
n_aliases	An array, terminated with a NULL pointer, of alternative names for the network.
n_addrtype	The type of network address returned. The call always sets this value to AF_INET.
n_net	The network number, returned in host byte order.

Return Values

The return value points to static data that is overwritten by subsequent calls. A pointer to a *netent* structure indicates success. A NULL pointer indicates an error or end-of-file. When a NULL pointer or zero is returned, the value of errno is indeterminate, and therefore, the output from a tcperror() call cannot be validated.

Related Calls

endnetent(), getnetbyaddr(), getnetent(), setnetent(), endhostent()

getnetent()

The getnetent() call reads the next NETsite entry of the *hlq*.HOSTS.SITEINFO data set.

Note: Data sets *hlq*.HOSTS.LOCAL, *hlq*.HOSTS.ADDRINFO, and *hlq*.HOSTS.SITEINFO are described in *OS/390 IBM Communications Server: IP Configuration Reference*.

```
#include <manifest.h>
#include <netdb.h>
struct netent *getnetent()
```

The getnetent() call points to the next NETsite entry in the *hlq*.HOSTS.SITEINFO data set.

The *netent* structure is defined in the NETDB.H header file and contains the following elements:

Element	**Description**
n_name	The official name of the network.
n_aliases	An array, terminated with a NULL pointer, of alternative names for the network.
n_addrtype	The type of network address returned. The call always sets this value to AF_INET.
n_net	The network number, returned in host byte order.

Return Values

The return value points to static data that is overwritten by subsequent calls. A pointer to a *netent* structure indicates success. A NULL pointer indicates an error or end-of-file.

Related Calls

endnetent(), getnetbyaddr(), getnetbyname(), setnetent(), endhostent()

getpeername()

The getpeername() call returns the name of the peer connected to socket descriptor *s*. For AF_IUCV, *namelen* must be initialized to reflect the size of the space pointed to by *name*; it is set to the number of bytes copied into the space before the call returns. For AF_INET, the input value in the contents of *namelen* are ignored, but are set before the call returns. The size of the peer name is returned in bytes. If the buffer of the local host is too small to receive the entire peer name, the name is truncated.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
int getpeername(int s, struct sockaddr *name, int *namelen)
```

Parameter	Description
s	The socket descriptor.
name	Points to a structure containing the internet address of the connected socket that is filled in by getpeername() before it returns. The exact format of *name* is determined by the domain in which communication occurs. The name length is restricted to 24 characters.
namelen	Points to a fullword containing the size of the address structure pointed to by *name* in bytes.

Return Values

The value zero indicates success; the value negative one indicates an error. Errno identifies the specific error.

Errno	Description
EBADF	The *s* parameter is not a valid socket descriptor.
EFAULT	Using the *name* and *namelen* parameters as specified would result in an attempt to access storage outside of the caller address space.
ENOTCONN	The socket is not in the connected state.

Related Calls

accept(), connect(), getsockname(), socket()

getprotobyname()

The getprotobyname() call searches the *hlq*.ETC.PROTO data set for the specified protocol name.

The getprotobyname() call returns a pointer to a *protoent* structure for the network protocol specified on the call.

```
#include <manifest.h>
#include <netdb.h>
struct protoent *getprotobyname(char *name)
```

Parameter	Description
name	Points to the specified protocol.

The *protoent* structure is defined in the NETDB.H header file and contains the following elements:

Element	Description
p_name	The official name of the protocol.
p_aliases	An array, terminated with a NULL pointer, of alternative names for the protocol.
p_proto	The protocol number.

Return Values

The return value points to static data that is overwritten by subsequent calls. A pointer to a *protoent* structure indicates success. A NULL pointer indicates an error or end-of-file. When a NULL pointer or zero is returned, the value of errno is indeterminate, and therefore, the output from a tcperror() call cannot be validated.

Related Calls

endprotoent(), getprotobynumber(), getprotoent(), setprotoent()

getprotobynumber()

The getprotobynumber() call searches the *hlq*.ETC.PROTO data set for the specified protocol number.

The getprotobynumber() call returns a pointer to a *protoent* structure for the network protocol specified on the call.

```
#include <manifest.h>
#include <netdb.h>
struct protoent *getprotobynumber(int proto)
```

Parameter	Description
proto	Protocol number

The *protoent* structure is defined in the NETDB.H header file and contains the following elements:

Element	Description
p_name	The official name of the protocol.
p_aliases	An array, terminated with a NULL pointer, of alternative names for the protocol.
p_proto	The protocol number.

Return Values

The return value points to static data that is overwritten by subsequent calls. A pointer to a *protoent* structure indicates success. A NULL pointer indicates an error or end-of-file. When a NULL pointer or zero is returned, the value of errno is indeterminate, and therefore, the output from a tcperror() call cannot be validated.

Related Calls

endprotoent(), getprotobyname(), getprotoent(), setprotoent()

getprotoent()

The getprotoent() call reads the *hlq*.ETC.PROTO data set, and the getprotoent() call returns a pointer to the next entry in the *hlq*.ETC.PROTO data set.

```
#include <manifest.h>
#include <netdb.h>
struct protoent *getprotoent()
```

The *protoent* structure is defined in the NETDB.H header file and contains the following elements:

Element	Description
p_name	The official name of the protocol.
p_aliases	An array, terminated with a NULL pointer, of alternative names for the protocol.
p_proto	The protocol number.

Return Values

The return value points to static data that is overwritten by subsequent calls. A pointer to a *protoent* structure indicates success. A NULL pointer indicates an error or end-of-file. When a NULL pointer or zero is returned, the value of errno is indeterminate, and therefore, the output from a tcperror() call cannot be validated.

Related Calls

endprotoent(), getprotobyname(), getprotobynumber(), setprotoent()

getservbyname()

The getservbyname() call searches the *hlq*.ETC.SERVICES data set for the specified service name. Service name searches must match the protocol, if a protocol is stated.

The getservbyname() call returns a pointer to a *servent* structure for the network service specified on the call.

```
#include <manifest.h>
#include <netdb.h>
struct servent *getservbyname(char *name, char *proto)
```

Parameter	Description
name	Points to the specified service name
proto	Points to the specified protocol

The *servent* structure is defined in the NETDB.H header file and contains the following elements:

Element	Description
s_name	The official name of the service.
s_aliases	An array, terminated with a NULL pointer, of alternative names for the service.
s_port	The port number of the service.
s_proto	The protocol required to contact the service.

Return Values

The return value points to static data that is overwritten by subsequent calls. Points to a *servent* structure indicates success. A NULL pointer indicates an error or end-of-file. When a NULL pointer or zero is returned, the value of errno is indeterminate, and therefore, the output from a tcperror() call cannot be validated.

Related Calls

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

getservbyport()

The getservbyport() call searches the *hlq*.ETC.SERVICES data set for the specified port number. Searches for a port number must match the protocol, if a protocol is stated.

The getservbyport() call returns a pointer to a *servent* structure for the port number specified on the call.

```
#include <manifest.h>
#include <netdb.h>
struct servent *getservbyport(int port, char *proto)
```

Parameter	**Description**
port	Port number
proto	Points to the specified protocol

The *servent* structure is defined in the NETDB.H header file and contains the following elements:

Element	**Description**
s_name	The official name of the service.
s_aliases	An array, terminated with a NULL pointer, of alternative names for the service.
s_port	The port number of the service.
s_proto	The protocol required to contact the service.

Return Values

The return value points to static data that is overwritten by subsequent calls. Points to a *servent* structure indicates success. A NULL pointer indicates an error or end-of-file. When a NULL pointer or zero is returned, the value of errno is indeterminate, and therefore, the output from a tcperror() call cannot be validated.

Related Calls

endservant(), getservbyname(), getservent(), setservent()

getservent()

The getservent() call reads the next line of the *hlq*.ETC.SERVICES data set, and returns a pointer to the next entry in the *hlq*.ETC.SERVICES data set.

```
#include <manifest.h>
#include <netdb.h>
struct servent *getservent()
```

The *servent* structure is defined in the NETDB.H header file and contains the following elements:

Element	**Description**
s_name	The official name of the service.
s_aliases	An array, terminated with a NULL pointer, of alternative names for the service.
s_port	The port number of the service.
s_proto	The required protocol to contact the service.

Return Values

The return value points to static data that is overwritten by subsequent calls. Points to a *servent* structure indicates success. A NULL pointer indicates an error or end-of-file. When a NULL pointer or zero is returned, the value of errno is indeterminate, and therefore, the output from a tcperror() call cannot be validated.

Related Calls

endservant(), getservbyname(), getservbyport(), setservent()

getsockname()

The getsockname() call stores the current name for the socket specified by the *s* parameter into the structure pointed to by the *name* parameter. It returns the address to the socket that has been bound. If the socket is not bound to an address, the call returns with the family set, and sets the rest of the structure to zero. For example, an inbound socket in the internet domain would cause the name to point to a *sockaddr_in* structure with the *sin_family* field set to AF_INET, and all other fields zeroed.

Stream sockets are not assigned a name, until a call is successful— bind(), connect(), or accept().

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

For AF_IUCV, *namelen* must be initialized to indicate the size of the space pointed to by *name*, and is set to the number of bytes copied into the space before the call returns. For AF_INET, the input value in the contents of *namelen* is ignored, but set before the call returns.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
int getsockname(int s, struct sockaddr *name, int *namelen)
```

Parameter	**Description**
s	The socket descriptor.
name	The address of the buffer into which getsockname() copies the name of *s*.
namelen	Points to a fullword containing the size of the address structure pointed to by *name* in bytes.

Return Values

A value of zero indicates success; the value negative one indicates an error. Errno identifies the specific error.

Errno Value	**Description**
EBADF	The *s* parameter is not a valid socket descriptor.
EFAULT	Using the *name* and *namelen* parameters as specified would result in an attempt to access storage outside of the caller address space.

Related Calls

accept(), bind(), connect(), getpeername(), socket()

getsockopt()

The getsockopt() call gets options associated with a socket. It can be called only for sockets in the AF_INET domain. This call is not supported in the AF_IUCV domain. While options can exist at multiple protocol levels, they are always present at the highest socket level.

When manipulating socket options, you must specify the level at which the option resides, and the name of that option. To manipulate options at the socket level, the *level* parameter must be set to SOL_SOCKET as defined in SOCKET.H. To manipulate options at any other level, such as the TCP or IP level, supply the appropriate protocol number for the protocol controlling the option. Currently, only the SOL_SOCKET level is supported. The getprotobyname() call can be used to return the protocol number for a named protocol.

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

int getsockopt(int s, int level, int optname, char *optval, int *optlen
```

Parameter	Description
s	The socket descriptor.
level	The level to which the option is set; only SOL_SOCKET is supported
optname	The name of a specified socket option
optval	Points to option data
optlen	Points to the length of the option data

The *optval* and *optlen* parameters are used to return data used by the particular get command. The *optval* parameter points to a buffer that is to receive the data requested by the get command. The *optlen* parameter points to the size of the buffer pointed to by the *optval* parameter. Initially, this size must be set to the size of that buffer before calling getsockopt(). On return it is set to the size of the data actually returned.

All of the socket level options except SO_LINGER expect *optval* to point to an integer and *optlen* to be set to the size of an integer. When the integer is nonzero, the option is enabled. When it is zero, the option is disabled. The SO_LINGER option expects *optval* to point to a *linger* structure as defined in SOCKET.H. This structure is defined in the following example:

```
#include <manifest.h>
struct  linger
{
       int     l_onoff;               /* option on/off */
       int     l_linger;              /* linger time */
};
```

The *l_onoff* field is set to zero if the SO_LINGER option is being disabled. A nonzero value enables the option. The *l_linger* field specifies the amount of time to linger on close.

The following options are recognized at the socket level:

Option	Description

SO_ACCEPTCONN

Indicates whether listen() was called for a stream socket.

SO_BROADCAST

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

SO_CLUSTERCONNTYPE

Returns a bit mapped 32–bit value. One or more than one of the following will be returned when a socket is connected:

- None means that the socket is active, but the partner is not in the same cluster. If this indicator is set, the other three indicators are zero.
- Same cluster means that the connection, and the partner is in the same cluster.
- Same image means that the connection is active, and the partner is in the same MVS image. SO_CLUSTERCONNTYPE_SAME_CLUSTER will also be set.
- Internal means that only this end of the connection flows over a single link (host route) to the stack hosting the partner application. To determine if both ends of the connection flow over a single link, the partner application must also issue this getsockopt() and both ends exchange their results from this socket call (through an application-dependent method).

  Note: An internal indicator means that for this side of the connection, the link type is one of the following:
  - – MPCPTP (including XCF and IUTSAMEH connections)
  - – CTC

On return, one or more of the following bits are set:

```
00000000  00000000 00000000 00000001'-SO_CLUSTERCONNTYPE_NONE
00000000  00000000 00000000 00000010'-SO_CLUSTERCONNTYPE_SAME_CLUSTER
00000000  00000000 00000000 00000100'-SO_CLUSTERCONNTYPE_SAME_IMAGE
00000000  00000000 00000000 00001000'-SO_CLUSTERCONNTYPE_SAME_INTERNAL
00000000  00000000 00000000 00000000'-SO_CLUSTERCONNTYPE_SAME_NOCONN
```

Note: A value of all zeros means that there is no active connection on the socket. This is usually the case for a listening socket. This is also true for a client socket before the client issues connect(). The caller should first check for a returned value of SO_CLUSTERCONNTYPE_NOCONN before checking for any of the other returned indicators.

If getsockopt() (SO_CLUSTERCONNTYPE) is issued before the connection has been established results in a return value of zero.

If the application issues getsockopt() (SO_CLUSTERCONNTYPE) on a connected socket, and has received an indication of SO_CLUSTERCONNTYPE_INTERNAL, any subsequent rerouting decisions due to current route failure will either select an alternate route, which is also SO_CLUSTERCONNTYPE_INTERNAL, or fail the connection with no route available indications. This means that when an application has received an indication of SO_CLUSTERCONNTYPE_INTERNAL on a connection, any subsequent rerouting preserves that indication on the new route, or will fail the connection. This ensures that a connection that an application relies on as being internal does not transparently become non-internal due to a routing change.

If the application never issues the new getsockopt() or if the connection was previously reported as not SO_CLUSTERCONNTYPE_INTERNAL, rerouting decisions are made as at present, and the rerouting is transparent to the application a long as any alternate route exists.

SO_DEBUG The sock_debug() call provides the socket library tracing facility. The *onoff* parameter can have a value of zero or nonzero. When *onoff*=0, (the default) no socket library tracing is done; when *onoff*=nonzero, the system traces for socket library calls and interrupts.

SO_ERROR Returns any error pending on the socket and clears the error status. It can be used to check for asynchronous errors on connected datagram sockets and for other asynchronous errors (errors returned explicitly by one of the socket calls).

SO_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 along an otherwise idle connection. If the remote TCP does not respond to the packet or to retransmissions of the packet, the connection is terminated with the error ETIMEDOUT.

SO_LINGER Lingers on close if data is present. The default is *disabled*. When this option is enabled and there is unsent data present when close() is called, the calling application is blocked during the close() call until the data is transmitted, or the connection has timed out. If this option is disabled, the close() call returns without blocking the caller and the TCP/IP address space still waits before trying to send the data. Although the data transfer is usually successful, it cannot be guaranteed, because the TCP/IP address space waits only a finite amount of time while trying to send the data. This option has meaning only for stream sockets.

SO_OOBINLINE
Toggles reception of out-of-band data. The default is *disabled*. When this option is enabled, it causes out-of-band data to be placed in the normal data input queue as it is received, making it available to recv(), recvfrom(), and recvmsg() without specifying the MSG_OOB flag in those calls. When this option is disabled, it causes out-of-band data to be placed in the priority data input queue as it is received, making it available to recv(), recvfrom(), and recvmsg() only by specifying the MSG_OOB flag in those calls. This option has meaning only for stream sockets.

SO_RCVBUF Returns the size of the data portion of the TCP/IP send buffer in *optval*. The size of the data portion of the receive buffer is protocol-specific.

SO_REUSEADDR
Toggles local address reuse. The default is disabled. This alters the normal algorithm used in the bind() call.

The normal bind() call algorithm allows each internet address and port combination to be bound only once. If the address and port have been bound already, a subsequent bind() will fail and result error EADDRINUSE.

After the SO_REUSEADDR option is active, the following situations are supported:

- A server can bind() the same port multiple times as long as every invocation uses a different local IP address, and the wildcard address INADDR_ANY is used only one time per port.
- A server with active client connections can be restarted and can bind to its port, without having to close all of the client connections.
- For datagram sockets, multicasting is supported so multiple bind() calls can be made to the same class D address and port number.

SO_SNDBUF Returns the size of the data portion of the TCP/IP send buffer in *optval*. The size of the data portion of the send buffer is protocol-specific.

SO_TYPE Returns the type of the socket. On return, the integer pointed to by *optval* is set to SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW.

IP_MULTICAST_TTL
Gets the IP time-to-live of outgoing multicast datagrams. The default value is 1 (multicast is available only to the local subnet).

IP_MULTICAST_LOOP
Enables or disables the loopback of outgoing multicast datagrams. The default value is enable.

IP_MULTICAST_IF
Gets the interface for sending outbound multicast datagrams from the socket application.

Note: Multicast datagrams can be transmitted only on one interface at a time.

Return Values

The value zero indicates success; the value negative one indicates an error. Errno identifies the specific error.

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

Example

The following are examples of the getsockopt() call. See “setsockopt()” on page 174 to see how the setsockopt() call options are set.

Example 1

```
#include <manifest.h>

int rc;
int s;
int optval;
int optlen;
struct linger l;
int getsockopt(int s, int level, int optname, char *optval, int *optlen);

⋮
/* Is out of band data in the normal input queue? */
optlen = sizeof(int);
rc = getsockopt(
        s, SOL_SOCKET, SO_OOBINLINE, (char *) &optval, &optlen);
if (rc == 0)
{
    if (optlen == sizeof(int))
    {
         if (optval)
            /* yes it is in the normalqueue */
         else
            /* no it is not                  */
    }
}

⋮
```

Example 2

```
/* Do I linger on close? */
optlen = sizeof(l);
rc = getsockopt(
        s, SOL_SOCKET, SO_LINGER, (char *) &l, &optlen);
if (rc == 0)
{
    if (optlen == sizeof(l))
    {
         if (l.l_onoff)
            /* yes I linger */
         else
            /* no I do not  */
    }
}
```

Related Calls

bind(), close(), getprotobyname(), setsockopt(), socket()

givesocket()

The givesocket() call tells TCP/IP to make the specified socket available to a takesocket() call issued by another program. Any connected stream socket can be given. Typically, givesocket() is used by a master program that obtains sockets by means of accept() and gives them to slave programs that handle one socket at a time.

This call can be used only in the AF_INET domain.

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

int  givesocket(int d, struct clientid *clientid)
```

Parameter **Description**

d
The descriptor of a socket to be given to another application.

clientid
Points to a client ID structure specifying the target program to which the socket is to be given.

To pass a socket, the master program first calls givesocket() with the client ID structure filled in as follows:

Field **Description**

domain
This call is supported only in the AF_INET domain.

name
The slave program address space name, left-justified and padded with blanks. The slave program can run in the same address space as the master program, in which case this field is set to the master program address space. If this field is set to blanks, any MVS address space can take the socket.

subtaskname
Specifies blanks.

reserved
Specifies binary zeros.

The master program then calls getclientid() to obtain its client ID, and passes its client ID, along with the descriptor of the socket to be given, to the slave program. One way to pass a socket is by passing the slave program startup parameter list.

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

Waiting for the slave program to take the socket, the master program uses select() to test the given socket for an exception condition. When select() reports that an exception condition is pending, the master program calls close() to free the given socket.

If your program closes the socket before a pending exception condition is indicated, the TCP connection is immediately reset, and the target program call to takesocket() call is unsuccessful. Calls other than the close() call issued on a given socket return a value of negative one, with errno set to EBADF.

Sockets that have been given and not taken for a period of four days will be closed, and become unavailable. If a *select* for the socket is outstanding, it is posted.

Return Values

The value zero indicates success. The value negative one indicates an error. Errno identifies a specific error.

Errno	Description
EBADF	The *d* parameter is not a valid socket descriptor. The socket has already been given. The socket domain is not AF_INET.
EBUSY	Listen() has been called for the socket.
EFAULT	Using the *clientid* parameter as specified would result in an attempt to access storage outside the caller address space.
EINVAL	The *clientid* parameter does not specify a valid client identifier.
ENOTCONN	The socket is not connected.
EOPNOTSUPP	The socket type is not SOCK_STREAM.

Related Calls

accept(), close(), getclientid(), listen(), select(), takesocket()

htonl()

The htonl() call translates a long integer from host byte order to network byte order.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>

unsigned long htonl(unsigned long a)
```

Parameter	Description
a	The unsigned long integer to be put into network byte order.

Return Values

Returns the translated long integer.

Related Calls

htons(), ntohs(), ntohl()

htons()

The htons() call translates a short integer from host byte order to network byte order.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>

unsigned short htons(unsigned short a)
```

Parameter	Description
a	The unsigned short integer to be put into network byte order.

Return Values

Returns the translated short integer.

Related Calls

ntohs(), htonl(), ntohl()

inet_addr()

The inet_addr() call interprets character strings representing host addresses expressed in standard dotted decimal notation and returns host addresses suitable for use as internet addresses.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>
#include <inet.h>

unsigned long inet_addr(char *cp)
```

Parameter	Description
cp	A character string in standard dotted decimal (.) notation.

Values specified in standard dotted decimal notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When a four-part address is specified, each part is interpreted as a byte of data and assigned, from left to right, to one of the four bytes of an internet address.

When a three-part address is specified, the last part is interpreted as a 16-bit quantity, and placed in the two rightmost bytes of the network address. This makes the three-part address a good format for specifying Class B network addresses as 128.net.host.

When a two-part address is specified, the last part is interpreted as a 24-bit quantity, and placed in the three rightmost bytes of the network address. This makes the two-part address a good format for specifying Class A network addresses as net.host.

When a one-part address is specified, the value is stored directly in the network address space without any rearrangement of its bytes.

Numbers supplied as address parts in standard dotted decimal notation can be decimal, hexadecimal, or octal. Numbers are interpreted using C language syntax. A leading *0x* implies hexadecimal; a leading zero implies octal. A number without a leading zero implies decimal.

Return Values

The internet address is returned in network byte order.

Negative one is returned as an error.

Related Calls

inet_lnaof(), inet_makeaddr(), inet_netof(), inet_network(), inet_ntoa()

inet_lnaof()

The inet_lnaof() call breaks apart the existing internet host address, and returns the local network address portion.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>
#include <inet.h>

unsigned long inet_lnaof(struct in_addr in)
```

Parameter	Description
in	The host internet address.

Return Values

The local network address is returned in host byte order.

Related Calls

inet_addr(), inet_makeaddr(), inet_netof(), inet_network(), inet_ntoa()

inet_makeaddr()

The inet_makeaddr() call combines an existing network number and a local network address to construct an internet address.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>
#include <inet.h>

struct in_addr
inet_makeaddr(unsigned long net, unsigned long lna)
```

Parameter	Description
net	The network number.
lna	The local network address.

Return Values

The internet address is returned in network byte order.

Related Calls

inet_addr(), inet_lnaof(), inet_netof(), inet_network(), inet_ntoa()

inet_netof()

The inet_netof() call breaks apart the existing internet host address, and returns the network number portion.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>
#include <inet.h>

unsigned long inet_netof(struct in_addr in)
```

Parameter	Description
in	The internet address in network byte order.

Return Values

The network number is returned in host byte order.

Related Calls

inet_addr(), inet_lnaof(), inet_makeaddr(), inet_ntoa()

inet_network()

The inet_network() call interprets character strings representing addresses expressed in standard dotted decimal notation, and returns numbers suitable for use as a network number.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>
#include <inet.h>

unsigned long inet_network(char *cp)
```

Parameter	Description
cp	A character string in standard, dotted decimal (.) notation.

Return Values

The network number is returned in host byte order.

Related Calls

inet_addr(), inet_lnaof(), inet_makeaddr(), inet_ntoa()

inet_ntoa()

The inet_ntoa() call returns a pointer to a string expressed in dotted decimal notation. The inet_ntoa() call accepts an internet address expressed as a 32-bit quantity in network byte order, and returns a string expressed in dotted decimal notation.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>
#include <inet.h>

char *inet_ntoa(struct in_addr in)
```

Parameter	Description
in	The host internet address.

Return Values

Returns a pointer to the internet address expressed in dotted decimal notation.

Related Calls

inet_addr(), inet_lnaof(), inet_makeaddr(), inet_network(), inet_ntoa()

ioctl()

The operating characteristics of sockets can be controlled using the ioctl() call.

This call can be used only in the AF_INET domain.

```
#include <manifest.h>
#include <socket.h>
#include <bsdtypes.h>
#include <ioctl.h>
#include <rtrouteh.h>
#include <if.h>
#include <ezbcmonc.h>
int ioctl(int s, unsigned long cmd, char *arg)
```

Parameter	**Description**
s	The socket descriptor.
cmd	The command to perform.
arg	Points to the data associated with *cmd*.

The operations to be controlled are determined by *cmd*. The *arg* parameter is Points to data associated with the particular command, and its format depends on the command being requested. The following are valid ioctl() keywords:

Keyword **Description**

FIONBIO Sets or clears nonblocking I/O for a socket. The variable *arg* points to an integer. If the integer is zero, nonblocking I/O on the socket is cleared; otherwise, the socket is set for nonblocking I/O.

FIONREAD Gets for the socket the number of immediately readable bytes. The variable *arg* points to an integer.

SIOCADDRT Adds a routing table entry. The variable *arg* points to a *rtentry* structure, as defined in RTROUTE.H. The routing table entry, passed as an argument, is added to the routing tables.

SIOCATMARK
Queries whether the current location in the data input is pointing to out-of-band data. The variable *arg* points to an integer of one when the socket points to a mark in the data stream for out-of-band data; otherwise, zero.

SIOCDELRT Deletes a routing table entry. The variable *arg* points to a *rtentry* structure, as defined in RTROUTE.H. If the structure exists, the routing table entry passed as an argument is deleted from the routing tables.

SIOCGIFADDR
Gets the network interface address. The variable *arg* points to an *ifreq* structure, as defined in IF.H. The interface address is returned in the argument.

SIOCGIFBRDADDR
Gets the network interface broadcast address. The variable *arg* points to an *ifreq* structure, as defined in IF.H. The interface broadcast address is returned in the argument.

SIOCGIFCONF
Gets the network interface configuration. The variable *arg* points to an *ifconf* structure, as defined in IF.H. The interface configuration is returned in the argument.

SIOCGIFDSTADDR
Gets the network interface destination address. The variable *arg* points to an *ifreq* structure, as defined in IF.H. The interface destination (point-to-point) address is returned in the argument.

SIOCGIFFLAGS
Gets the network interface flags. The variable *arg* points to an *ifreq* structure, as defined in IF.H. The interface flags are returned in the argument.

SIOCGIFMETRIC
Gets the network interface routing metric. The variable *arg* points to an *ifreq* structure, as defined in IF.H. The interface routing metric is returned in the argument.

SIOCGIFNETMASK
Gets the network interface network mask. The variable *arg* points to an *ifreq* structure, as defined in IF.H. The interface network mask is returned in the argument.

SIOCSIFMETRIC
Sets the network interface routing metric. The variable *arg* points to an *ifreq* structure, as defined in IF.H. The interface routing metric is set to the value passed in the argument.

SIOCGMONDATA
Returns the stack data reported by the CBSAMPLE program. The variable arg points to a MonDataIn structure as defined in EZBZMONC.H. The output is mapped by the MonDataOut structure as defined in EZBZMONC.H.

The ARP counter data provided differs depending on the type of device. Refer to the section on devices that support ARP Offload in the *OS/390 IBM Communications Server: IP Configuration Guide* for more information on what is supported for each device.

Return Values

The value zero indicates success; the value negative one indicates an error. Errno identifies the specific error.

Errno	**Description**
EBADF	The *s* parameter is not a valid socket descriptor.
EINVAL	The request is not valid, or not supported.
EFAULT	The arg is a bad pointer.

Example

```
int s;
int dontblock;
int rc;
⋮
/* Place the socket into nonblocking mode */
dontblock = 1;
rc = ioctl(s, FIONBIO, (char *) &dontblock);
⋮
```

listen()

The listen() call applies only to stream sockets. It performs two tasks: it completes the binding necessary for a socket *s*, if bind() has not been called for *s*, and it creates a connection request queue of length *backlog* to queue incoming connection requests. When the queue is full, additional connection requests are ignored.

The listen() call indicates a readiness to accept client connection requests. It transforms an active socket into a passive socket. Once called, *s* can never again be used as an active socket to initiate connection requests. Calling listen() is the third of four steps that a server performs when it accepts a connection. It is called after allocating a stream socket using socket(), and after binding a name to *s* using bind(). It must be called before calling accept().

```
#include <manifest.h>
#include <socket.h>
int listen(int s, int backlog)
```

Parameter	**Description**
s	Socket descriptor
backlog	Maximum queue length for pending connections

If the backlog is less than zero, *backlog* is set to zero. If the backlog is greater than SOMAXCONN, as defined in the TCPIP.PROFILE file, *backlog* is set to SOMAXCONN. There is a SOMAXCONN variable in the SOCKET.H file that is hardcoded at ten. If your C socket programs use this variable to determine the maximum listen() *backlog* queue length, remember to change the header file to reflect the value you specified for TCP/IP in TCPIP.PROFILE.

Return Values

The value zero indicates success; the value negative one indicates an error. Errno identifies the specific error.

Errno	**Description**
EBADF	The *s* parameter is not a valid socket descriptor.
EOPNOTSUPP	The *s* parameter is not a socket descriptor that supports the listen() call.

Related Calls

accept(), bind(), connect(), socket()

maxdesc()

The maxdesc() call reserves additional space in the TCP/IP address space to allow socket numbers to extend beyond the default range of zero through 49. Socket numbers zero, one, and two are never assigned, so the default maximum number of sockets is 47.

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

int maxdesc(int *totdesc, int *inetdesc)
```

Parameter	Description
totdesc	Points to an integer containing a value one greater than the largest socket number desired. The maximum allowed value is 2000. **Note:** If a totdesc value greater than 2000 is specified, the internal value is set to 2000. In all cases, use getdtablesize() to verify the value set by maxdesc().
inetdesc	Points to an integer containing a value one greater than the largest socket number usable for AF_INET sockets desired. The maximum allowed value is 2000.

Set to one more than the desired maximum socket number the integer pointed to by *totdesc*. If your program does not use AF_INET sockets, set to zero the integer pointed to by *inetdesc*. If your program uses AF_INET sockets, set the integer pointed to by *inetdesc* to the same value as *totdesc*; maxdesc() must be called before your program creates its first socket. Your program should use getdtablesize() to verify that the number of sockets has been changed.

Note: Increasing the size of the bit sets for the select() call must be done at compile time. To increase the size of the bit sets, before including BSDTYPES.H, define FD_SETSIZE to be the largest value of any socket. The default size of FD_SETSIZE is 255 sockets.

Return Values

The value zero indicates success. (Your application should check the integer pointed to by *inetdesc*. It might contain less than the original value, if there was insufficient storage available in the TCP/IP address space. In this case, the desired number of AF_INET sockets are not available.) The value negative one indicates an error. Errno identifies the specific error.

Errno	Description
EFAULT	Using the *totdesc* or *inetdesc* parameters as specified results in an attempt to access storage outside of the caller address space, or storage not able to be modified by the caller.
EALREADY	Your program called maxdesc() after creating a socket, or after a previous call to maxdesc().
EINVAL	Indicates that **totdesc* is less than **inetdesc*; **totdesc* is less than or equal to zero; or **inetdesc* is less than zero.
ENOMEM	Your address space lacks sufficient storage.

Example

```
int totdesc, inetdesc;
totdesc = 100;
inetdesc = 0;
rc = maxdesc(&totdesc, &inetdesc)
```

If successful, your application can create 97 sockets, all of type AF_IUCV. Socket numbers run from 3–99.

```
int totdesc, inetdesc;
totdesc = 100;
inetdesc = 100;
rc = maxdesc(&totdesc, &inetdesc)
```

If successful, your application can create 97 sockets, each of which can be of type AF_INET or AF_IUCV. The socket numbers run from 3–99.

Related Calls

select(), socket(), getdtablesize()

ntohl()

The ntohl() call translates a long integer from network byte order to host byte order.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>

unsigned long ntohl(unsigned long a)
```

Parameter	Description
a	The unsigned long integer to be put into host byte order.

Return Values

Returns the translated long integer.

Related Calls

htonl(), htons(), ntohs()

ntohs()

The ntohs() call translates a short integer from network byte order to host byte order.

```
#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>

unsigned short ntohs(unsigned short a)
```

Parameter	Description
a	The unsigned short integer to be put into host byte order.

Return Values

Returns the translated short integer.

Related Calls

ntohl(), htons(), htonl()

read()

The read() call reads data on a socket with descriptor *s*, and stores it in a buffer. The read() call applies only to connected sockets. This call returns as many as *len* bytes of data. If fewer than the number of bytes requested is available, the call returns the number currently available. If data is not available for the socket *s*, and *s* is in blocking mode, the read() call blocks the caller until data arrives. If data is not available, and *s* is in nonblocking mode, read() returns a negative one and sets errno to EWOULDBLOCK. See “ioctl()” on page 138, or “fcntl()” on page 99 for a description of how to set nonblocking mode.

If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard extra bytes. Stream sockets act like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and Application A sends 1000 bytes, each call to this function can return one byte, or ten bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, calling this function until all data has been received.

```
#include <manifest.h>
#include <socket.h>
int read(int s, char *buf, int len)
```

Parameter	Description
s	Socket descriptor
buf	Points to the buffer that receives the data
len	Length in bytes of the buffer pointed to by *buf*

Return Values

If successful, the number of bytes copied into the buffer is returned. The value ten indicates that the connection is closed. The value negative one indicates an error. Errno identifies the specific error.

Errno	Description
EBADF	Indicates that *s* is not a valid socket descriptor.
EFAULT	Using the *buf* and *len* parameters would result in an attempt to access storage outside the caller address space.
EWOULDBLOCK	Indicates an unconnected socket (RAW). **Note:** ENOTCONN is returned for TCP, and EINVAL is returned for UDP.
EMSGSIZE	For non-TCP sockets, this indicates that the length exceeds the maximum data size. This is determined by getsockopt() using SO_SNDBUF for the socket type (TCP, UDP, or RAW).

Related Calls

connect(), fcntl(), getsockopt(), ioctl(), readv(), recv(), recvmsg(), recvfrom(), select(), selectex(), send(), sendmsg(), sendto(), setsockopt(), socket(), write(), writev()

readv()

The readv() call reads data on a socket with descriptor *s*, and stores it in a set of buffers. The readv() call applies to connected sockets only.

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

int readv(int s, struc iovec *iov, int iovcnt)
```

Parameter	**Description**
s	The socket descriptor.
iov	Points to an iovec structure.
iovcnt	The number of buffers pointed to by the *iov* parameter.

The data is scattered into the buffers specified by iov[0]...iov[iovcnt–1]. The *iovec* structure is defined in UIO.H and contains the following variables:

Variable	**Description**
iov_base	Points to the buffer
iov_len	The length of the buffer

The readv() call applies only to connected sockets.

This call returns up to *len* bytes of data. If less than the number of bytes requested is available, the call returns the number currently available. If data is not available for the socket *s*, and *s* is in blocking mode, the readv() call blocks the caller until data arrives. If data is not available and *s* is in nonblocking mode, readv() returns a negative one and sets errno to EWOULDBLOCK. See "fcntl()" on page 99 or "ioctl()" on page 138 for a description how to set nonblocking mode. When a datagram packet is too long to fit in the supplied buffer, datagram sockets discard extra bytes. Stream sockets act like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and Application A sends 1000 bytes, each call to this function can return one byte, or ten bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop and call this function until all data has been received.

Return Values

If successful, the number of bytes read into the buffers is returned. The value zero indicates that the connection is closed. The value negative one indicates an error. Errno identifies the specific error.

Errno	**Description**
EBADF	Indicates that *s* is not a valid socket descriptor.
EFAULT	Using *iov* and *iovcnt* would result in an attempt to access storage outside the caller address space.
EINVAL	*Iovcnt* was not valid, or one of the fields in the *iov* array was not valid. Also returned for a NULL *iov* pointer.
EWOULDBLOCK	Indicates that *s* is in nonblocking mode, and data is not available to read.

Related Calls

connect(), fcntl(), getsockopt(), ioctl(), read(), recv(), recvmsg(), recvfrom(), select(), selectex(), send(), sendmsg(), sendto(), setsockopt(), socket(), write(), writev()

recv()

The recv() call receives data on a socket with descriptor *s* and stores it in a buffer. The recv() call applies only to connected sockets.

This call returns the length of the incoming message or data. If data is not available for socket *s*, and, *s* is in blocking mode, the recv() call blocks the caller until data arrives. If data is not available and *s* is in nonblocking mode, recv() returns a negative one and sets errno to EWOULDBLOCK. See “fcntl()” on page 99 or “ioctl()” on page 138 for a description of how to set nonblocking mode.

If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard extra bytes. Stream sockets act like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and Application A sends 1000 bytes, each call to this function can return one byte, or ten bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop and call this function until all data has been received.

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

int recv(int s, char *buf, int len, int flags)
```

Parameter **Description**

s Socket descriptor

buf Points to the buffer that receives the data

len Length in bytes of the buffer pointed to by *buf*

flags Set the *flags* parameter by specifying one or more of the following flags. If more than one flag is specified, the logical OR operator (|) must be used to separate them. Setting this parameter is supported only for sockets in the AF_INET domain. Setting these flags is not supported for sockets in the AF_IUCV domain.

MSG_OOB
Reads any out-of-band data on the socket. This is valid for stream (TCP) sockets only.

MSG_PEEK
Peeks at the data present on the socket; the data is returned but not consumed, so that a subsequent receive operation sees the same data.

Return Values

If successful, the byte length of the message or datagram is returned. The value negative one indicates an error. The value zero indicates connection closed. Errno identifies the specific error.

Errno **Description**

EBADF Indicates that *s* is not a valid socket descriptor.

EFAULT Using the *buf* and *len* parameters would result in an attempt to access storage outside the caller address space.

EWOULDBLOCK
Indicates that *s* is in nonblocking mode, and data is not available to read.

ENOTCONN Indicates an unconnected TCP socket.

EMSGSIZE For non-TCP sockets, this indicates that length exceeds the maximum data size as determined by getsockopt() using SO_SNDBUF for the socket type, either TCP, UDP, or RAW.

Related Calls

connect(), fcntl(), getsockopt(), ioctl(), read(), readv(), recvfrom(), recvmsg(), select(), selectex(), send(), sendmsg(), sendto(), setsockopt(), socket(), write(), writev()

recvfrom()

The recvfrom() call receives data on a socket by name with descriptor *s* and stores it in a buffer. The recvfrom() call applies to any datagram socket, whether connected or unconnected. For a datagram socket, when *name* is nonzero, the source address of the message is filled. Parameter *namelen* must first be initialized to the size of the buffer associated with *name*; then it is modified on return to identify indicate the actual size of the address stored there.

This call returns the length of the incoming message or data. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard extra bytes. If data is not available for the socket *s*, and *s* is in blocking mode, the recvfrom() call blocks the caller until data arrives. If data is not available, and *s* is in nonblocking mode, recvfrom() returns a negative one and sets errno to EWOULDBLOCK. See "fcntl()" on page 99 or "ioctl()" on page 138 to set nonblocking mode.

For datagram sockets, this call returns the entire datagram sent, providing the datagram can fit into the specified buffer. Stream sockets act like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket, and Application A sends 1000 bytes, each call to this function can return one byte, or ten bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, calling this function until all data has been received.

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

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

int recvfrom(int s, char *buf, int len, int flags,
struct sockaddr *name, int *namelen)
```

Parameter **Description**

s Socket descriptor

buf Pointer to the buffer to receive the data

len Length in bytes of the buffer pointed to by *buf*

flags A parameter that can be set to zero or MSG_PEEK, or MSG_OOB. Setting this parameter is supported only for sockets in the AF_INET domain. Setting these flags is not supported for sockets in the AF_IUCV domain.

MSG_OOB
Reads any out-of-band data on the socket. This is valid for stream (TCP) sockets only.

MSG_PEEK
Peeks at the data present on the socket; the data is returned but not consumed, so that a subsequent receive operation sees the same data.

name Points to a *socket address* structure from which data is received. If *name* is a nonzero value, the source address is returned (datagram sockets).

namelen Points to the size of *name* in bytes.

Return Values

If successful, the length of the message or datagram is returned in bytes. The value zero indicates that the connection is closed. The value negative one indicates an error. Errno identifies the specific error.

Errno	Description
EBADF	Indicates that *s* is not a valid socket descriptor.
EFAULT	Using the *buf* and *len* parameters would result in an attempt to access storage outside the caller address space.
EWOULDBLOCK	Indicates that *s* is in nonblocking mode, and data is not available to read.
ENOTCONN	Indicates an unconnected TCP socket.
EMSGSIZE	For non-TCP sockets, this indicates that length exceeds the maximum data size as determined by getsockopt() using SO_SNDBUF for the socket type, either TCP, UDP, or RAW.
EINVAL	Parameter *namelen* is not valid.

Related Calls

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

recvmsg()

The recvmsg() call receives messages on the socket with descriptor *s* and stores the messages in an array of message headers.

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

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

int recvmsg(int s, struct msghdr *msg, int flags)
```

Parameter **Description**

s Socket descriptor

msg Points to an msghdr structure

flags Set the *flags* parameter by specifying one or more of the following flags. If more than one flag is specified, the logical OR operator (|) must be used to separate them. Setting this parameter is supported only for sockets in the AF_INET domain. Setting these flags is not supported for sockets in the AF_IUCV domain.

MSG_OOB
Reads any out-of-band data on the socket. This is valid for stream (TCP) sockets only.

MSG_PEEK
Peeks at the data present on the socket; the data is returned but not consumed, so that a subsequent receive operation will see the same data.

A message header is defined by structure msghdr. The definition of this structure can be found in the SOCKET.H header file and contains the following elements:

Variable **Description**

msg_name An optional pointer to a buffer where the sender address is stored for datagram sockets.

msg_namelen The size of the address buffer.

msg_iov An array of iovec buffers into which the message is placed. An iovec buffer contains the following variables:

iov_base
Points to the buffer.

iov_len
The length of the buffer.

msg_iovlen The number of elements in the msg_iov array.

msg_accrights The access rights received. This field is ignored.

msg_accrightslen
The length of access rights received. This field is ignored.

The recvmsg() call applies to sockets, regardless whether they are in the connected state, except for TCP sockets, which must be connected.

This call returns the length of the data received. If data is not available for socket *s*, and *s* is in blocking mode, the recvmsg() call blocks the caller until data arrives. If data is not available, and *s* is in nonblocking mode, recvmsg() returns a negative one and sets errno to EWOULDBLOCK. See "fcntl()" on page 99 or "ioctl()" on page 138 to see how to set nonblocking mode.

If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard extra bytes. Stream sockets act like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket, and Application A sends 1000 bytes, each call to this function can return one byte, or ten bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, and call this function until all data has been received.

Return Values

If successful, the length of the message in bytes is returned. The value zero indicates that the connection is closed. The value negative one indicates an error. Errno identifies the specific error.

Errno **Description**

EBADF Indicates that *s* is not a valid socket descriptor.

EFAULT Using *msg* would result in an attempt to access storage outside the caller address space. Also returned when *msg_namelen* is not valid.

EWOULDBLOCK
Indicates that *s* is in nonblocking mode, and data is not available to read.

ENOTCONN Returned for an unconnected TCP socket.

EMSGSIZE For non-TCP sockets, this indicates that length exceeds the maximum data size determined by getsockopt() using SO_SNDBUF for the socket type (TCP, UDP, or RAW).

Related Calls

connect(), fcntl(), getsockopt(), ioctl(), read(), readv(), recv(), recvfrom(), select(), selectex(), send(), sendmsg(), sendto(), setsockopt(), socket(), write(), writev()

select()

The select() call monitors activity on a set of sockets looking for sockets ready for reading, writing, or with an exception condition pending.

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

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout)
```

Parameter **Description**

nfds The number of socket descriptors to be checked. This value should be one greater than the greatest number of sockets to be checked.

You can use the select() call to pass a bit set containing the socket descriptors for the sockets you want checked. The bit set is fixed in size using one bit for every possible socket. Use the *nfds* parameter to force select() to check only a subset of the allocated socket bit set.

If your application allocates sockets three, four, five, six, and seven, and you want to check all of your allocations, *nfds* should be set to eight, the highest socket descriptor you specified, plus one. If your application checks sockets three and four, *nfds* should be set to five.

Socket numbers are assigned starting with number three because numbers zero, one, and two are used by the C socket interface.

readfds Points to a bit set of descriptors to check for reading.

writefds Points to a bit set of descriptors to check for writing.

exceptfds Points to a bit set of descriptors to check for exception conditions pending.

timeout Points to the time to wait for select() to complete.

If *timeout* is not a NULL pointer, it specifies a maximum time to wait for the selection to complete. If *timeout* is a NULL pointer, the select() call blocks until a socket becomes ready. To poll the sockets and return immediately, *timeout* should be a non-NULL pointer to a zero-valued *timeval* structure.

If you are using both AF_INET and AF_IUCV sockets in the socket descriptor sets, the timer value is ignored and processed as if *timeout* were a non-NULL pointer to a zero-valued *timeval* structure.

To use select() as a timer in your program, do either of the following:

- Set the read, write, and except arrays to zero.
- Set nfds to be a NULL pointer.

To completely understand the implementation of the select() call, you must understand the difference between a socket and a port. TCP/IP defines ports to represent a certain process on a certain machine. A port represents the location of one process; it does not represent a connection between processes. In the MVS programming interface for TCP/IP, a socket describes an endpoint of communication. Therefore, a socket describes both a port and a machine. Like file descriptors, a socket is a small integer representing an index into a table of communication endpoints in a TCP/IP address space.

To test more than one socket at a time, place the sockets to be tested into a bit set of type FD_SET. A bit set is a string of bits where when X is an element of the set, the bit representing X is set to one. If X is not an element of the set, the bit representing X is set to zero. For example, if Socket 33 is an element of a bit set, then bit 33 is set to one. If Socket 33 is not an element of a bit set, then Bit 33 is set to zero.

Because the bit sets contain a bit for every socket that a process can allocate, the size of the bit sets is constant. The function getdtablesize() returns the number of sockets that your program can allocate. If your program needs to allocate a large number of sockets, use getdtablesize() and maxdesc() to increase the number of sockets that can be allocated. Increasing the size of the bit sets must be done when you compile the program. To increase the size of the bit sets, define FD_SETSIZE before including BSDTYPES.H. FD_SETSIZE is the largest value of any socket. It is defined to be 255 in BSDTYPES.H.

The following macros can manipulate bit sets.

Macro **Description**

FD_ZERO(&*fdset***)**
Sets all bits in bit set *fdset* to zero. After this operation, the bit set does not contain sockets as elements. This macro should be called to initialize the bit set before calling FD_SET() to set a socket as a member.

FD_SET(*sock***, &***fdset***)**
Sets the bit for the socket *sock* to a one, making *sock* a member of bit set *fdset*.

FD_CLR(*sock***, &***fdset***)**
Clears the bit for the socket *sock* in bit set *fdset*. This operation sets the appropriate bit to a zero.

FD_ISSET(*sock***, &***fdset***)**
Returns greater than zero if *sock* is a member of the bit set *fdset*. Returns zero if *sock* is not a member of *fdset*. (This operation returns the bit representing *sock*.)

A socket is ready to be read when incoming data is buffered for it, or when a connection request is pending. A call to accept(), read(), recv(), or recvfrom() does not block. To test whether any sockets are ready to be read, use FD_ZERO() to initialize the *readfds* bit set, and invoke FD_SET() for each socket to be tested.

A socket is ready to be written if there is buffer space for outgoing data. A nonblocking stream socket in the process of connecting (connect() returned EINPROGRESS) is selected for write when the connect() completes. A call to write(), send(), or sendto() does not block providing that the amount of data is less than the amount of buffer space. If a socket is selected for write, the amount of available buffer space is guaranteed to be at least as large as the size returned from using SO_SNDBUF with getsockopt(). To test whether any sockets are ready for writing, initialize *writefds* using FD_ZERO(), and use FD_SET() for each socket to be tested.

The select() call checks for a pending exception condition on the given socket, to indicate that the target program has successfully called takesocket(). When select() indicates a pending exception condition, your program calls close() to close the given socket. A socket has exception conditions pending if it has received out-of-band data. A stream socket that was given using givesocket() is selected for exception when another application successfully takes the socket using takesocket().

The programmer can pass NULL for any bit sets without sockets to test. For example, if a program need only check a socket for reading, it can pass NULL for both *writefds* and *exceptfds*.

Because the sets of sockets passed to select() are bit sets, the select() call must test each bit in each bit set before polling the socket for status. For efficiency, the *nfsd* parameter specifies the largest socket passed in any of the bit sets. The select() call then tests only sockets in the range zero to *nfsd*-1. Variable *nfsd* can be the result of getdtablesize(); but if the application has only two sockets and *nfsd* is the result of getdtablesize(), select() tests every bit in each bit set.

Return Values

The total number of ready sockets in all bit sets is returned. The value negative one indicates an error; check errno. The value zero indicates an expired time limit. If the return value is greater than zero, the sockets that are ready in each bit set are set to one. Sockets in each bit set that are not ready are set to zero. Use macro FD_ISSET() with each socket to test its status.

Errno	**Description**
EBADF	One of the bit sets specified an incorrect socket. (FD_ZERO() was probably not called before the sockets were set.)
EFAULT	One of the bit sets pointed to a value outside the caller address space.
EINVAL	One of the fields in the timeval structure is not valid.

Note: If the number of ready sockets is greater than 65,535, only 65,353 is reported.

Example

In the following example, select() is used to poll sockets for reading (socket r), writing (socket w), and exception (socket e) conditions.

```
/* sock_stats(r, w, e) - Print the status of sockets r, w, and e. */
int sock_stats(r, w, e)
int r, w, e;
{
   fd_set reading, writing, except;
   struct timeval timeout;
   int rc, max_sock;

   /* initialize the bit sets */
   FD_ZERO( &reading );
   FD_ZERO( &writing );
   FD_ZERO( &except );

   /* add r, w, and e to the appropriate bit set */
   FD_SET( r, &reading );
   FD_SET( w, &writing );
   FD_SET( e, &except );

   /* for efficiency, what's the maximum socket number? */
   max_sock = MAX( r, w );
   max_sock = MAX( max_sock, e );
   max_sock ++;

   /* make select poll by sending a 0 timeval */
   memset( &timeout, 0, sizeof(timeout) );

   /* poll */
   rc = select( max_sock, &reading, &writing, &except, &timeout );

   if ( rc < 0 ) {
        /* an error occurred during the select() */
        tcperror( "select" );
   }
   else if ( rc == 0 ) {
        /* none of the sockets were ready in our little poll */
        printf( "nobody is home.\n" );
   } else {
        /* at least one of the sockets is ready */
        printf("r is %s\n", FD_ISSET(r,&reading) ? "READY" : "NOT READY");
        printf("w is %s\n", FD_ISSET(w,&writing) ? "READY" : "NOT READY");
        printf("e is %s\n", FD_ISSET(e,&except)  ? "READY" : "NOT READY");
   }
}
```

Related Calls

getdtablesize(), maxdesc(), selectex()

selectex()

The selectex() call provides an extension to the select() call by allowing you to use an ECB or ECB list that defines an event not described by *readfs, writefds, or exceptfds*.

The selectex() call monitors activity on a set of different sockets until a timeout expires to see whether any sockets are ready for reading or writing, or if any exception conditions are pending. See “select()” on page 154 for more information about selectex().

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

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

Parameter **Description**

nfds The number of socket descriptors to be checked.

readfds Points to a bit set of descriptors to be checked for reading.

writefds Points to a bit set of descriptors to be checked for writing.

exceptfds Points to a bit set of descriptors to be checked for exception pending conditions.

timeout Points to the time to wait for selectex() to complete.

ecbptr Points to the event control block (ECB) or ECB list. For an ECB list, the high-order bit must be turned on in *ecbptr.* The last entry in the ECB list must also have its high-order bit set to one, signifying list end. The maximum ECBs allowed is 63.

Note: ECB list is only supported for AF_INET sockets.

Return Values

The total number of ready sockets (in all bit sets) is returned. The returned value negative one indicates an error. The returned value of zero indicates either an expired time limit or that the caller ECB has been posted. To determine which of these two conditions occurred, check the ECB value. If the value of the ECB is non-zero, then the ECB has been POSTed; otherwise, the time limit has expired. The caller must initialize the ECB value to zero before issuing selectex(). If the caller’s ECB has been POSTed, the caller descriptor sets are also set to zero. If the return value is greater than zero, the socket descriptors in each bit set that are ready are set to one. All others are set to zero.

Errno **Description**

EBADF One of the descriptor sets specified an incorrect descriptor.

EFAULT One of the parameters pointed to a value outside the caller address space.

EINVAL One of the fields in the *timeval* structure is not valid.

Note: If the number of ready sockets is greater than 65,535, only 65,353 is reported.

Related Calls

accept(), connect(), getdtablesize(), recv(), send(), select()

send()

The send() call sends datagrams on the socket with descriptor *s*. The send() call applies to all connected sockets.

If buffer space is not available to hold the socket data to be transmitted, and the socket is in blocking mode, send() blocks the caller until more buffer space becomes available. If the socket is in nonblocking mode, send() returns a negative one and sets errno to EWOULDBLOCK. See "fcntl()" on page 99 or "ioctl()" on page 138 to set nonblocking mode. See "select()" on page 154 for additional information.

For datagram sockets, this call sends the entire datagram, providing the datagram can fit into the TCP/IP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application wishes to send 1000 bytes, each call to this function can send one byte, or ten bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop and call this function until all data has been sent.

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

int send(int s, char *msg, int len, int flags)
```

Parameter **Description**

s Socket descriptor

msg Points to the buffer containing the message to transmit

len Length of the message pointed to by *msg*

flags Set the *flags* parameter by specifying one or more of the following flags. If more than one flag is specified, the logical OR operator (|) must be used to separate them. Setting this parameter is supported only for sockets in the AF_INET domain. Setting these flags is not supported in the AF_IUCV domain.

MSG_OOB
Sends out-of-band data on sockets that support this function. Only SOCK_STREAM sockets created in the AF_INET address family support out-of-band data.

MSG_DONTROUTE
The MSG_DONTROUTE option is turned on for the duration of the operation. This is usually used only by diagnostic or routing programs.

Return Values

No indication of failure to deliver is implicit in a send() routine. The value negative one indicates locally detected errors. Errno identifies the specific error.

Errno **Description**

EBADF Indicates that *s* is not a valid socket descriptor.

EFAULT Using the *msg* and *len* parameters would result in an attempt to access storage outside the caller address space.

ENOBUFS Buffer space is not available to send the message.

EWOULDBLOCK

Indicates that *s* is in nonblocking mode, and there is not enough space in TCP/IP to accept the data.

Related Calls

connect(), fcntl(), getsockopt(), ioctl(), read(), readv(), recv(), recvfrom(), recvmsg(), select(), selectex(), sendmsg(), sendto(), socket(), write(), writev()

sendmsg()

The sendmsg() call sends messages on a socket with descriptor *s* passed in an array of message headers.

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

int sendmsg(int s, struct msghdr *msg, int flags)
```

Parameter **Description**

s Socket descriptor

msg Points to an msghdr structure

flags Set the *flags* parameter by specifying one or more of the following flags. If more than one flag is specified, the logical OR operator (|) must be used to separate them. Setting this parameter is supported only for sockets in the AF_INET domain. Setting these flags is not supported in the AF_IUCV domain.

MSG_OOB
Sends out-of-band data on the socket.

MSG_DONTROUTE
The SO_DONTROUTE option is turned on for the duration of the operation; usually used by diagnostic or routing programs only.

A message header is defined by a msghdr. The definition of this structure can be found in the SOCKET.H header file and contains the following parameters.

Parameter **Description**

msg_name The pointer to the buffer containing the recipient address. This is required for datagram sockets where an explicit connect() has not been done.

msg_namelen The size of the address buffer. This is required for datagram sockets where an explicit connect() has not been done.

msg_iov An array of iovec buffers containing the message. The iovec buffer contains the following:

iov_base
Points to the buffer.

iov_len
The length of the buffer.

msg_iovlen The number of elements in the msg_iov array.

msg_accrights The access rights sent. This field is ignored.

msg_accrightslen
The length of the access rights sent. This field is ignored.

The sendmsg() call applies to sockets regardless whether they are in the connected state, and returns the length of the data sent

If there is not buffer available space to hold the socket data to be transmitted, and the socket is in blocking mode, sendmsg() blocks the caller until more buffer space becomes available. If the socket is in nonblocking mode, sendmsg() returns a negative one and sets errno to EWOULDBLOCK. See “fcntl()” on page 99 or “ioctl()” on page 138 to set nonblocking mode.

For datagram sockets, this call sends the entire datagram, providing the datagram can fit into the TCP/IP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application wishes to send 1000 bytes, each call to this function can send one byte, or ten bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, and call this function until all data has been sent.

Return Values

If successful, the length of the message in bytes is returned. The value negative one indicates an error. Errno identifies the specific error.

Errno	**Description**
EBADF	Indicates that *s* is not a valid socket descriptor.
EFAULT	Using *msg* would result in an attempt to access storage outside the caller address space.
EINVAL	Indicates that *msg_namelen* is not the size of a valid address for the specified address family.
EMSGSIZE	The message was too big to be sent as a single datagram.
ENOBUFS	Buffer space is not available to send the message.
EWOULDBLOCK	Indicates that *s* is in nonblocking mode, and there is not enough space in TCP/IP to accept the data.

Related Calls

connect(), fcntl(), getsockopt(), ioctl(), read(), readv(), recv(), recvfrom(), recvmsg(), select(), selectex(), send(), sendto(), setsockopt(), socket(), write(), writev()

sendto()

The sendto() call sends datagrams on the socket with descriptor *s*. The sendto() call applies to any datagram socket, whether connected or unconnected.

If there is not enough available buffer space to hold the socket data to be transmitted, and the socket is in blocking mode, sendto() blocks the caller until more buffer space becomes available. If the socket is in nonblocking mode, sendto() returns a negative one and sets errno to EWOULDBLOCK. See “fcntl()” on page 99 or “ioctl()” on page 138 to set nonblocking mode.

For datagram sockets, this call sends the entire datagram, providing the datagram can fit into the TCP/IP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application wishes to send 1000 bytes, each call to this function can send one byte, or ten bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, and call this function until all data has been sent.

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

int sendto(int s, char *msg, int len, int flags, struct sockaddr *to,
    int tolen)
```

Parameter **Description**

s
Socket descriptor

msg
Points to the buffer containing the message to be transmitted

len
Length of the message in the buffer pointed to by *msg*

flags
A parameter that can be set to zero or MSG_DONTROUTE. Setting this parameter is supported only for sockets in the AF_INET domain. Setting these flags is not supported in the AF_IUCV domain.

MSG_DONTROUTE
The SO_DONTROUTE option is turned on for the duration of the operation. This is usually used by diagnostic or routing programs only.

to
Address of the target

tolen
Size of the structure pointed to by *to*

Return Values

If successful, the number of characters sent is returned. The value negative one indicates an error. Errno identifies the specific error.

No indication of failure to deliver is implied in the return value of this call when used with datagram sockets.

Errno **Description**

EBADF
Indicates that *s* is not a valid socket descriptor.

EFAULT
Using the *msg* and *len* parameters would result in an attempt to access storage outside the caller address space.

EINVAL
Tolen is not the size of a valid address for the specified address family.

EMSGSIZE
The message was too big to be sent as a single datagram. The default is large-envelope-size.

ENOBUFS
Buffer space is not available to send the message.

EWOULDBLOCK
Indicates that *s* is in nonblocking mode, and there is not enough space in TCP/IP to accept the data.

Related Calls

read(), readv(), recv(), recvfrom(), recvmsg(), send(), select(), selectex(), sendmsg(), socket() write(), writev()

sethostent()

The sethostent() call opens and rewinds the *hlq*.HOSTS.ADDRINFO and *hlq*.HOSTS.SITEINFO data sets. The *hlq*.HOSTS.LOCAL data set contains information about known hosts. If the *stayopen* flag is nonzero, the *hlq*.HOSTS.SITEINFO data set remains open after each call.

The sethostent() call is available only when RESOLVE_VIA_LOOKUP is defined before *manifest.h* is included.

Note: Data sets *hlq*.HOSTS.LOCAL, *hlq*.HOSTS.ADDRINFO, and *hlq*.HOSTS.SITEINFO are described in *TCP/IP for MVS: Customization and Administration Guide*.

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

int sethostent(int stayopen)
```

Parameter	**Description**
stayopen	The flag to be set to prevent data set *hlq*.HOSTS.SITEINFO closing after each call to gethostent().

Return Values

The value zero indicates success; the value negative one indicates an error. Errno identifies the specific error, returning the errno value of the fopen() call.

Related Calls

endhostent(), endnetent(), gethostbyaddr(), gethostbyname(), gethostent()

setibmopt()

The setibmopt() call chooses the TCP/IP image with which to connect. It is used in conjunction with getibmopt(), which returns the number of TCP/IP images installed on a given MVS system and their names, versions, and states. With this information, the caller can dynamically choose the TCP/IP image with which to connect through the setibmopt() call.

Note: Images from pre-V3R2 releases of TCP/IP for MVS are excluded. The setibmopt() call is not meaningful for pre-V3R2 releases.

The setibmopt() call is optional. If setibmopt is not used, the standard method for determining the connecting TCP/IP image is followed. If setibmopt is used, it must be issued before any other socket calls that establish the connection to TCP/IP.

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

int setibmopt(int cmd, struct ibm_tcpimage *buf)

struct ibm_tcpimage {
    unsigned short status;
    unsigned short version;
    char name[8];
}
```

Parameter	**Description**
cmd	The command to perform. For TCP/IP V3R2 for MVS, IBMTCP_IMAGE is supported.
buf	The address of the buffer to be used.

Parameter *buf* is the address of the struct ibm_tcpimage buffer containing the name and version of the TCP/IP image to which the caller wishes to connect. The name must be left-justified and padded with blanks. The TCP/IP name is always the PROC name, left-justified and padded with blanks. The TCP/IP version and status are ignored. The caller is responsible to fill in *name* before issuing the call. If setibmopt is not one of the active TCP/IP supported images on the system, subsequent socket calls will fail. This call checks the validity of the contents of the *name* field in the structure pointed to by *buf*. It checks the validity by verifying that the TCP/IP name is in the list generated by a getibmopt () call. It does not check the *status* or *version* fields. This call sets the image of the connection to be created on another call.

Typically, the caller issues getibmopt() to verify the choice for the TCP/IP image. On successful return, the caller's choice will be honored when attempting the connection to TCP/IP.

Return Values

A zero indicates success; the value negative one indicates an error. Errno identifies the specific error.

Errno	**Description**
EOPNOTSUPP	This is not supported in this release of TCP/IP.
EALREADY	Your program is already connected to a TCP/IP image.

EFAULT Using *buf* would result in an attempt to copy the address into a portion of the caller address space into which information cannot be written.

setibmsockopt()

Like setsockopt() call, the setibmsockopt() call sets the options associated with a socket in the AF_INET domain. This call is for options specific to the IBM implementation of sockets.

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

int setibmsockopt(int s, int level, int optname, char *optval, int optlen)
```

Parameter	Description
s	Socket descriptor.
level	Level for which the option is being set. Only SOL_SOCKET is supported.
optname	The name of a specified socket option.
optval	Points to option data.
optlen	The length of the option data.

For option SO_NONBLOCKLOCAL, optval should point to an integer. The option is meaningful only for sockets that have been enabled for BULKMODE using the setibmsockopt() call with SO_BULKMODE. If optval points to one, the socket is placed in nonblocking mode. In nonblocking mode, when the application-side queue is empty, the socket library returns negative one on a receive and sets *errno* to EWOULDBLOCK. If optval points to zero, the socket is placed in blocking mode.

Before the application calls SO_NONBLOCKLOCAL, the socket is in blocking mode.

SO_IGNOREINCOMINGPUSH is another option to consider. This option is meaningful only for stream sockets. This option is effective only for connections established through an offload box. If optval points to one, the option is set. If optival points to zero, the option is off.

The SO_IGNOREINCOMINGPUSH option causes a receive call to return when:

- The requested length is reached.
- The internal TCP/IP length is reached.
- The peer application closes the connection.

The amount of data returned for each call is maximized and the amount of CPU time consumed by your program and TCP/IP is reduced.

This option is not appropriate to your operation if your program depends on receiving data before the connection is closed. For example, this option is appropriate for an FTP data connection, but not for a Telnet connection.

Return Values

The value zero indicates success; the value negative one indicates an error. Errno identifies the specific error.

Errno	Description
EBADF	The s parameter is not a valid socket descriptor.
EFAULT	Using *optval* and *optlen* parameters would result in an attempt to access storage outside the caller address space.

ENOPROTOOPT

The optname parameter is unrecognized, or the level parameter is not SOL_SOCKET.

Related Calls

getibmsockopt(), getsockopt(), ibmsflush(), setsockopt()

Example

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

{ struct ibm_bulkmode_struct bulkstr;
  int optlen, rc;

  optlen = sizeof(bulkstr);
  rc = getibmsockopt(s, SOL_SOCKET, SO_BULKMODE, (char *), &bulkstr, &optlen);
  if (rc < 0) {
      tcperror("on getibmsockopt()");
      exit(1);
  }
  fprintf(stream,"%d byte buffer available for outbound queue.\n",
          bulkstr.b_max_send_queue_size_avail);

  bulkstr.b_max_send_queue_size=bulkstr.b_max_send_queue_size_avail;
  bulkstr.b_onoff = 1;
  bulkstr.b_teststor = 0;
  bulkstr.b_move_data = 1;
  bulkstr.b_max_receive_queue_size = 65536;
  rc = setibmsockopt(s, SOL_SOCKET, SO_BULKMODE, (char *), &bulkstr, optlen);
  if (rc < 0) {
      tcperror("on setibmsockopt()");
      exit(1);
  }
}
```

setnetent()

The setnetent() call opens the *hlq*.HOSTS.SITEINFO data set and resets it to its starting point. The *hlq*.HOSTS.SITEINFO data set contains information about known networks. If the *stayopen* flag is nonzero, the *hlq*.HOSTS.SITEINFO data set remains open after each call to setnetent().

Note: Data sets *hlq*.HOSTS.SITEINFO data set is described in *TCP/IP for MVS: Customization and Administration Guide*.

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

int setnetent(int stayopen)
```

Parameter	**Description**
stayopen	A flag that can be set to prevent data set *hlq*.HOSTS.SITEINFO closing after every call to setnetent().

Return Values

The value zero indicates success; the value negative one indicates an error. Errno identifies the specific error, returning the errno value of the fopen() call.

Related Calls

endnetent(), endhostent(), getnetbyaddr(), getnetbyname(), getnetent()

setprotoent()

The setprotoent() call opens the *hlq*.ETC.PROTO data set and sets it to the data set starting point. If the *stayopen* flag is nonzero, the *hlq*.ETC.PROTO data set remains open after every call.

Note: The *hlq*.ETC.PROTO data set is described in *TCP/IP for MVS: Customization and Administration Guide*.

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

int setprotoent(int stayopen)
```

Parameter	Description
stayopen	A flag that can be set to prevent data set *hlq*.ETC.PROTO closing after every call to setprotoent().

Return Values

The value zero indicates success; the value negative one indicates an error. Errno identifies the specific error, returning the errno value of the fopen() call.

Related Calls

endprotoent(), getprotobyname(), getprotobynumber(), getprotoent()

setservent()

The setservent() call opens the *hlq*.ETC.SERVICES data set and resets it to its starting point. If the *stayopen* flag is nonzero, the *hlq*.ETC.SERVICES data set remains open after every call.

Note: The *hlq*.ETC.SERVICES data set is described in *TCP/IP for MVS: Customization and Administration Guide*.

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

int setservent(int stayopen)
```

Parameter	Description
stayopen	A flag that can be set to prevent data set *hlq*.ETC.SERVICES after each call to setservent().

Return Values

The value zero indicates success; the value negative one indicates an error. Errno identifies the specific error, returning the errno value of the fopen() call.

Related Calls

endservent(), getservbyname(), getservent()

setsockopt()

The setsockopt() call sets options associated with a socket. It can be called only for sockets in the AF_INET domain. Options can exist at multiple protocol levels; they are always present at the highest socket level.

When manipulating socket options, you must specify the level at which the option resides and the name of the option. To manipulate options at the socket level, the *level* parameter must be set to SOL_SOCKET, as defined in SOCKET.H. To manipulate options at any other level, such as the TCP or IP level, supply the appropriate protocol number for the protocol controlling the option. Currently, only the SOL_SOCKET level is supported. The getprotobyname() call can be used to return the protocol number for a named protocol.

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

int setsockopt(int s, int level, int optname, char *optval, int optlen)
```

Parameter	**Description**
s	The socket descriptor.
level	The level for which the option is being set. Only SOL_SOCKET is supported.
optname	The name of a specified socket option.
optval	The pointer to option data.
optlen	The length of the option data.

The *optval* and *optlen* parameters are used to pass data used by the particular set command. The *optval* parameter points to a buffer containing the data needed by the set command. The *optlen* parameter must be set to the size of the data pointed to by *optval*.

All of the socket level options except SO_LINGER expect *optval* to point to an integer and *optlen* to be set to the size of an integer. When the integer is nonzero, the option is enabled. For toggle type options, if the integer is nonzero, the option is enabled, if it is zero, the option is disabled. The SO_LINGER option expects *optval* to point to a *linger* structure, as defined in SOCKET.H. This structure is defined in the following example:

```
struct  linger
{
        int     l_onoff;                /* option on/off */
        int     l_linger;               /* linger time */
};
```

The *l_onoff* field is set to zero if the SO_LINGER option is begin disabled. A nonzero value enables the option. The *l_linger* field specifies the amount of time to wait on close. The units of *l_linger* are seconds.

The following keywords are recognized at the socket level:

Keyword	**Description**
SO_RCVBUF	Sets the size of the data portion of the TCP/IP receive buffer in

OPTVAL. The size of the data portion of the receive buffer is protocol-specific. If the requested size exceeds the allowed size, the following occurs:

- If the protocol is TCP, a return value of negative one and errno of ENOBUFS is set. The receive buffer size in unchanged.

  For maximum values for the TCP protocol, see the TCPCONFIG TCPRCVBUFRSIZE and TCPMAXRCVBUFSIZE parameters in the *OS/390 IBM Communications Server: IP Configuration Guide*
- If the protocol is UDP or RAW, a return value of zero is returned and the buffer size is set to 65535.

SO_SNDBUF Sets the size of the data portion of the TCP/IP send buffer in OPTVAL. The size of the data portion of the send buffer is protocol-specific. If the requested size exceeds the allowed size, the following occurs:

- If the protocol is TCP, a return value of negative one and errno of ENOBUFS is set. The send buffer size in unchanged.

  For maximum values for the TCP protocol, see the TCPCONFIG TCPSENDBUFRSIZE parameters in the *OS/390 IBM Communications Server: IP Configuration Guide*
- If the protocol is UDP or RAW, a return value of zero is returned and the buffer size is set to 65535.

SO_BROADCAST
Toggles the ability to broadcast messages. The default is *disabled*. If this option is enabled, it allows the application to send broadcast messages over *s* when the interface specified in the destination supports broadcasting of packets. This option has no meaning for stream sockets.

SO_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 ended with the error ETIMEDOUT.

SO_LINGER Lingers on close if data is present. The default is *disabled*. When this option is enabled and there is unsent data present when close() is called, the calling application is blocked during the close() call until the data is transmitted, or the connection has timed out. If this option is disabled, the close() call returns without blocking the caller, and the TCP/IP address space still waits to try to send the data. Although the data transfer is usually successful, it cannot be guaranteed, because the TCP/IP address space waits a finite amount of time while trying to send the data. This option has meaning for stream sockets only.

SO_OOBINLINE
Toggles the reception of out-of-band data. The default is *disabled*. When this option is enabled, it causes out-of-band data to be placed in the normal data input queue as it is received, making it available to recv(), recvfrom(), and recvmsg() without having to specify the MSG_OOB flag in those calls. When this option is disabled, it causes out-of-band data to be placed in the priority data input queue as it is received, making it available to recv(),

recvfrom(), and recvmsg() only by specifying the MSG_OOB flag in those calls. This option has meaning for stream sockets only.

SO_REUSEADDR

Toggles local address reuse. The default is disabled. This alters the normal algorithm used in the bind() call.

The normal bind() call algorithm allows each internet address and port combination to be bound only once. If the address and port have been bound already, a subsequent bind() will fail and result error EADDRINUSE.

After the 'SO_REUSEADDR' option is active, the following situations are supported:

- A server can bind() the same port multiple times as long as every invocation uses a different local IP address, and the wildcard address INADDR_ANY is used only one time per port.
- A server with active client connections can be restarted and can bind to its port, without having to close all of the client connections.
- For datagram sockets, multicasting is supported so multiple bind() calls can be made to the same class D address and port number.

IP_MULTICAST_TTL

Sets the IP time-to-live of outgoing multicast datagrams. The default value is 1 (multicast is available only to the local subnet).

IP_MULTICAST_LOOP

Enables or disables the loopback of outgoing multicast datagrams. The default value is enable.

IP_MULTICAST_IF

Sets the interface for sending outbound multicast datagrams from the socket application.

Note: Multicast datagrams can be transmitted only on one interface at a time.

IP_ADD_MEMBERSHIP

Joins a multicast group on a specific interface. An interface has to be specified with this option. Only applications that want to receive multicast datagrams need to join multicast groups.

IP_DROP_MULTICAST

Exits a multicast group.

Return Values

The value zero indicates success; the value negative one indicates an error. Errno identifies the specific error.

Errno	**Description**
EBADF	The *s* parameter is not a valid socket descriptor.
EFAULT	Using *optval* and *optlen* parameters would result in an attempt to access storage outside the caller address space.
ENOBUFS	No buffer space is available.

ENOPROTOOPT
The *optname* parameter is unrecognized, or the *level* parameter is not SOL_SOCKET.

Example

See “getsockopt()” on page 123 to see how the getsockopt() options set is queried.

```
int rc;
int s;
int optval;
struct linger l;
int setsockopt(int s, int level, int optname,char *optval, int optlen);
⋮
/* I want out of band data in the normal inputqueue */
optval = 1;
rc = setsockopt(s, SOL_SOCKET, SO_OOBINLINE, (char *) &optval, sizeof(int));

⋮
/* I want to linger on close */
l.l_onoff  = 1;
l.l_linger = 100;
rc = setsockopt(s, SOL_SOCKET, SO_LINGER, (char *) &l, sizeof(l));
```

Related Calls

fcntl(), getprotobyname(), getsockopt(), ioctl(), socket()

shutdown()

The shutdown() call shuts down all or part of a duplex connection. Parameter *how* sets the condition for shut down to the socket *s* connection.

If you issue a shutdown() for a socket that currently has outstanding socket calls pending, see Table 3 on page 35 to determine the effects of this operation on the outstanding socket calls.

Note: Issue a shutdown() call before issuing a close() call for a socket.

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

int shutdown(int s, int how)
```

Parameter	Description
s	The socket descriptor.
how	The *how* condition can have a value of zero, one, or two, where: • Zero ends further receive operations. • One ends further send operations. • Two ends further send and receive operations.

Return Values

The value zero indicates success; the value negative one indicates an error. Errno identifies the specific error.

Errno Description

EBADF
Indicates that *s* is not a valid socket descriptor.

EINVAL
The *how* parameter was not set to a valid value: zero, one, or two.

sock_debug()

The sock_debug() call provides the socket library tracing facility. The *onoff* parameter can have a value of zero or nonzero. If *onoff*=0 (the default), no socket library tracing is done. If *onoff*=nonzero, the system traces for socket library calls and interrupts.

Note: You can include the statement SOCKDEBUG in date set TCPIP.DATA, as an alternative to calling sock_debug() with *onoff* not equal to zero.

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

void sock_debug(init onoff)
```

Parameter	**Description**
onoff	A parameter that can be set to zero or nonzero.

Related Calls

accept(), close(), connect(), socket()

sock_do_teststor()

The sock_do_teststor() call is used to check for calls that attempt to access storage outside the caller address space.

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

void sock_do_teststor(int onoff)
```

Parameter	Description
onoff	A parameter that can be set to zero or nonzero.

If *onoff* is not zero for either inbound or outbound sockets, both the address of the message buffer and the message buffer itself are checked for addressability at every socket call. The error condition, EFAULT, is set if there is an addressing problem. If *onoff* is set to zero, address checking is not done by the socket library program. If an error occurs when *onoff* is zero, normal runtime error handling reports the exception condition.

The default for *onoff* is zero. Addresses are not checked for addressability for parameters of C socket calls. While you are testing your program, you might find it useful to set *onoff* to a nonzero value.

Notes:

1. You can include the statement SOCKNOTESTSTOR in data set TCPIP.DATA, as an alternative to calling sock_do_teststor() with *onoff* equal to zero.
2. You can include the statement SOCKTESTSTOR in the data set TCPIP.DATA, which is in the client's catalog when the socket program is started, as an alternative to calling sock_do_teststor() with *onoff* not equal to zero.

Restrictions

None

socket()

The socket() call creates an endpoint for communication and returns a socket descriptor representing that endpoint. Different types of sockets provide different communication services.

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

int socket(int domain, int type, int protocol)
```

Parameter	**Description**
domain	The address domain requested. It is either AF_INET or AF_IUCV.
type	The type of socket created, either SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW.
protocol	The protocol requested. Possible values are zero, IPPROTO_UDP, or IPPROTO_TCP.

The *domain* parameter specifies the communication domain within which communication is to take place. This parameter specifies the address family (format of addresses within a domain) to be used. The families supported are AF_INET, which is the internet domain, and AF_IUCV, which is the IUCV domain. These constants are defined in the SOCKET.H header file.

The *type* parameter specifies the type of socket created. The type is analogous to the communication requested. These socket type constants are defined in the SOCKET.H header file. The types supported are:

Socket Type **Description**

SOCK_STREAM
Provides sequenced, two-way byte streams that are reliable and connection-oriented. They support a mechanism for out-of-band data. This type is supported in both the AF_INET and AF_IUCV domains.

SOCK_DGRAM
Provides datagrams, which are connectionless messages, of a fixed maximum length whose reliability is not guaranteed. Datagrams can be corrupted, received out of order, lost, or delivered repeatedly. This type is supported in the AF_INET domain only.

SOCK_RAW Provides the interface to internal protocols (such as IP and ICMP). This type is supported in the AF_INET domain only.

Note: To use raw sockets, the application must APF-authorized.

The *protocol* parameter specifies the particular protocol to be used with the socket. In most cases, a single protocol exists to support a particular type of socket within a particular addressing family (not true with raw sockets). If the *protocol* parameter is set to zero, the system selects the default protocol number for the domain and socket type requested. Protocol numbers are found in the *hlq*.ETC.PROTO data set. Alternatively, the getprotobyname() call can be used to get the protocol number for a protocol with a known name. The *protocol* field must be set to zero if the *domain* parameter is set to AF_IUCV. The *protocol* defaults are TCP for stream sockets and UDP for datagram sockets. There is no default for raw sockets.

SOCK_STREAM sockets model duplex byte streams. They provide reliable, flow-controlled connections between peer applications. Stream sockets are either active or passive. Active sockets are used by clients who initiate connection requests using connect(). By default, socket() creates active sockets. Passive sockets are used by servers to accept connection requests from the connect() call. An active socket is transformed into a passive socket by binding a name to the socket using the bind() call, and by indicating a willingness to accept connections with the listen() call. Once a socket is passive, it cannot be used to initiate connection requests.

In the AF_INET domain, the bind() call applied to a stream socket lets the application specify the networks from which it will accept connection requests. The application can fully specify the network interface by setting the *internet address* field in the *address* structure to the internet address of a network interface. Alternatively, the application can use a *wildcard* to specify that it wants to receive connection requests from any network. This is done by setting the *internet address* field within the *address* structure to the constant INADDR_ANY, as defined in the SOCKET.H header file.

Once a connection has been established between stream sockets, any of the data transfer calls can be used: (read(), write(), send(), recv(), readv(), writev(), sendto(), recvfrom(), sendmsg(), and recvmsg()). Usually, the read-write or send-recv pairs are used to send data on stream sockets. If out-of-band data is to be exchanged, the send-recv pair is normally used.

SOCK_DGRAM sockets model datagrams. They provide connectionless message-exchange without guarantee of reliability. Messages sent are limited in size. Datagram sockets are not supported in the AF_IUCV domain.

There is no active or passive analogy to stream sockets with datagram sockets. Servers must still call bind() to name a socket and to specify from which network interface from which it wants to receive packets. Wildcard addressing, as described for stream sockets, applies to datagram sockets also. Because datagram sockets are connectionless, the listen() call has no meaning for them, and must not be used with them.

After an application has received a datagram socket, it can exchange datagrams using the sendto() and recvfrom(), or sendmsg() and recvmsg() calls. If the application goes one step further by calling connect() and fully specifying the name of the peer with which all messages are to be exchanged, then the other data transfer calls of read(), write(), readv(), writev(), send(), and recv() can be used also. See “connect()” on page 92, for more information about placing a socket into the connected state.

Datagram sockets allow messages to be broadcast to multiple recipients. Setting the destination address to a broadcast address (depends on the class of address, and whether subnets are used). The constant INADDR_BROADCAST, defined in socket.h, can be used to broadcast to the primary network when the primary network configured supports broadcast.

SOCK_RAW sockets give the application an interface to lower layer protocols, such as IP and ICMP. This interface is often used to bypass the transport layer when direct access to lower layer protocols is needed. Raw sockets are also used to test new protocols. Raw sockets are not supported in the AF_IUCV domain.

Raw sockets are connectionless and data transfer semantics are the same as those described previously for datagram sockets. The connect() call can be used similarly to specify the peer.

Outgoing packets have an IP header prefixed to them. IP options can be set and inspected using the setsockopt() and getsockopt() calls respectively. Incoming packets are received with the IP header and options intact.

Notes:

1. Sockets are deallocated using the close() call.
2. Only SOCK_STREAM sockets are supported in the AF_IUCV domain.
3. The setsockopt() and getsockopt() calls are not supported for sockets in the AF_IUCV domain.
4. The flags field in the send(), recv(), sendto(), recvfrom(), sendmsg(), and recvmsg() calls is not supported in the AF_IUCV domain.

Return Values

A nonnegative socket descriptor indicates success. The value negative one indicates an error. Errno identifies the specific error.

Errno Description

EPROTONOSUPPORT
The *protocol* is not supported in this *domain* or this socket *type*.

EACCES
Access denied. The application is not an APF-authorized application.

EAFNOSUPPORT
The specified address family is not supported by this protocol family.

Example

```
int s;
struct protoent *p;
struct protoent *getprotobyname(char *name);
int socket(int domain, int type, int protocol);
⋮
/* Get stream socket in internetdomain with default protocol */
s = socket(AF_INET, SOCK_STREAM, 0);
⋮
/* Get stream socket in iucvdomain with default protocol */
s = socket(AF_IUCV, SOCK_STREAM, 0);
⋮
/* Get raw socket in internetdomain for ICMP protocol */
p = getprotobyname("iucv");
s = socket(AF_INET, SOCK_RAW, p->p_proto);
```

Related Calls

accept(), bind(), close() connect(), fcntl(), getprotobyname(), getsockname(), getsockopt(), ioctl(), maxdesc(), read(), readv(), recv(), recvfrom(), recvmsg(), select(), selectex(), send(), sendmsg(), sendto(), shutdown(), write(), writev()

takesocket()

The takesocket() call acquires a socket from another program. Typically, the other program passes its client ID and socket descriptor to your program through your program startup parameter list.

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

int takesocket(struct clientid *clientid, int hisdesc)
```

Parameter **Description**

clientid Points to the *clientid* of the application from which you are taking a socket.

hisdesc Describes the socket to be taken.

The takesocket() call acquires a socket from another program. Typically, the other program passes its client ID and socket descriptor to your program through your program startup parameter list.

Return Values

A nonnegative socket descriptor indicates success. The value negative one indicates an error. Errno identifies a specific error.

Errno **Description**

EACCES The other application did not give the socket to your application.

EBADF The *hisdesc* parameter does not specify a valid socket descriptor owned by the other application. The socket has already been taken.

EFAULT Using the *clientid* parameter as specified would result in an attempt to access storage outside the caller address space.

EINVAL The *clientid* parameter does not specify a valid client identifier.

EMFILE The socket descriptor table is already full.

ENOBUFS The operation cannot be performed because of a shortage of control blocks (SCB or SKCB) in the TCP/IP address space.

EPFNOSUPPORT
The domain field of the *clientid* parameter is not AF_INET.

Related Calls

getclientid(), givesocket()

tcperror()

When a socket call produces an error, the call returns a negative value and the variable *errno* is set to an error value found in TCPERRNO.H. The tcperror() call prints a short error message describing the last error that occurred. If *s* is non-NULL, tcperror() prints the string *s* followed by a colon, followed by a space, followed by the error message, and terminating with a new-line character. If *s* is NULL or points to a NULL string, only the error message and the new-line character are output.

The tcperror() function is equivalent to the UNIX perror() function.

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

void tcperror(char *s)
```

Parameter	**Description**
s	A NULL or NULL-terminated character string

Example

Example 1

```
if   ((s=socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
     tcperror("socket()");
     exit(2);
}
```

If the socket() call produces error ENOMEM, socket() returns a negative value and *errno* is set to ENOMEM. When tcperror() is called, it prints the string:

```
    socket():  Not enough storage (ENOMEM)
```

Example 2

```
if ((s=socket(AF_INET, SOCK_DGRAM, 0)) < 0)
   tcperror(NULL);
```

If the socket() call produces error ENOMEM, socket() returns a negative value and *errno* is set to ENOMEM. When tcperror() is called, it prints the string:

```
     Not enough storage (ENOMEM)
```

write()

The write() call writes data from a buffer on a socket with descriptor *s*. The write() call applies only to connected sockets.

This call writes up to *len* bytes of data.

If there is not enough available buffer space to hold the socket data to be transmitted, and the socket is in blocking mode, write() blocks the caller until more buffer space is available. If the socket is in nonblocking mode, write() returns a negative one and sets *errno* to EWOULDBLOCK. See “fcntl()” on page 99, or “ioctl()” on page 138 to set nonblocking mode.

For datagram sockets, this call sends the entire datagram, providing the datagram can fit into the TCP/IP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application wishes to send 1000 bytes, each call to this function can send one byte or ten bytes or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, and call this function until all data has been sent.

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

int write(int s, char *buf, int len)
```

Parameter	**Description**
s	Socket descriptor
buf	Points to the buffer holding the data to be written
len	Length in bytes of *buf*

Return Values

If successful, the number of bytes written is returned. The value negative one indicates an error. Errno identifies the specific error.

Errno	**Description**
EBADF	Indicates that *s* is not a valid socket descriptor.
EFAULT	Using the *buf* and *len* parameters would result in an attempt to access storage outside the caller address space.
ENOBUFS	Buffer space is not available to send the message.
EWOULDBLOCK	Indicates that *s* is in nonblocking mode, and there is not enough space in TCP/IP to accept the data.

Related Calls

connect(), fcntl(), getsockopt(), ioctl(), read(), readv() recv(), recvfrom(), recvmsg(), select(), selectex(), send(), sendmsg(), sendto(), setsockopt(), socket(), writev()

writev()

The writev() call writes data from a set of buffers on a socket using descriptor *s*.

The writev() call applies only to connected sockets.

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

int writev(int s, struct iovec *iov, int iovcnt)
```

Parameter	**Description**
s	Socket descriptor
iov	Points to an array of iovec buffers
iovcnt	Number of buffers in the array

The data is gathered from the buffers specified by iov[0]...iov[iovcnt−1]. The *iovec* structure is defined in UIO.H and contains the following fields:

Parameter	**Description**
iov_base	Points to the buffer
iov_len	The length of the buffer.

This call writes the sum of the *iov_len* bytes of data.

If buffer space is not available to hold the socket data to be transmitted, and the socket is in blocking mode, writev() blocks the caller until additional buffer space becomes available. If the socket is in a nonblocking mode, writev() returns a negative one and sets *errno* to EWOULDBLOCK. For a description of how to set nonblocking mode, see “fcntl()” on page 99 or “ioctl()” on page 138.

For datagram sockets, this call sends the entire datagram, providing the datagram can fit into the TCP/IP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application wishes to send 1000 bytes, each call to this function can send one byte, or ten bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, calling this function until all data has been sent.

Return Values

If successful, the number of bytes written from the buffer(s) is returned. The value negative one indicates an error. Errno identifies the specific error.

Errno **Description**

EBADF Indicates that *s* is not a valid socket descriptor.

EFAULT Using the *iov* and *iovcnt* parameters would result in an attempt to access storage outside the caller address space.

ENOBUFS Buffer space is not available to send the message.

EWOULDBLOCK
Indicates that *s* is in nonblocking mode, and there is not enough space in TCP/IP to accept the data.

Related Calls

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

Sample C Socket Programs

This section contains sample C socket programs. The C source code can be found in the *hlq*.SEZAINST data set.

Following are the sample socket programs available:

Program	Description
TCPC	C socket TCP client
TCPS	C socket TCP server
UDPC	C socket UDP client
UDPS	C socket UDP server

For samples of the multitasking C programs in the following table see, “Appendix A. Multitasking C Socket Sample Program” on page 535.

Program	Description
MTCCLNT	C socket MTC client
MTCSRVR	C socket MTC server
MTCCSUB	C socket subtask MTCCSUB

Executing TCPS and TCPC Modules

To start the TCPS server, execute TCPS 9999 on the other MVS address space (server).

To run the TCPC client, execute TCPC MVS13 9999. (MVS13 is the host name where the TCPS server is running, and 9999 is the port you have assigned.)

After executing the TCPC client, the following output is displayed on the server session:

```
Server Ended Successfully
```

Executing UDPS and UDPC Modules

To start the UDPS server, execute UDPS on the other MVS address space (server). The following message is displayed:

```
Port assigned is 1028
```

To run the UDPC client, execute UDPC 9.67.60.10 1028. (Address 9.67.60.10 is the IP machine address where the UDPS server is running, and 1028 is the port assigned by the UDPS server.)

After executing the UDPC client, the following message is displayed:

```
Received Message Hello....
```

C Socket TCP Client (TCPC)

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

```
/*** IBMCOPYR ********************************************************/
/*                                                                   */
/* Part Name: TCPC                                                   */
/*                                                                   */
/* Copyright:                                                         */
/*   Licensed Materials - Property of IBM                             */
/*   This product contains "Restricted Materials of IBM"              */
/*   5735-FAL (C) Copyright IBM Corp. 1992.                           */
/*   5655-HAL (C) Copyright IBM Corp. 1992, 1996.                     */
/*   All rights reserved.                                             */
/*   US Government Users Restricted Rights -                          */
/*   Use, duplication or disclosure restricted by GSA ADP Schedule   */
/*   Contract with IBM Corp.                                          */
/*   See IBM Copyright Instructions.                                  */
/*                                                                    */
/*  TCP/IP for MVS                                                    */
/*  SMP/E Distribution Name: EZAEC01V                                 */
/*                                                                    */
/*** IBMCOPYR ********************************************************/
static char ibmcopyr[] =
   "TCPC     - Licensed Materials - Property of IBM. "
   "This module is \"Restricted Materials of IBM\" "
   "5735-FAL (C) Copyright IBM Corp. 1992. "
   "5655-HAL (C) Copyright IBM Corp. 1996. "
   "See IBM Copyright Instructions.";
/*
 * Include Files.
 */
#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>
#include <socket.h>
#include <netdb.h>
#include <stdio.h>
/*
 * Client Main.
 */
main(argc, argv)
int argc;
char **argv;
{
    unsigned short port;       /* port client will connect to         */
    char buf[12];              /* data buffer for sending & receiving */
    struct hostent *hostnm;    /* server host name information        */
    struct sockaddr_in server; /* server address                      */
    int s;                     /* client socket                       */
    /*
     * Check Arguments Passed. Should be hostname and port.
     */
    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s hostname port\n", argv[0]);
        exit(1);
    }
    /*
     * The host name is the first argument. Get the server address.
     */
    hostnm = gethostbyname(argv[1]);
    if (hostnm == (struct hostent *) 0)
    {
        fprintf(stderr, "Gethostbyname failed\n");
        exit(2);
    }
    /*
     * The port is the second argument.
     */
    port = (unsigned short) atoi(argv[2]);
    /*
     * Put a message into the buffer.
     */
    strcpy(buf, "the message");
    /*
     * Put the server information into the server structure.
     * The port must be put into network byte order.
     */
    server.sin_family      = AF_INET;
    server.sin_port        = htons(port);
    server.sin_addr.s_addr = *((unsigned long *)hostnm->h_addr);
    /*
     * Get a stream socket.
     */
    if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        tcperror("Socket()");
        exit(3);
    }
    /*
     * Connect to the server.
     */
    if (connect(s, (struct sockaddr *)&server, sizeof(server)) < 0)
    {
        tcperror("Connect()");
        exit(4);
    }
    if (send(s, buf, sizeof(buf), 0) < 0)
    {
        tcperror("Send()");
        exit(5);
    }
    /*
     * The server sends back the same message. Receive it into the
     * buffer.
     */
    if (recv(s, buf, sizeof(buf), 0) < 0)
    {
        tcperror("Recv()");
        exit(6);
    }
    /*
     * Close the socket.
     */
    close(s);
    printf("Client Ended Successfully\n");
    exit(0);
}
```

C Socket TCP Server (TCPS)

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

```
/*** IBMCOPYR *********************************************************/
/*                                                                    */
/* Component Name: TCPS                                               */
/*                                                                    */
/* Copyright:                                                         */
/*   Licensed Materials - Property of IBM                             */
/*   This product contains "Restricted Materials of IBM"              */
/*   5735-FAL (C) Copyright IBM Corp. 1992.                           */
/*   5655-HAL (C) Copyright IBM Corp. 1992, 1996.                     */
/*   All rights reserved.                                             */
/*   US Government Users Restricted Rights -                          */
/*   Use, duplication or disclosure restricted by GSA ADP Schedule   */
/*   Contract with IBM Corp.                                          */
/*   See IBM Copyright Instructions.                                  */
/*                                                                    */
/*  TCP/IP for MVS                                                    */
/*  SMP/E Distribution Name: EZAEC01X                                 */
/*                                                                   */
/*** IBMCOPYR ********************************************************/
static char ibmcopyr[] =
   "TCPS     - Licensed Materials - Property of IBM. "
   "This module is \"Restricted Materials of IBM\" "
   "5735-FAL (C) Copyright IBM Corp. 1992. "
   "5655-HAL (C) Copyright IBM Corp. 1996. "
   "See IBM Copyright Instructions.";
#include <manifest.h>
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>
#include <netdb.h>
#include <stdio.h>
/*
 * Server Main.
 */
main(argc, argv)
int argc;
char **argv;
{
    unsigned short port;       /* port server binds to                */
    char buf[12];              /* buffer for sending & receiving data */
    struct sockaddr_in client; /* client address information          */
    struct sockaddr_in server; /* server address information          */
    int s;                     /* socket for accepting connections    */
    int ns;                    /* socket connected to client          */
    int namelen;               /* length of client name               */
    /*
     * Check arguments. Should be only one: the port number to bind to.
     */
    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s port\n", argv[0]);
        exit(1);
    }
    /*
     * First argument should be the port.
     */
    port = (unsigned short) atoi(argv[1]);
    /*
     * Get a socket for accepting connections.
     */
    if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        tcperror("Socket()");
        exit(2);
    }
    /*
     * Bind the socket to the server address.
     */
    server.sin_family = AF_INET;
    server.sin_port   = htons(port);
    server.sin_addr.s_addr = INADDR_ANY;
    if (bind(s, (struct sockaddr *)&server, sizeof(server)) < 0)
    {
        tcperror("Bind()");
        exit(3);
    }
    /*
     * Listen for connections. Specify the backlog as 1.
     */
    if (listen(s, 1) != 0)
    {
        tcperror("Listen()");
        exit(4);
    }
    /*
     * Accept a connection.
     */
    namelen = sizeof(client);
    if ((ns = accept(s, (struct sockaddr *)&client, &namelen)) == -1)
    {
        tcperror("Accept()");
        exit(5);
    }
    /*
     * Receive the message on the newly connected socket.
     */
    if (recv(ns, buf, sizeof(buf), 0) == -1)
    {
        tcperror("Recv()");
        exit(6);
    }
    /*
     * Send the message back to the client.
     */
    if (send(ns, buf, sizeof(buf), 0) < 0)
    {
        tcperror("Send()");
        exit(7);
    }
    close(ns);
    close(s);
    printf("Server ended successfully\n");
    exit(0);
}
```

C Socket UDP Server (UDPS)

The following is an example of a C socket UDP server program. The source code can be found in the UDPS member of the *hlq*..SEZAINST data set.

```
/*** IBMCOPYR ********************************************************/
/*                                                                   */
/* Component Name: UDPS                                              */
/*                                                                   */
/* Copyright:                                                        */
/*   Licensed Materials - Property of IBM                            */
/*   This product contains "Restricted Materials of IBM"             */
/*   5735-FAL (C) Copyright IBM Corp. 1992.                          */
/*   5655-HAL (C) Copyright IBM Corp. 1992, 1996.                    */
/*   All rights reserved.                                            */
/*   US Government Users Restricted Rights -                         */
/*   Use, duplication or disclosure restricted by GSA ADP Schedule   */
/*   Contract with IBM Corp.                                         */
/*   See IBM Copyright Instructions.                                 */
/*                                                                   */
/*  TCP/IP for MVS                                                   */
/*  SMP/E Distribution Name: EZAEC021                                */
/*                                                                   */
/*** IBMCOPYR ********************************************************/
static char ibmcopyr[] =
   "UDPS     - Licensed Materials - Property of IBM. "
   "This module is \"Restricted Materials of IBM\" "
   "5735-FAL (C) Copyright IBM Corp. 1992. "
   "5655-HAL (C) Copyright IBM Corp. 1992, 1996. "
   "See IBM Copyright Instructions.";
#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>
#include <socket.h>
#include <netdb.h>
#include <stdio.h>
```

```
main()
{
   int s, namelen, client_address_size;
   struct sockaddr_in client, server;
   char buf[32];
   /*
    * Create a datagram socket in the internet domain and use the
    * default protocol (UDP).
    */
   if ((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
   {
       tcperror("socket()");
       exit(1);
   }
   /*
    * Bind my name to this socket so that clients on the network can
    * send me messages. (This allows the operating system to demultiplex
    * messages and get them to the correct server)
    *
    * Set up the server name. The internet address is specified as the
    * wildcard INADDR_ANY so that the server can get messages from any
    * of the physical internet connections on this host. (Otherwise we
    * would limit the server to messages from only one network
    * interface.)
    */
   server.sin_family      = AF_INET;  /* Server is in Internet Domain */
   server.sin_port        = 0;         /* Use any available port      */
   server.sin_addr.s_addr = INADDR_ANY;/* Server's Internet Address   */
   if (bind(s, (struct sockaddr *)&server, sizeof(server)) < 0)
   {
       tcperror("bind()");
       exit(2);
   }
   /* Find out what port was really assigned and print it */
   namelen = sizeof(server);
   if (getsockname(s, (struct sockaddr *) &server, &namelen) < 0)
   {
       tcperror("getsockname()");
       exit(3);
   }
   printf("Port assigned is %d\n", ntohs(server.sin_port));
   /*
    * Receive a message on socket s in buf  of maximum size 32
    * from a client. Because the last two paramters
    * are not null, the name of the client will be placed into the
    * client data structure and the size of the client address will
    * be placed into client_address_size.
    */
   client_address_size = sizeof(client);
   if(recvfrom(s, buf, sizeof(buf), 0, (struct sockaddr *) &client,
            &client_address_size) <0)
   {
       tcperror("recvfrom()");
       exit(4);
   }
   /*
    * Print the message and the name of the client.
    * The domain should be the internet domain (AF_INET).
    * The port is received in network byte order, so we translate it to
    * host byte order before printing it.
    * The internet address is received as 32 bits in network byte order
    * so we use a utility that converts it to a string printed in
    * dotted decimal format for readability.
    */
   printf("Received message %s from domain %s port %d internet\
 address %s\n",
       buf,
       (client.sin_family == AF_INET?"AF_INET":"UNKNOWN"),
       ntohs(client.sin_port),
       inet_ntoa(client.sin_addr));
   /*
    * Deallocate the socket.
    */
   close(s);
}
```

C Socket UDP Client (UDPC)

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

```
/*** IBMCOPYR ********************************************************/
/*                                                                   */
/* Component Name: UDPC                                              */
/*                                                                   */
/* Copyright:                                                        */
/*   Licensed Materials - Property of IBM                            */
/*   This product contains "Restricted Materials of IBM"             */
/*   5735-FAL (C) Copyright IBM Corp. 1992.                          */
/*   5655-HAL (C) Copyright IBM Corp. 1992, 1996.                    */
/*   All rights reserved.                                            */
/*   US Government Users Restricted Rights -                         */
/*   Use, duplication or disclosure restricted by GSA ADP Schedule   */
/*   Contract with IBM Corp.                                         */
/*   See IBM Copyright Instructions.                                 */
/*                                                                   */
/*  TCP/IP for MVS                                                   */
/*  SMP/E Distribution Name: EZAEC020                                */
/*                                                                   */
/*** IBMCOPYR ********************************************************/
static char ibmcopyr[] =
   "UPDC  - Licensed Materials - Property of IBM. "
   "This module is \"Restricted Materials of IBM\" "
   "5735-FAL (C) Copyright IBM Corp. 1992. "
   "5655-HAL (C) Copyright IBM Corp. 1992, 1996. "
   "See IBM Copyright Instructions.";
#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>
#include <socket.h>
#include <netdb.h>
#include <stdio.h>
main(argc, argv)
int argc;
char **argv;
{
   int s;
   unsigned short port;
   struct sockaddr_in server;
   char buf[32];
   /* argv[1] is internet address of server argv[2] is port of server.
    * Convert the port from ascii to integer and then from host byte
    * order to network byte order.
    */
   if(argc != 3)
   {
      printf("Usage: %s <host address> <port> \n",argv[0]);
      exit(1);
   }
   port = htons(atoi(argv[2]));
   /* Create a datagram socket in the internet domain and use the
    * default protocol (UDP).
    */
   if ((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
   {
       tcperror("socket()");
       exit(1);
   }
   /* Set up the server name */
   server.sin_family      = AF_INET;            /* Internet Domain    */
   server.sin_port        = port;               /* Server Port        */
   server.sin_addr.s_addr = inet_addr(argv[1]); /* Server's Address   */
   strcpy(buf, "Hello");
   /* Send the message in buf to the server */
   if (sendto(s, buf, (strlen(buf)+1), 0,
                 (struct sockaddr *)&server, sizeof(server)) < 0)
   {
       tcperror("sendto()");
       exit(2);
   }
   /* Deallocate the socket */
   close(s);
}
```

Chapter 9. Using the X/Open Transport Interface (XTI)

This chapter describes the XTI application program interface (API) and contains the following topics:

- Software requirements
- What is provided
- How XTI works in the OS/390 environment
- Creating an application
- Coding XTI calls
- Compiling and linking XTI applications using cataloged procedures
- Understanding XTI sample programs

The XTI allows you to write applications in the OS/390 environment to access the open transport interface.

Note: The XTI calls in this chapter apply only to unconnected sessions.

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

For more information about sockets, see *UNIX Programmer's Reference Manual*.

Software Requirements

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

- *hlq*.SEZACMAC (macro library routines)
- *hlq*.SEZACMTX (executable modules)
- *hlq*.SEZALINK (executable modules)
- *hlq*.SEZAINST (sample programs)
- Current OS/390 language environment run-time library

What is Provided

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

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

  For more information about RFC1006, see "Appendix D. Related Protocol Specifications (RFCs)" on page 567.

How XTI Works in the OS/390 Environment

The XTI is a network-transparent protocol. In the OS/390 environment, XTI system support is a set of application calls to create the XTI protocol, as requested by your application. The services request is communicated to the XTI transport system using the RFC1006 protocol mapper. RFC1006 translates messages to transport class zero service requests before passing them to the XTI.

Figure 38 is a high-level diagram to show how the XTI interface works in an OS/390 environment.

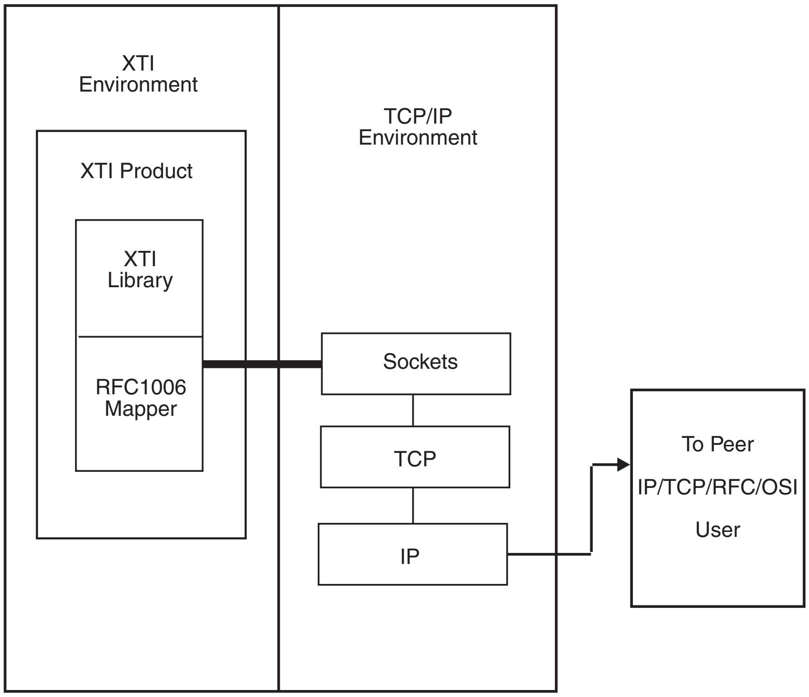


Figure 38. Using XTI with TCP/IP

In the OS/390 environment, external names must be eight characters or fewer. If the XTI application program interface names exceed this limit, those names longer than eight characters are remapped to new names using the C compiler preprocessor. This name remapping is found in a file called X11GLUE.H, which is automatically included in your program when you include the header file called XLIB.H. When debugging your application, you can refer to the X11GLUE.H file to find the remapped names of the XTI programs.

Creating an Application

To create an application that uses the XTI protocol, you should study the XTI application program interface in *CAE Specification: X/Open Transport Interface (XTI)*. In addition, both “XTI Socket Client Sample Program” on page 202, and “XTI Socket Server Sample Program” on page 208 illustrate programs that use the XTI interface. These programs are distributed with TCP/IP.

Coding XTI Calls

The following tables list the call instructions supported by the XTI for TCP/IP. These call instructions are for unconnected sessions only, and are listed by type of service.

Initializing a Transport Endpoint

Table 8 on page 199 lists the routines needed to initialize a transport endpoint. For more information see *CAE Specification: X/Open Transport Interface (XTI)*.

Table 8. Initializing a Call

Call	Description
t_bind()	Finds the endpoint for an address, and activates the endpoint.
t_open()	Creates a transport endpoint, and identifies the transport provided.

Establishing a Connection

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

Table 9. Establishing a Connection

Call	Description
t_accept()	Accepts a connection after a connect indication is received.
t_connect()	Requests connection to a transport user at a known destination.
t_listen()	Listens for connect information from other transport users.
t_rcvconnect()	Checks the status of a completed connect.

Transferring Data

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

Table 10. Transferring Data

Routine	Description
t_rcv()	Receives normal or expedited data over a transport connection.
t_snd()	Sends normal or expedited data over a transport connection.

Releasing a Connection

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

Table 11. Releasing a Connection

Call	Description
t_rcvdis()	Determines the reason for an abortive release or connection reject.
t_snddis()	Sends an abortive release or a connection reject.

Disabling a Connection

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

Table 12. Disabling a Connection

Call	Description
t_close()	Informs the XTI manager that you have finished with the endpoint, and frees any locally allocated resources assigned to endpoint.
t_unbind()	Resets the path to the transport endpoint. The connection is removed from the transport system, and requests for this path are denied.

Managing Events

Table 13 lists the routines needed to manage events. Each XTI call handles one event at a time. Events are processed one at a time, and you can wait on only one event at a time. For more information see *CAE Specification: X/Open Transport Interface (XTI)*.

Table 13. Managing Events

Call	Description
t_look()	Returns the events current for a transport endpoint, and notifies the calling program of an asynchronous event when the calling program is in synchronous mode.

Using Utility Calls

Table 14 lists utility routines that you can use to solve problems and monitor connections. For more information see *CAE Specification: X/Open Transport Interface (XTI)*.

Table 14. Using Utilities

Call	Description
t_error()	Returns the last error that occurred on a call to a transport function. You can add an identifying prefix to this call to aid in problem solving.
t_getinfo()	Returns information about the underlying transport protocol for the connection associated with file descriptor *fd*.
t_getstate()	Returns information about the state of the transport provider associated with file descriptor *fd*.

Using System Calls

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

Table 15. System Function Calls

Call	Description
fcntl()	Controls the operating characteristics of sockets. For more information, see “selectex()” on page 158.
select()	Checks descriptor sets to see if information is available for a read or a write. Select() also checks for pending exception conditions. For more information, see “select()” on page 154.
selectex()	Extends the select() calls by allowing you to add an ECB to define extra events. For more information, see “selectex()” on page 158.

Compiling and Linking XTI Applications Using Cataloged Procedures

Several methods are available to compile, link-edit, and run your XTI program. This section contains information about the data sets that you must include to run your XTI source program, using IBM-supplied cataloged procedures.

The following compile and link-edit sample procedures are supplied by IBM:
- XTICL is a sample compile and link-edit procedure.
- XTIC is a sample client execute procedure.
- XTIS is a sample server execute procedure.

```
//XTICL JOB XTICLJOB
⋮
//CCOMP PROC REG='3072K',
//           CPARM='DEF(MVS),SOURCE,LIST,NOMARG,SEQ(73,80)',
//           INSTLIB=,
//           SEZALNK=,
//           SEZAMAC=,
⋮
//*                                                                    *
//**********************************************************************
//*  COMPILE STEP:
//**********************************************************************
//*                                                                    *
//COMPILE EXEC PGM=EDCCOMP,
//        PARM=('&CPARM'),
//        REGION=&REG
//STEPLIB   DD DSN=&SEZALNK,DISP=SHR
//          DD DSN=&SEDCLNK,DISP=SHR
//          DD DSN=&IBMLINK,DISP=SHR
//          DD DSN=&SEDCOMP,DISP=SHR
//SYSLIB    DD DSN=&SEZAMAC,DISP=SHR
//          DD DSN=&CHEADRS,DISP=SHR,DCB=(BLKSIZE=3120)
//SYSIN     DD DSN=&INSTLIB(&INSTMEM),DISP=SHR
//SYSLIN    DD DSN=&OBJLIB(&INSTMEM),DISP=SHR
//SYSMSGS   DD DSN=&CMSGS,DISP=SHR
//SYSPRINT  DD SYSOUT=&SOUT
//SYSCPRT   DD SYSOUT=&SOUT
//SYSTERM   DD DUMMY
⋮
//*
//**********************************************************************
//* LINKEDIT STEP:
//**********************************************************************
//*
//LKED    EXEC PGM=IEWL,COND=(1,LT),
//          REGION=&REG
//OBJLIB    DD DSN=&OBJLIB,DISP=SHR
//SYSLIB    DD DSN=&SEZAMTX,DISP=SHR
//          DD DSN=&SEDCBSE,DISP=SHR
//          DD DSN=&IBMBASE,DISP=SHR
//SYSLMOD   DD DSN=&XTILOAD,DISP=SHR
//SYSPRINT  DD SYSOUT=&SOUT
//SYSUT1    DD DSN=&&SYSUT1,
//          UNIT=VIO,
//          DISP=(NEW,DELETE),
//          SPACE=(32000,(30,30))
// PEND
//*
//XTIC EXEC CCOMP,INSTMEM=XTICC
//LKED.SYSLIN DD *
       INCLUDE OBJLIB(XTICC)
       INCLUDE SYSLIB(XTI)
       MODE AMODE(31),RMODE(ANY)
       ENTRY CEESTART
       NAME XTIC(R)
//*
//XTIS EXEC CCOMP,INSTMEM=XTISC
//LKED.SYSLIN DD *
       INCLUDE OBJLIB(XTISC)
       INCLUDE SYSLIB(XTI)
       MODE AMODE(31),RMODE(ANY)
       ENTRY CEESTART
       NAME XTIS(R)
//*
```

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

Understanding XTI Sample Programs

This section contains sample XTI socket programs. The XTI source code can be found in the *hlq*.SEZAINST data set.

Note: As with all TCP/IP applications, dynamic data set allocations are used unless explicitly overridden. For example, the tcpip.DATA file can be specified using the SYSTCPD DD JCL statement.

The following sample XTI socket programs are available:

Name When Shipped	Alias Name	Description
XTICC	EZAEC0YL	XTI socket client sample program
XTISC	EZAEC0YM	XTI socket server sample program

XTI Socket Client Sample Program

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

```
/*********************************************************************/
/* XTIC Sample : Client                                              */
/*                                                                   */
/* Function:                                                         */
/*                                                                   */
/*  1. Establishes an XTI endpoint  (Asynchronous mode)              */
/*  2. Sends a connection request to an XTI server                   */
/*  3. Receives the request                                          */
/*  4. Sends a block of data to the server                           */
/*  5. Receives a block of data from the server                      */
/*  6. Disconnects from the server                                   */
/*  7. Client stops                                                  */
/*                                                                   */
/* Command line:                                                     */
/*                                                                   */
/*  XTIC hostname                                                    */
/*                                                                   */
/*       hostname - name of the host that the server is running.     */
/*                                                                   */
/*********************************************************************/

#include "xti.h"
#include "xti1006.h"
#include "stdio.h"

/*
*  bind request structure for t_bind()
*/

struct t_bind req,ret;

/*
*  for client to make calls to server
*/

struct t_call scall,rcall;

/*
*  store fd returned on open()
*/
```

```
int fd;

int tot_received;

char *hostname;

/*
*  data buffer
*/

char buf[25];

int looking;

/*
*  flags returned from t_rcv()
*/

int rflags,sflags;

/*
*  transport provider for t_open()
*/

char tprov[1][8] =
     { "RFC1006" } ;

/*
*  args that are optional
*/

int args;

int pnum = 102;
char *port = "102";
char *ctsel = "client";
char *stsel = "server";
unsigned int rqlen = 0;
struct xti1006tsap tsap, tsapret;
void cleanup(int);
void form_addr_1006(struct xti1006tsap *,int, char *, char*, int, int);
/*
*  MAIN line program starts here !!!
*/

main(argc,argv)
int argc;
char *argv[];
{

  /*
  *  Check arguments. The host name is required.  Host name is the
  *  last parameter passed.  Port can be changed by passing it as the
  *  first parameter.
  */

  if ((argc > 3) | (argc < 2))  {
    fprintf(stderr,"Usage XTIC <port> <host>\n");
    exit(1);
  }

  if(argc==2)
    hostname = argv[1];
  else
  {
    hostname = argv[2];
    port = argv[1];
    pnum = (unsigned short) atoi(argv[1]);
  }

  /*
  *  assume normal data
  */

  sflags = 0;

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

  if ((fd = t_open(tprov[0],O_NONBLOCK,NULL)) < 0)
  {
    t_error("Error on t_open for FD");
    exit(t_errno);
  }

  /*
  *  compose req structure for t_bind() calls
  */

  /*
  *  length of tsap
  */

  req.qlen = 0;
  req.addr.len = sizeof(tsap);

  /*
  *  allocate the buffer to contain the
  *  port and tsel to bind server to
  */

  req.addr.buf = (char *)malloc(sizeof(tsap));

  /*
  *  fill address buffer with the address information
  */

  form_addr_1006((struct xti1006tsap *)req.addr.buf,pnum,NULL, \
                  ctsel,fd,-1);

  /*
  *  now that we're done composing the req,
  *  do the bind of fd to addr in req
  */

  if (t_bind(fd,&req,NULL) != 0)
  {
    t_error("ERROR ON BIND FOR FD");
    exit(t_errno);
  }

  /*
  *  compose call structure for t_connect() call
  */

  scall.addr.len = sizeof(tsap);
  scall.addr.buf = (char *)malloc(sizeof(tsap));

  /*
  *  fill address buffer with the address information
  */
```

```
form_addr_1006((struct xti1006tsap *)scall.addr.buf,-1,hostname, \
                stsel,fd,-1);

scall.opt.maxlen = 0;
scall.opt.len = 0;
scall.opt.buf = NULL;
scall.udata.len = 0;
scall.udata.buf = NULL;

rcall.addr.maxlen = sizeof(tsapret);
rcall.addr.buf = (char *)malloc(sizeof(tsapret));
rcall.opt.maxlen = 0;
rcall.udata.maxlen = 0;
rcall.udata.buf = NULL;
/*
*  issue connect request
*/

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

looking = 1;
while (looking)
{
  looking = t_look(fd);
  if (looking == T_CONNECT & looking > 0)
    looking = 0;
  else
    if (looking != 0)
    {
      t_error("ERROR ON LOOK");
      cleanup(fd);
      exit(t_errno);
    }
    else
      looking = 1;
}

/*
*  establish connection
*/

looking = 1;
while (looking)
  if (t_rcvconnect(fd,&rcall) == 0)
    looking = 0;
  else
    if (t_errno != TNODATA)
    {
      t_error("ERROR ON RCVCONNECT");
      cleanup(fd);
      exit(t_errno);
    }

/*
*  place message in buffer
*/

memset(buf,'B',25);

/*
*  send message to server
```

```
*/

looking = 1;
while (looking)
  if ((looking = t_snd(fd,buf,sizeof(buf),sflags)) < 0)
  {
    t_error("ERROR SENDING MESSAGE TO SERVER");
    cleanup(fd);
    exit(t_errno);
  }
  else
    if (looking == 0)
      looking = 1;
    else
      looking = 0;

/*
*  receive data back from the server
*/

looking = 1;
while (looking)
{
  if ((looking = t_rcv(fd,buf,sizeof(buf),&rflags)) > 0)
    looking = 0;
  else
    {
      if (looking < 0 & t_errno != TNODATA)
      {
         t_error("ERROR RECEIVING DATA FROM SERVER");
         cleanup(fd);
         exit(t_errno);
      }
      else
        looking = 1;
    }
}

/*
*  disconnect from server
*/

looking = 1;
while (looking)
  if (t_snddis(fd,NULL) == 0)
    looking = 0;
  else
  {
    t_error("ERROR DISCONNECTING FROM SERVER");
    cleanup(fd);
    exit(t_errno);
  }

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

if (t_unbind(fd) != 0)
{
  t_error("ERROR ON BIND FOR FD");
  exit(t_errno);
}

cleanup(fd);
```

```
  printf("Client ended successfully\n");
  exit(0);

}
/*********************************************************************/

void form_addr_1006(addrbuf1006,portnum,hostnmstr,tselstr1006,fd1,fd2)

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

/*
*  address buffer to be filled in
*/

struct xti1006tsap *addrbuf1006;

int   portnum;

/*
*  hostnmstr represented as a string
*/

char            *hostnmstr;

/*
*  tsel represented as a string
*/

char            *tselstr1006;

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

int   fd1;

/*
*  other possible endpoint to close
*/

int   fd2;

{


  /*
  *  check validity of hostname
  *  there's no way program can
  *  continue without valid addr
  */

  if (strlen(hostnmstr) > 64)
  {
     fprintf(stderr,"hostname %s too long\n",hostnmstr);
     /*
     *  don't want TADDRBUSY when you try to reuse the address
     */
     cleanup(fd1);
     cleanup(fd2);
     exit(TBADADDR);
   }

  addrbuf1006->xti1006_hostnm_len = strlen(hostnmstr);
  strcpy(addrbuf1006->xti1006_hostnm,hostnmstr);

  /*
  *  check validity of hostname
  *  there's no way program can
  *  continue without valid addr
  */

  if (strlen(tselstr1006) > 64)
  {
     fprintf(stderr,"tsel %s too long\n",tselstr1006);
     /*
     *  don't want TADDRBUSY when you try to reuse the address
     */
     cleanup(fd1);
     cleanup(fd2);
     exit(TBADADDR);
  }

  addrbuf1006->xti1006_tsel_len = strlen(tselstr1006);
  strcpy(addrbuf1006->xti1006_tsel,tselstr1006);

  if (tselstr1006 == "Nulltsap")
  {
     addrbuf1006->xti1006_tsel_len = 0;
     strcpy(addrbuf1006->xti1006_tsel,NULL);
  }
  else
  {
     addrbuf1006->xti1006_tsel_len = strlen(tselstr1006);
     strcpy(addrbuf1006->xti1006_tsel,tselstr1006);
  } /* endif */

  if (portnum != -1)
     addrbuf1006->xti1006_tset = portnum;

}
/*******************************************************************/

void cleanup(fd)

int fd;

{
  if (fd >= 0)
    if (t_close(fd) != 0)
    {
       fprintf(stderr,"unable to t_close() endpoint while");
       fprintf(stderr," cleaning up from error\n");
    }
}
```

XTI Socket Server Sample Program

As with all TCP/IP applications, dynamic dataset allocations are used unless explicitly overridden. For example, the tcpip.DATA file can be specified using the SYSTCPD DD JCL statement. For more information, see “Chapter 8. C Socket Application Programming Interface (API)” on page 73.

The following is an example of an XTI socket server program.

```
/*******************************************************************/
/* XTIS Sample : Server                                            */
/*                                                                 */
/* Function:                                                       */
/*                                                                 */
/*  1. Establishes an XTI endpoint  (Asynchronous mode)            */
/*  2. Listens for a connection request from an XTI client           */
/*  3. Accepts the connection request                                */
/*  4. Receives a block of data from the client                      */
/*  5. Echos the data back to the client                             */
/*  6. Waits for the disconnect request from the XTI client          */
/*  7. Server stops                                                  */
/*                                                                   */
/* Command line:                                                     */
/*                                                                   */
/*  XTIS     H                                                       */
/*                                                                   */
/*********************************************************************/
#include "xti.h"
#include "xti1006.h"
#include "stdio.h"

/*
*  bind request structure for t_bind()
*/

struct t_bind req,ret;

/*
*  for server to listen for calls with
*/

struct t_call call;

/*
*  descriptor to t_listen() on
*/

int fd;

/*
*  descriptor to t_accept() on
*/

int resfd;

int tot_received;

/*
*  data buffer
*/

char buf[25];

int looking;

/*
*  flags returned from t_rcv()
*/

int rflags,sflags;

/*
*  transport provider for t_open()
*/

char tprov[1][8] =
     { "RFC1006" } ;

/*
*  args that are optional
*/
```

```
int args;

int tot_sent;
int pnum = 102;
char *port = "102";
char *hostnm;
char *stsel = "server";
unsigned int rqlen = 0;
struct xti1006tsap tsap, tsapret;
void cleanup(int);
void form_addr_1006(struct xti1006tsap *,int, char *, char*, int, int);

/*
*  MAIN line program starts here !!!
*/

main(argc,argv)
int argc;
char *argv[];
{

  /*
  *  Check arguments. No arguments should be passed to the server
  */

  if (argc > 2) {
    fprintf(stderr,"Usage : XTIS <port>\n");
    exit(1);
  }

  if(argc == 2)
  {
    pnum = (unsigned short) atoi(argv[1]);
    port = argv[1];
  }
  /*
  *  assume normal data
  */

  sflags = 0;

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

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

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

  if ((resfd = t_open(tprov[0],O_NONBLOCK,NULL)) < 0)
  {
    t_error("Error on t_open");
    cleanup(fd);
    exit(t_errno);
  }
  /*
  *  compose req structure for t_bind() calls
  */

  /*
  *  length of tsap
```

```
*/

req.addr.len = sizeof(tsap);

/*
*  allocate the buffer to contain the
*  port and tsel to bind server to
*/

req.addr.buf = (char *)malloc(sizeof(tsap));

/*
*  fill address buffer with the address information
*/

form_addr_1006((struct xti1006tsap *)req.addr.buf, \
                pnum,                              \
                NULL,                              \
                stsel,                             \
                fd,                                \
                resfd);

/*
*  length of tsap
*/

ret.addr.maxlen = sizeof(tsapret);
ret.addr.buf = (char *)malloc(sizeof(tsapret));

/*
*  listening endpoint needs qlen > 0,
*  ability to queue 10 requests
*/

req.qlen = 10;
ret.qlen = rqlen;

/*
*  now that we're done composing the req,
*  do the bind of fd to addr in req
*/

if (t_bind(fd,&req,&ret) != 0)
{
  t_error("Error on t_bind");
  cleanup(fd);
  cleanup(resfd);
  exit(t_errno);
}

/*
*  accepting endpoint with same addr needs qlen == 0
*/

req.qlen = 0;

/*
*  now that we're done composing the req,
*  do the bind of resfd to addr in req
*/

if (t_bind(resfd,&req,&ret) != 0)
{
  t_error("Error on t_bind");
  cleanup(fd);
  cleanup(resfd);
  exit(t_errno);
}

/*
*  initialize call receipt structure for t_listen()
*/

call.opt.maxlen = 0;
call.addr.len = 0;
call.opt.len = 0;
call.udata.len = 0;
call.opt.buf = NULL;

call.addr.maxlen = sizeof(tsapret); /* listen for return*/
call.addr.buf = (char *)malloc(sizeof(tsapret));

call.udata.maxlen = 0;
call.udata.buf = NULL;

/*
*  wait for connect req & get seq num in the call variable
*/

looking = 1;
while (looking)
  if (t_listen(fd,&call) == 0)
    looking = 0;
  else
    if (t_errno != TNODATA)
    {
      t_error("Error on t_accept");
      cleanup(fd);
      cleanup(resfd);
      exit(t_errno);
    }

/*
*  accept the connection on the accepting endpoint
*/

if (t_accept(fd,resfd,&call) != 0)
{
  t_error("Error on t_accept");
  cleanup(fd);
  cleanup(resfd);
  exit(t_errno);
}
/*
*  receive data from the client
*/

looking = 1;
while (looking)
  if (t_rcv(resfd,buf,sizeof(buf),&rflags) > 0)
    looking = 0;
  else
    if (t_errno != TNODATA)
    {
      t_error("Error on t_rcv");
      cleanup(fd);
      cleanup(resfd);
      exit(t_errno);
    }

/*
*  sent data back to the client
*/
```

```
  strcpy(buf,"DATA FROM SERVER");

  looking = 1;
  while (looking)
    if (t_snd(resfd,buf,sizeof(buf),sflags) > 0)
      looking = 0;

  /*
  *  wait for disconnect from the client
  */

  looking = 1;
  while (looking)
    if (t_look(resfd) == T_DISCONNECT)
      looking = 0;

  /*
  *  receive the disconnect request
  */

  looking = 1;
  while (looking)
    if (t_rcvdis(resfd,NULL) == 0)
      looking = 0;

  /*
  *  unbind the endpoints
  */

  if (t_unbind(resfd) != 0)
  {
    t_error("Error on t_unbind for resfd");
    cleanup(fd);
    cleanup(resfd);
    exit(t_errno);
  }

  if (t_unbind(fd) != 0)
  {
    t_error("Error on t_unbind for fd");
    cleanup(fd);
    cleanup(resfd);
    exit(t_errno);
  }

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

  cleanup(fd);

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

  cleanup(resfd);

  printf("Server ended successfully\n");
  exit(0);

}
/*********************************************************************/

void form_addr_1006(addrbuf1006,portnum,hostnmstr,tselstr1006,fd1,fd2)

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

/*
*  address buffer to be filled in
*/

struct xti1006tsap *addrbuf1006;

int   portnum;

/*
*  hostnmstr represented as a string
*/

char            *hostnmstr;

/*
*  tsel represented as a string
*/

char            *tselstr1006;

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

int   fd1;

/*
*  other possible endpoint to close
*/

int   fd2;

{


  /*
  *  check validity of hostname
  *  there's no way program can
  *  continue without valid addr
  */

  if (strlen(hostnmstr) > 64)
  {
     fprintf(stderr,"hostname %s too long\n",hostnmstr);
     /*
     *  don't want TADDRBUSY when you try to reuse the address
     */
     cleanup(fd1);
     cleanup(fd2);
     exit(TBADADDR);
  }

  addrbuf1006->xti1006_hostnm_len = strlen(hostnmstr);
  strcpy(addrbuf1006->xti1006_hostnm,hostnmstr);

  /*
  *  check validity of hostname
  *  there's no way program can
  *  continue without valid addr
  */

  if (strlen(tselstr1006) > 64)
  {
     fprintf(stderr,"tsel %s too long\n",tselstr1006);
     /*
     *  don't want TADDRBUSY when you try to reuse the address
     */
     cleanup(fd1);
     cleanup(fd2);
     exit(TBADADDR);
  }

  addrbuf1006->xti1006_tsel_len = strlen(tselstr1006);
  strcpy(addrbuf1006->xti1006_tsel,tselstr1006);

  if (tselstr1006 == "Nulltsap")
  {
     addrbuf1006->xti1006_tsel_len = 0;
     strcpy(addrbuf1006->xti1006_tsel,NULL);
  }
  else
  {
     addrbuf1006->xti1006_tsel_len = strlen(tselstr1006);
     strcpy(addrbuf1006->xti1006_tsel,tselstr1006);
  } /* endif */

  if (portnum != -1)
     addrbuf1006->xti1006_tset = portnum;

}
/*********************************************************************/

void cleanup(fd)

int fd;

{
  if (fd >= 0)
    if (t_close(fd) != 0)
    {
       fprintf(stderr,"unable to t_close() endpoint while");
       fprintf(stderr," cleaning up from error\n");
     }
}
```

Chapter 10. Using the Macro Application Programming Interface (API)

This chapter describes the macro API for application programs written in System/390 assembler language.

The macro interface can be used to produce reentrant modules and can be used in a multithread environment.

The following topics are included:

- Environmental restrictions and programming requirements
- Defining storage for the API macro
- Understanding common parameter descriptions
- Characteristics of stream sockets
- Task management and asynchronous function processing
- Using an unsolicited event exit routine
- Error messages and return codes
- Diagnosing problems in applications using the macro API
- Macros for assembler programs
- Macro interface assembler language sample programs

Environmental Restrictions and Programming Requirements

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

- SRB mode

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

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

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

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

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

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

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

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

- Use of UNIX System Services

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

Defining Storage for the API Macro

The macro API requires both global and task storage areas.

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

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

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

The second storage area is a task storage area that must be known to and addressable by all socket users communicating across a specified connection. A connection runs between the application and TCP/IP. The most common way to organize storage is to assign one connection to each MVS subtask. If there are multiple modules using sockets within a single task or connection, you must provide the address of the task storage to every user.

The following describes how to define the address of the task storage:

- Code the instruction EZASMI TYPE=TASK with STORAGE=CSECT as part of the program code. This makes the program nonreentrant, but simplifies the code.
- Code the instruction EZASMI TYPE=TASK with STORAGE=DSECT as part of the program code. The expansion of this instruction generates the equate field, TIELENTH, which is equal to the length of the storage area. This can be used to issue an MVS GETMAIN to allocate the required storage.

The defining program must make the address of this storage available to all other programs using this connection. Programs running in these tasks must define the storage mapping with an EZASMI TYPE=TASK with STORAGE=DSECT.

The EZASMI TYPE=TASK macro generates only one parameter list for a connection. This can lead to overlay problems for programs using APITYPE=3 connections (multiple calls can be issued simultaneously). For more detail on APITYPE=3 connections, see “Task Management and Asynchronous Function

Processing” on page 220. A program should use the following format to build unique parameter list storage areas if it will be issuing mulitple calls simultaneously on one connection:

```
BINDPRML   EZASMI    MF=L  This will generate the storage used for
                      building the parm list in the following BIND call
      EZASMI    TYPE=BIND,
               S=SOCKDESC,
               NAME=NAMEID
               ERRNO=ERRNO,
               RETCODE=RETCODE,
               ECB=ECB1
               MF=(E,BINDPRML)
```

This example of an asynchronous BIND macro would use the MF=L macro to generate the parameter list. The fields that are common across all macro calls, for example, RETCODE and ERRNO, must be unique for each outstanding call.

You can create multiple connections to TCP/IP from a single task. Each of these connections functions independently of the other and is identified by its own task interface element (TIE). The TASK parameter can be used to explicitly reference a TIE. If you do not include the TASK parameter, the macro uses the TIE generated by the EZASMI TYPE=TASK macro.

```
TIE1    DS XL(TIELENTH)         Length of TIE

EZASMI TYPE=INITAPI,
       MAXSOC=MAX75,
        ERRNO=ERRNO
        RETCODE=RETCODE,
        APITYPE=2,
        MAXSNO=MAXS,
        TASK=TIE1

  EZASMI  TYPE=SOCKET,
           AF='INET',
           SOCTYPE='STREAM',
           ERRNO=ERRNO,
           RETCODE=RETCODE,
           TASK=TIE1
```

In this example, the TIE TIE1 is used for the connection, not the TIE generated by the EZASMI TYPE=TASK macro.

Understanding Common Parameter Descriptions

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

Parameter **Description**

address
The name of the field that contains the value of the parameter. The following example illustrates a BIND macro where SOCKNO is set to 2.

```
MVC   SOCKNO,=H'2'
EZASMI TYPE=BIND,S=SOCKNO
```

**indaddr*
The name of the address field that contains the address of the field containing the parameter. The following example produces the same result as the example above.

```
MVC   SOCKNO,=H'2'
LA    0,SOCKNO
ST    0,SOCKADD
EZASMI TYPE=BIND,S=*SOCKADD
```

(reg) The name (equated to a number) or the number of a general purpose register. Do not use a register 0, 1, 14, or 15. The following example produces the same result as the previous examples.

```
MVC   SOCKNO,=H'2'
LA    3,SOCKNO
EZASMI TYPE=BIND,SOCKNO=(3)
```

'value' A literal value for the parameter; for example, AF='INET'

Characteristics of Stream Sockets

For stream sockets, data is processed as streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and application A sends 1000 bytes, each call to the SEND function can return one byte, ten bytes, or the entire 1000 bytes, with the number of bytes sent returned in the RETCODE call. Therefore, applications using stream sockets should place the READ call and the SEND call in a loop that repeats until all of the data has been sent or received.

Task Management and Asynchronous Function Processing

The sockets extended interface allows asynchronous operation, although by default the task issuing a macro request is put into a WAIT state until the requested function complete. At that time, the issuing task resumes and continues execution.

If you do not want the issuing task to be placed into a WAIT while its request is processed, use asynchronous function processing.

How It Works

The macro API provides for asynchronous function processing in two forms. Both forms cause the system to return control to the application immediately after the function request has been sent to TCP/IP. The difference between the two forms is in how the application is notified when the function is completed:

ECB method
Enables you to pass an MVS event control block (ECB) on each socket call. The socket library returns control to the program immediately and posts the ECB when the call has completed.

EXIT method
Enables you to specify the entry point of an exit routine using the INITAPI() call. The individual socket calls immediately return control to the program and the socket library drives the specified exit routine when the socket call is complete.

In either case, the function is completed when the notification is delivered. Note that the notification may be delivered at any time, in some cases even before the application has received control back from the EZASMI macro call. It is therefore important that the application is ready to handle a notification as soon as it issues the EZASMI macro call.

Like nonblocking calls, asynchronous calls return control to your program immediately. But in this case, there is no need to reissue the call. When the requested event has taken place, an ECB is posted or an exit routine is driven.

Using the API macro, you can specify APITYPE=2 or APITYPE=3

APITYPE=2 Allows an asynchronous macro API program to have only one outstanding socket call per socket descriptor. An APITYPE=2 program can use macro API asynchronous calls, but synchronous calls are equally well supported.

APITYPE=3 Allows an asynchronous macro API program to have many outstanding socket calls per socket descriptor. Only the macro API supports APITYPE=3. An APITYPE=3 program must use macro API asynchronous calls with either an ECB or REQAREA parameter.

The REQAREA parameter is used in macros using the EXIT form. This parameter is mutually exclusive with the ECB parameter used with the ECB form.

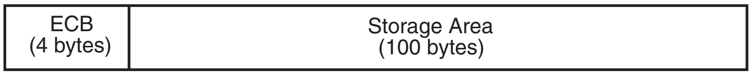


Figure 39. ECB Input Parameter

Like the ECB parameter, the REQAREA parameter points to an area that contains:

- A four-byte token that is presented to your asynchronous exit routine when the response to this function request is complete.
- A 100-byte storage area that is used by the interface to save the state information.

Note: This storage must not be modified or freed until the macro function has completed and the ECB has been posted or the asynchronous exit has been driven.

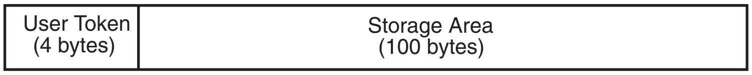


Figure 40. User Token Setting

Before you issue the macro, you must set the first word of the 104 bytes to a token of any value. The token is used by your asynchronous exit routine to determine the function completion event for which it is being invoked.

Asynchronous functions are processed in the following sequence:

1. The application must issue the EZASMI TYPE=INITAPI with ASYNC='ECB' or ASYNC=('EXIT', AEEXIT). The ASYNC parameter notifies the API that asynchronous processing is to be used for this connection. The API notes the type of asynchronous processing to be used, ECB or EXIT, and specifies the use of the asynchronous exit routine for this connection.
2. When a function request is issued by the application, the API does one of the following:

- If the type of asynchronous processing is ECB, and an ECB is supplied in the function request, the API returns control to the application. If Register 15 is zero, the ECB will be posted when the function has completed. Note that the ECB may be posted prior to when control is returned to the application.
- If the type of asynchronous processing is EXIT, and a REQAREA parameter is supplied in the function request, the API returns control to the application. If Register 15 is zero, the exit routine is invoked when the function has completed. Note that the exit can be invoked prior to when control is returned to the application.

In either case, Register 15 is used to inform the caller whether or not the ECB will be posted or asynchronous exit driven. Therefore, you must not use Register 15 for the RETCODE parameter.

When the asynchronous exit routine is invoked, the following linkage conventions are used:

GPR0 Register Setting

0 Normal return

1 TCP/IP address space has terminated (TCPEND)

2 TCP/IP has terminated the connection to the user (TCPCONEND).

3 An error has been detected in the asynchronous exit routine. Perform applicable recovery processing.

GPR1 Points to a doubleword field containing the following:

WORD1
The token specified by the INITAPI macro.

WORD2
The token specified by the functional request macro. (First four bytes of the REQAREA storage.)

GPR13
Points to standard MVS save area in the same key as the application PSW at the time of the INITAPI command.

GPR14
Return address.

GPR15
Entry point of the exit routine.

The following example shows how to code an asynchronous macro function:

```
EZASMI TYPE=READ,     READ A BUFFER OF DATA FROM THE
      S=SOCKNO,       CONNECTION PEER.  I MAY NEED TO
      NBYTES=COUNT,   WAIT SO GIVE CONTROL BACK TO ME
      BUF=DATABUF,    AND LET ME ISSUE MY OWN WAIT.
      ERRNO=ERROR,    IT COULD BE PART OF A WAIT WHICH
      RETCODE=RCODE,  WOULD INCLUDE OTHER EVENTS.
      ECB=MYECB,      SPECIFY ECB/STORAGE AREA FOR INTERFACE
      ERROR=ERRORRTN

LTR    R15,R15        Was macro function passed to TCP/IP?
BNZ    BADRCODE       If no, ECB will not be posted
WAIT   ECB=MYECB      TELL MVS TO WAIT UNTIL READ IS DONE
```

Asynchronous Exit Environmental and Programming Considerations

When utilizing the ASYNC=EXIT option of the EZASMI macro the following requirements need to be considered:

- Asynchronous calls can only be issued from a single request block (RB) in a given task (TCB).

  The first RB that issues an ASYNC EZASMI call under a given task is deemed as the target RB that will be interrupted when an asynchronous exit needs to be driven. This means that after an asynchronous EZASMI macro call is invoked you should not invoke any services that cause the current RB to no longer be the top RB for this task (for example, a LINK call). If the target RB is no longer the top RB at the time that the exit needs to be driven then the exit will be deferred until the target RB becomes the top RB. One exception to this rule is that EZASMI calls may be issued under the asynchronous user exit (see next bullet).
- EZASMI macro calls within the asynchronous exits

  While running the asynchronous exit notification routine an application may issue other EZASMI calls, However, the application should avoid issuing any blocking calls and should not enter into long delays. Doing so will delay any additional exits from being driven and also blocks the TCB that made the original call. Note that TERMAPI should not be issued under the asynchronous exit.
- Linkage stack

  Applications issuing EZASMI macro asynchronous exit calls should not issue any PC instructions that cause the system linkage to be used. Doing so will delay the asynchronous exits from being driven until the linkage stack entry is removed. If the linkage stack entry is not removed, the exit will not be driven.
- Asynchronous exits are given control in the same key as the program status word (PSW) key of the TCB from which the EZASMI call was issued.

Using an Unsolicited Event Exit Routine

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

The user invokes this facility by the keyword UEEXIT parameter used in the EZASMI TYPE=INITAPI macro, as shown:

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

Keyword **Description**

UEEXIT Includes two positional parameters. The first is the address of an unsolicited event exit routine to be invoked when an unsolicited event takes place. The second is a token passed to the unsolicited exit routine at invocation. On entry to the unsolicited event exit, the general purpose registers (GPRs) contain:

GPR0 Register setting

1 TCP/IP address space has terminated (TCPEND).

2 TCP/IP has terminated the connection to the user (TCPCONEND).

4 Other. Call the IBM support center.

GPR1 Address of the token specified in the INITAPI macro.

GPR13
Points to standard MVS save area in the same key as the application PSW at the time of the INITAPI command.

GPR14
Return address.

Error Messages and Return Codes

For information about error messages, see *TCP/IP for MVS: Messages and Codes*.

For information about codes returned by TCP/IP, see “Appendix B. Return Codes” on page 547.

Diagnosing Problems in Applications Using the Macro API

TCP/IP provides a trace facility that can be helpful in diagnosing problems in applications using the Macro API. The trace is implemented using the TCP/IP Component Trace (CTRACE) SOCKAPI trace option. The SOCKAPI trace option allows all Macro socket API calls issued by an application to be traced in the TCP/IP CTRACE. The SOCKAPI trace records include information such as the type of socket call, input, and output parameters and return codes. This trace can be helpful in isolating failing socket API calls and in determining the nature of the error or the history of socket API calls that may the cause of an error. For more information on the SOCKAPI trace option, refer to *OS/390 IBM Communications Server: IP Diagnosis*.

Macros for Assembler Programs

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

ACCEPT

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

When issued, the ACCEPT macro:

1. Accepts the first connection on a queue of pending connections
2. Creates a new socket with the same properties as the socket used in the macro and returns the address of the client for use by subsequent server macros. The new socket cannot be used to accept new connections, but can be used by the calling program for its own connection. The original socket remains available to the calling program for more connection requests.
3. Returns the new socket descriptor to the calling program.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

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

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

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

Keyword **Description**

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

NAME
Output parameter. Initially, the application provides a pointer to the socket address structure; this structure is filled on completion of the call with the socket address of the connection peer.

Field **Description**

FAMILY
A halfword binary field specifying the addressing family. For TCP/IP the value is always 2, indicating AF_INET.

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

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

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

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

See "Appendix B. Return Codes" on page 547, for information about **ERRNO** return codes.

RETCODE Output parameter. If **RETCODE** is positive, **RETCODE** is the new socket number.

If **RETCODE** is negative, check **ERRNO** for an error number.

NS Input parameter. A value or the address of a halfword binary number specifying the descriptor number chosen for the new socket, which is the socket for the client at the time. If NS is not specified, the interface assigns it.

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

BIND

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

The BIND macro can specify the port or let the system choose the port. A listener program should always bind to the same well-known port so that clients know the socket address to use when issuing a CONNECT macro.

In the AF_INET domain, the BIND macro for a stream socket can specify the networks from which it is willing to accept connection requests. Your application can select the network interface by setting **ADDRESS** to the internet address of the network from which you want to accept connection requests. Alternatively, your application can accept connection requests from any network if you set the address field to a fullword of zeros.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

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

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

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

Keyword **Description**

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

NAME
Input parameter. The application provides a pointer to the socket

address structure, from which it specifies the port number and IP address the application can accept connections.

Field Description

FAMILY
A halfword binary field specifying the addressing family. For TCP/IP the value is always 2, indicating AF_INET.

PORT A halfword binary field set to the port number that will bind to the socket. If you set the port number to zero, TCP/IP assigns the port. The application can call the GETSOCKNAME macro after the BIND macro to discover the assigned port number.

IP-ADDRESS
A fullword binary field that is set to the 32-bit internet address, in network byte order, of the client host machine. If zero is specified, the application accepts connections from any network address.

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

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

See “Appendix B. Return Codes” on page 547, for information about **ERRNO** return codes.

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

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

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

CANCEL

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

```
►►──EZASMI──TYPE=CANCEL──────────────────────────────────────────────►

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

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

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

Keyword **Description**

CALAREA Input parameter. The **ECB** or **REQAREA** specified in the call being cancelled.

Note: To be compatible with TCP/IP for MVS V3R1, **CALAREA** can be specified as **CALLAREA**.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA

A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

ERRNO

Output parameter. A fullword binary field. If **RETCODE** is negative, this contains an error number.

RETCODE

Output parameter. A fullword binary field. If **RETCODE** is 0, the CANCEL was successful.

ERROR

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

TASK

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

CLOSE

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

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

Note: If a stream socket is closed while input or output data is queued, the stream connection is reset and data transmission can be incomplete. SETSOCKET can be used to set a SO_LINGER condition, in which TCP/IP continues to send data for a specified period of time after the CLOSE macro is issued. For information about SO_LINGER, see “SETSOCKOPT” on page 393.

The following requirements apply to this call:

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

Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

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

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

Keyword **Description**

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

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

See “Appendix B. Return Codes” on page 547, for information about **ERRNO** return codes.

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

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

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

CONNECT

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

For stream sockets, the CONNECT macro:

- Completes the binding process for a stream socket if BIND has not been previously issued.
- Attempts connection to a remote socket. This connection must be completed before data can be transferred.

For datagram sockets, CONNECT is not essential, but you can use it to send messages without specifying the destination.

For both types of sockets, the following CONNECT macro sequence applies:

1. The server issues BIND and LISTEN (stream sockets only) to create a passive open socket.
2. The client issues CONNECT to request a connection.
3. The server creates a new connected socket by accepting the connection on the passive open socket.

If the socket is in blocking mode, CONNECT blocks the calling program until the connection is established, or until an error is received.

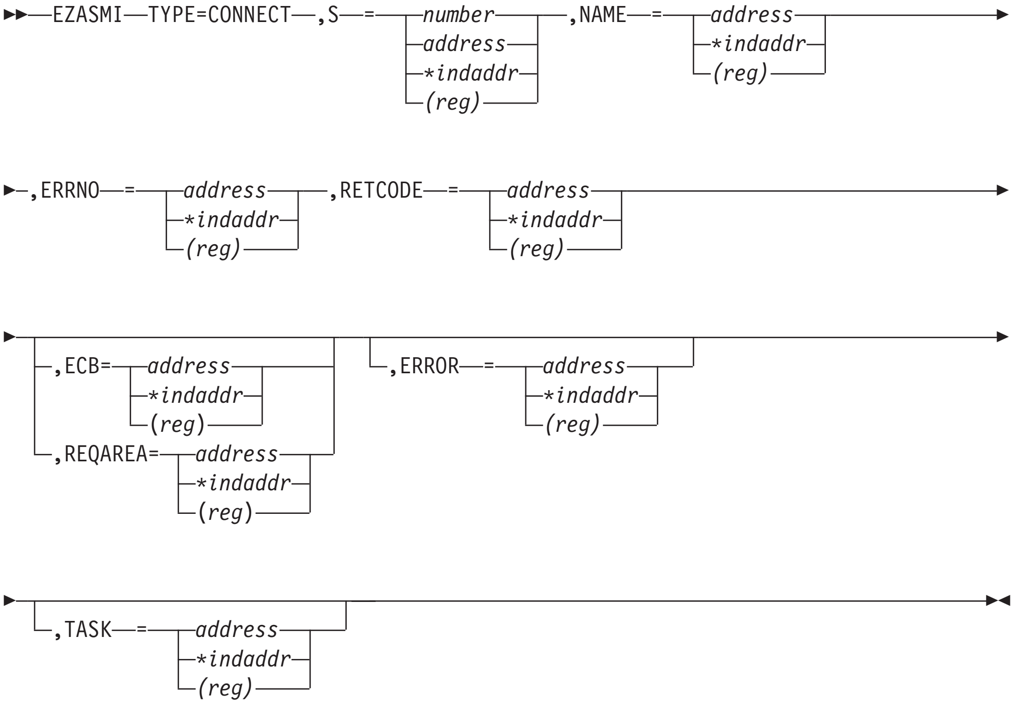
If the socket is in nonblocking mode, the return code indicates the success of the connection request.

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

The caller must test the completion of the connection setup by calling SELECT and testing for the ability to write to the socket. The completion cannot be checked by issuing a second CONNECT.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space



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

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

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

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

Keyword	Description
S	Input parameter. A value or the address of a halfword binary number specifying the socket descriptor.
NAME	Input parameter. The NAME parameter for CONNECT specifies the socket address of the connection peer.

Field **Description**

FAMILY
A halfword binary field specifying the addressing family. For TCP/IP the value is always 2, indicating AF_INET.

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

IP-ADDRESS
A fullword binary field specifying the 32-bit internet address of the server host machine in network byte order. For example, if the internet address is 129.4.5.12 in dotted decimal notation, it is set to X'8104050C'.

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

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

See “Appendix B. Return Codes” on page 547, for information about **ERRNO** return codes.

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

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

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

FCNTL

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

See “IOCTL” on page 363 for another way to control socket blocking.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

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

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

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

Keyword **Description**

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

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

Value **Description**

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

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

The FNDELAY flag sets the nonblocking mode for the socket. If data is not present on calls that can block (READ, READV, and RECV), the call returns a -1, and **ERRNO** is set to 35 (EWOULDBLOCK).

Note: Values for **COMMAND** that are supported by the OS/390 UNIX System Services FCNTL callable service are supported also. Refer to the *OS/390 UNIX System Services Programming: Assembler Callable Services Reference* for more information.

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

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

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

See “Appendix B. Return Codes” on page 547, for information about **ERRNO** return codes.

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

- If **COMMAND** was set to 3 (query), a bit string is returned.
  - If **RETCODE** contains X'00000004', the socket is nonblocking. The FNDELAY flag is on.
  - If **RETCODE** contains X'00000000', the socket is blocking. The FNDELAY flag is off.
- If the **COMMAND** field was 4 (set), a successful call returns zero in **COMMAND**. For either **COMMAND**, a **RETCODE** of -1 indicates an error. Check **ERRNO** for the error number.

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

GETCLIENTID

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

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

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

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

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

Keyword **Description**

CLIENT Output parameter. A client ID structure describing the identifier for your application, regardless whether a server or client.

Field **Description**

DOMAIN
Output parameter. A fullword binary number specifying the domain of the client. For TCP/IP the value is always 2, indicating AF_INET.

NAME Initially, the application provides a pointer to an eight-byte character field, which is filled, on completion of the call, with the client address space identifier.

TASK Output parameter. An eight-byte character field set to the client task identifier.

RESERVED
Output parameter. Specifies 20-bytes of binary zeros. This field is required, but it is not used.

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

See “Appendix B. Return Codes” on page 547, for information about **ERRNO** return codes.

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

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

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

GETHOSTBYADDR

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

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

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

Note: The storage for the HOSTENT structure returned by this call is released during TERMAPI processing; therefore, the application program must not use the HOSTENT storage after the TERMAPI.

Keyword **Description**

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

HOSTENT Input parameter. A fullword word containing the address of the

HOSTENT structure returned by the macro. For information about the HOSTENT structure, see Figure 41.

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

Value	**Description**
>0	Successful call
–1	An error occurred

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

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

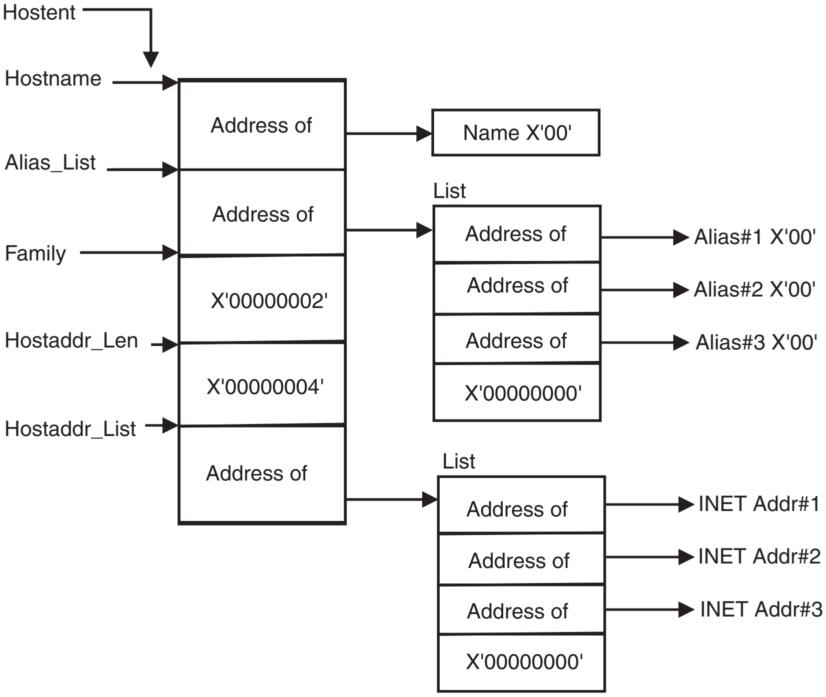


Figure 41. HOSTENT Structure Returned by the GETHOSTBYADDR Macro

GETHOSTBYADDR returns the HOSTENT structure shown in Figure 41. This structure contains:

- The address of the host name returned by the macro. The name length is variable and is ended by X'00'.
- The address of a list of addresses that point to the alias names returned by the GETHOSTBYADDR. This list is ended by the pointer X'00000000'. Each alias name is a variable length field ended by X'00'.
- The value returned in the FAMILY field is always 2 to signify AF_INET.
- The length of the host internet address returned in the HOSTADDR_LEN field is always 4 to signify AF_INET.
- The address of a list of addresses that point to the host internet addresses returned by the macro. The list is ended by the pointer X'00000000'.

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

GETHOSTBYNAME

The GETHOSTBYNAME macro returns the alias names and the internet addresses of a host whose domain name is specified in the macro.

TCP/IP tries to resolve the host name through a name server, if one is present. If a 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.

If the host name is not found, the return code is -1.

Important: Repeated use of GETHOSTBYNAME before calls which implicitly or explicitly invoke INITAPI can result in the allocation of unreleased storage.

HOSTS.LOCAL, HOSTS.ADDRINFO, and HOSTS.SITEINFO are described in *OS/390 IBM Communications Server: IP Configuration Reference*.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

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

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

Note: The storage for the HOSTENT structure returned by this call is released during TERMAPI processing; therefore, the application program must not use the HOSTENT storage after the TERMAPI.

Keyword **Description**

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

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

HOSTENT Output parameter. A fullword word containing the address of HOSTENT returned by the macro. For information about the HOSTENT structure, see Figure 42.

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

Value	Description
0	Successful call
–1	An error occurred

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

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

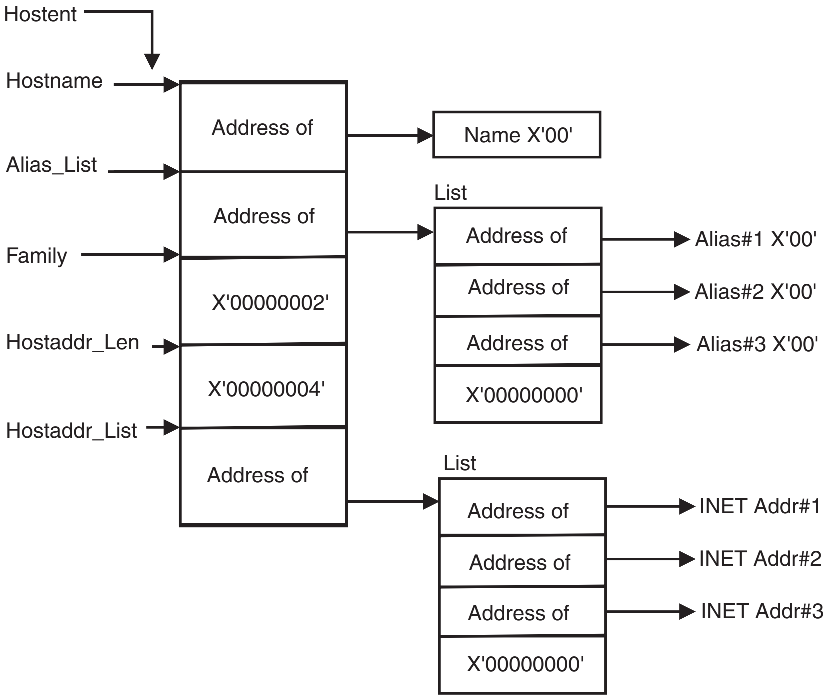


Figure 42. HOSTENT Structure Returned by the GETHOSTBYNAME Macro

GETHOSTBYNAME returns the HOSTENT structure shown in Figure 42. This structure contains:

- The address of the host name returned by the macro. The name length is variable and is ended by X'00'.
- The address of a list of addresses that point to the alias names returned by GETHOSTBYNAME. This list is ended by the pointer X'00000000'. Each alias name is a variable length field ended by X'00'.
- The value returned in the FAMILY field is always 2 to signify AF_INET.
- The length of the host internet address returned in the HOSTADDR_LEN field is always 4 to signify AF_INET.
- The address of a list of addresses that point to the host internet addresses returned by the macro. The list is ended by the pointer X'00000000'.

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

GETHOSTID

The GETHOSTID macro returns the 32-bit identifier for the current host. This value is the default home internet address.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

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

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

Keyword **Description**

RETCODE Output parameter. Returns a fullword binary field containing the 32-bit internet address of the host. A -1 in **RETCODE** indicates an error. There is no **ERRNO** parameter for this macro.

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

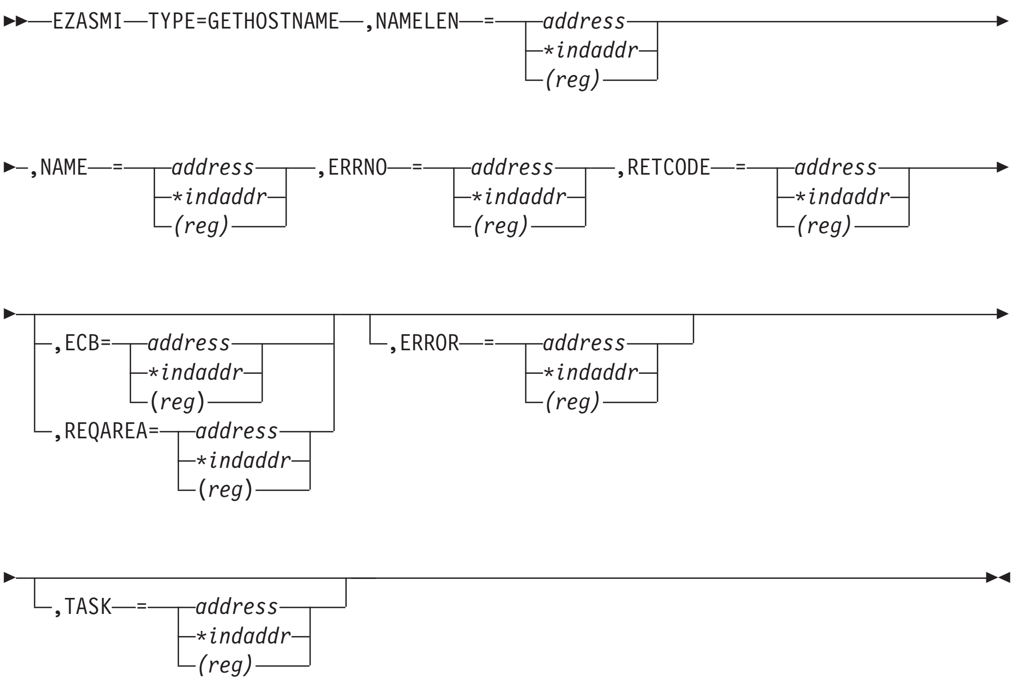
GETHOSTNAME

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked

Control parameters:	All parameters must be addressable by the caller and in the primary address space



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

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

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

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

Keyword
Description

NAMELEN
Input and output parameter. A fullword set to a value, or the address of a fullword binary field set to the length of the name field. The maximum length that can be specified in the field is 255 characters.

NAME
Initially, the application provides a pointer to a receiving field for the host name. TCP/IP for MVS allows a maximum length of 24 characters. This field is filled with a host name the length returned in **NAMELEN** when the call completes.

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

See “Appendix B. Return Codes” on page 547, for information about **ERRNO** return codes.

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

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

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

GETIBMOPT

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

Note: Images from pre-V3R2 releases of TCP/IP for MVS are excluded. The GETIBMOPT macro is not meaningful for pre-V3R2 releases. With this information, the caller can dynamically choose the TCP/IP image with which to connect, using the INITAPI macro. The GETIBMOPT macro 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

TCP/IP for MVS: Planning and Migration Guide contains detailed information about the standard method.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

```
►►──EZASMI──TYPE=GETIBMOPT──,COMMAND──=──┬─number───┬──,BUF──=──┬─address──┬──────────►
                                         ├─address──┤           ├─*indaddr─┤
                                         ├─*indaddr─┤           └─(reg)────┘
                                         └─(reg)────┘

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

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

Keyword **Description**

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

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

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

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

The status field can combine the following information:

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

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

When the status field returns Down and Stopped, TCP/IP abended. Stopped, returned alone, indicates that TCP/IP was stopped.

The version field is X'0308' for TCP/IP for CS for OS/390 Version 2 Release 8.

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

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

Figure 43. NUM_IMAGES Field Settings

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

See “Appendix B. Return Codes” on page 547, for information about **ERRNO** return codes.

RETCODE
Output parameter. A fullword binary field with the following values:

Value	Description
-1	Call returned error. See **ERRNO**.
>=0	Successful call.

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

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

GETPEERNAME

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

The following requirements apply to this call:

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

Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

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

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

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

Keyword
Description

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

NAME
Initially points to the peer name structure, filled when the call completes with the address structure for the remote socket connected to the local socket specified by *s.*

Field
Description

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

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

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

RESERVED
Input parameter. Specifies an eight-byte reserved field. This field is required, but not used.

ERRNO Output parameter. A fullword binary field. If **RETCODE** is negative, this field contains an error number. See “Appendix B. Return Codes” on page 547, for information about **ERRNO** return codes.

RETCODE Output parameter. A fullword binary field.

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

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

GETSOCKNAME

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

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

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

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

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

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

Keyword **Description**

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

NAME
Initially, the application provides a pointer to the socket name structure, which is filled in on completion of the call with the socket name.

Field **Description**

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

PORT Output parameter. A halfword binary field set to the port number bound to this socket. If the socket is not bound, a zero is returned.

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

RESERVED
Input parameter. Specifies eight bytes of binary zeros. This field is required, but not used.

ERRNO Output parameter. A fullword binary field. If **RETCODE** is negative, this field contains an error number. See “Appendix B. Return Codes” on page 547, for information about **ERRNO** return codes.

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

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

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

GETSOCKOPT

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

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key

Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

►─,OPTNAME──=──┬─'SO_REUSEADDR'─┬─,OPTVAL──=──┬─address──┬────────────────────────►
               ├─'SO_BROADCAST'─┤             ├─*indaddr─┤
               ├─'SO_KEEPALIVE'─┤             └─(reg)────┘
               ├─'SO_LINGER'────┤
               ├─'SO_OOBINLINE'─┤
               ├─'SO_SNDBUF'────┤
               ├─'SO_RCVBUF'────┤
               ├─'SO_ERROR'─────┤
               ├─'SO_TYPE'──────┤
               ├─address────────┤
               ├─*indaddr───────┤
               └─(reg)──────────┘

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

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

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

Keyword **Description**

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

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

SO_REUSEADDR
Toggles local address reuse. The default is disabled. This alters the normal algorithm used in the bind() call.

The normal bind() call algorithm allows each internet address and port combination to be bound only once. If the address and port have been bound already, a subsequent bind() will fail and result error EADDRINUSE.

After the 'SO_REUSEADDR' option is active, the following situations are supported:

- A server can bind() the same port multiple times as long as every invocation uses a different local IP address, and the wildcard address INADDR_ANY is used only one time per port.
- A server with active client connections can be restarted and can bind to its port, without having to close all of the client connections.
- For datagram sockets, multicasting is supported so multiple bind() calls can be made to the same class D address and port number.

SO_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 SO_LINGER.

- When the SO_LINGER option is enabled, and data transmission has not been completed, a CLOSE macro blocks the calling program until the data is transmitted or until the connection has timed out.
- If SO_LINGER is not enabled, a CLOSE call returns without blocking the caller and TCP/IP continues to try the send data function. Normally the send data function completes and the data is sent, but it cannot be guaranteed because TCP/IP can timeout before the send has been completed.

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

- When SO_OOBINLINE is enabled, out-of-band data is placed in the normal data input queue as it is received. RECV, and RECVFROM can then receive the data without enabling the MSG-OOB flag.
- When SO_OOBINLINE is disabled, out-of-band data is placed in the priority data input queue as it is received. RECV and RECVFROM must now enable the MSG_OOB flag to receive the data.

SO_ERROR
Requests pending errors on the socket and clears the error status. Use SO_ERROR to check for asynchronous errors on connected datagram sockets or for other errors that are not explicitly returned by one of the socket calls.

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

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

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

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

SO_TYPE
Returns STREAM, DATAGRAM, or RAW.

OPTVAL
Output parameter.

- For all values of **OPTNAME** except SO_LINGER, OPTVAL is a 32-bit fullword, containing the status of the requested option.
  - If the requested option is enabled, the field contains a positive value. If the requested option is disabled, the field contains a 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       DS   F
  LINGER      DS   F
  ```

  – A nonzero value returned in ONOFF indicates that the option is enabled and a zero value indicates that it is disabled.

  – The LINGER value indicates the time in seconds that TCP/IP continues to try to send the data after the CLOSE call is issued. For information about how to set the LINGER time, see “SETSOCKOPT” on page 393.

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

- For all values of **OPTNAME** except SO_LINGER, OPTVAL is one fullword and OPTLEN is set to 4 (one fullword).
- For SO_LINGER, OPTVAL contains two fullwords and OPTLEN is set to 8 (two fullwords).

ERRNO Output parameter. A fullword binary field. If **RETCODE** is negative, this field contains an error number. See “Appendix B. Return Codes” on page 547, for information about **ERRNO** return codes.

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

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

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA

A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

GIVESOCKET

The GIVESOCKET macro makes the socket available for a TAKESOCKET macro issued by another program. The GIVESOCKET macro can specify any connected stream socket. Typically, the GIVESOCKET macro is issued by a concurrent server program that creates sockets to be passed to a subtask.

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

Note: Both concurrent servers and iterative servers use this interface. An iterative server handles one client at a time. A concurrent server receives connection requests from multiple clients and creates subtasks that process the client requests. When a subtask is created, the concurrent server gets a new socket, passes the new socket to the subtask, and dissociates itself from the connection. The TCP/IP Listener program is an example of a concurrent server.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

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

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

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

Keyword **Description**

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

CLIENT Input parameter. The client ID for this application.

Field **Description**

DOMAIN
Input parameter. A fullword binary number specifying the domain of the client. For TCP/IP the value is always 2, indicating AF_INET.

NAME Initially, the application provides a pointer to an eight-character field, left-justified, padded to the right with blanks, which is filled in on completion of the call with the MVS address space name of the application that is going to take the socket. If the socket-taking application is in the same address space as the socket-giving application, **NAME** can be obtained using the GETCLIENTID call. If this field is set to blanks, any MVS address space requesting a socket can take this socket.

TASK Specifies an eight-character field that is set to the MVS subtask identifier of the socket-taking task (specified on the SUBTASK parameter on its INITAPI macro). If this field is set to blanks, any subtask in the address space specified in the **NAME** field can take the socket.

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

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

See “Appendix B. Return Codes” on page 547, for information about **ERRNO** return codes.

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

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

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

GLOBAL

The GLOBAL macro allocates a global storage area that is addressable by all socket users in an address space. If more than one module is using sockets, you must supply the address of the global storage area to each user. Each program using the sockets interface should define global storage using the instruction EZASMI TYPE=GLOBAL with STORAGE=DSECT.

If this macro is not named, the default name EZASMGWA is assumed.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

```
►►──EZASMI──TYPE=GLOBAL──,STORAGE──=──┬─DSECT─┬───────────────────────────►◄
                                      └─CSECT─┘
```

Keyword	Description
STORAGE	Input parameter. Defines one of the following storage definitions:

Keyword	Description
DSECT	Generates a DSECT.

CSECT Generates an in-line storage definition that can be used within a CSECT or as a part of a larger DSECT.

INITAPI

The INITAPI macro connects an application to the TCP/IP interface. Almost all sockets programs that are written in COBOL, PL/I, or assembler language must issue the INITAPI macro before they issue other sockets macros.

Note: Because the default INITAPI still requires the TERMAPI to be issued, it is recommended that you always code the INITAPI command.

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

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

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

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

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

►──┬───────────────────────────────────────────┬──┬───────────────────────────┬──►
   └─,ASYNC──=──┬─'NO'─────────────────────────┤  └─,ERROR──=──┬─indaddr─┬───┘
                ├─'ECB'────────────────────────┤               └─(reg)───┘
                └─('EXIT',──┬─address)──┬──────┘
                            ├─*indaddr)─┤
                            └─(reg))────┘

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

Keyword
Descriptions

MAXSOC
Optional input parameter. A halfword binary field specifying the maximum number of sockets supported by this application. The maximum number is 2000 and the minimum number is 50. The default value for **MAXSOC** is 50. If less than 50 are requested, **MAXSOC** defaults to 50.

SUBTASK
Optional input parameter. An eight-byte field that is used to identify a subtask in an address space that can contain multiple subtasks. It is suggested that you use your own 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.

IDENT
Optional input parameter. A structure containing the identities of the TCP/IP address space and your address space. Specify **IDENT** on the INITAPI macro from an address space. The structure is as follows:

Field **Description**

TCPNAME
Input parameter. An eight-byte character field set to the name of the TCP/IP address space that you want to connect to. If this is not specified, the system derives a value from the configuration file, as described in *OS/390 IBM Communications Server: IP Configuration Reference*.

ADSNAME
Input parameter. An eight-byte character field set to the name of the calling program's address space. If this is not specified, the system will derive a value from the MVS control block structure.

MAXSNO Output parameter. A fullword binary field containing the greatest descriptor number assigned to this application. The lowest socket number is zero. If you have 50 sockets, they are numbered in the range 0–49. If **MAXNO** is not specified, the value for **MAXNO** is 49.

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

See "Appendix B. Return Codes" on page 547, for information about **ERRNO** return codes.

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

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

APITYPE Optional input parameter. A halfword binary field specifying the APITYPE:

Value	Meaning
2	APITYPE 2 (AF_INET). This is the default.
3	APITYPE 3 (AF_INET with modified blocking) **Note:** For details on usage, see "Task Management and Asynchronous Function Processing" on page 220.

For an APITYPE value of 3, the ASYNC parameter must be either 'ECB' or 'EXIT'.

UEEXIT Optional input parameter. A doubleword value as follows:

- A fullword specifying the entry point address of the user unsolicited event exit.
- A fullword specifying the token that will be presented to the unsolicited event exit at invocation.

ASYNC Optional input parameter. One of the following:

- The literal 'NO' indicating no asynchronous support.
- The literal 'ECB' indicating the asynchronous support using ECBs is to be used.
- The combination of the literal 'EXIT' and the address of a doubleword value as follows:
  - A fullword specifying the entry point address of the user's asynchronous event exit.
  - A fullword specifying the token which will be presented to the asynchronous event exit at invocation.

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

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

IOCTL

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

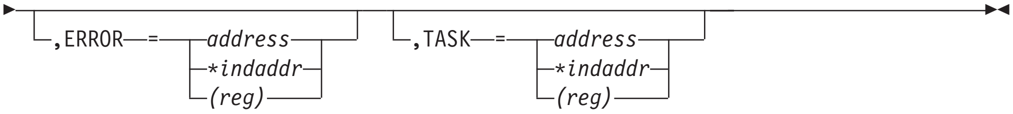
Before you issue an IOCTL macro, you must load a value representing the characteristic that you want to control in **COMMAND**.

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

```
►►──EZASMI──TYPE=IOCTL──,S──=──┬─number───┬──,COMMAND──=──┬─'FIONBIO'───────┬──────►
                               ├─address──┤               ├─'FIONREAD'──────┤
                               ├─*indaddr─┤               ├─'SIOCATMARK'────┤
                               └─(reg)────┘               ├─'SIOCGIFADDR'───┤
                                                          ├─'SIOCGIFBRDADDR'┤
                                                          ├─'SIOCGIFCONF'───┤
                                                          ├─'SIOCGIFDSTADDR'┤
                                                          ├─'SIOCGMONDATA'──┤
                                                          ├─'SIOCGSPLXFQDN'─┤
                                                          ├─address─────────┤
                                                          ├─*indaddr────────┤
                                                          └─(reg)───────────┘

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

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



Keyword
Description

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

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

Value
Description

'FIONBIO'
Sets or clears blocking status.

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

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

'SIOCGIFADDR'
Requests the network interface address for an interface name. For the address format, see Figure 44.

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

Figure 44. Interface Request Structure (IFREQ) for IOCTL Macro

'SIOCGIFBRDADDR'
Requests the network interface broadcast address for an interface name. For the address format, see Figure 44.

'SIOCGIFCONF'
Requests the network interface configuration. The configuration consists of a variable number of 32-byte arrays, formatted as shown in Figure 44.

- When IOCTL is issued, the first word in **REQARG** must contain the length (in bytes) of the array to be returned, and the second word in **REQARG** should be set to the number of interfaces requested times 32 (one address structure for each network interface). The maximum number of array elements that TCP/IP for MVS will return is 100.

- When IOCTL is issued, **RETARG** must be set to the beginning of the area in your program's storage which is reserved for the array that is to be returned by IOCTL.
- The **COMMAND** 'SIOGIFCONF' returns a variable number of network interface configurations. Figure 45 contains an example of a routine that can be used to work with such a structure.

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

Figure 45. Assembler Language Example for SIOCGIFCONF

To get length:

– Multiply COUNT by 32 to get ARRAY-LENGTH

– Set REQARG equal to ARRAY-LENGTH

– Enter; EZASMI TYPE=, S=, COMMAND=, REQARG=, RETARG=, ERRNO=, RETCODE=, ECB=, ERROR=

– Set GTABLE to RETARG.

'SIOCGIFDSTADDR'
Requests the network interface destination address.

'SIOCGMONDATA'
Returns the stack data reported by the CBSAMPLE program.

The ARP counter data provided differs depending on the type of device. Refer to the section on devices that support ARP Offload in the *OS/390 IBM Communications Server: IP Configuration Guide* for more information on what is supported for each device.

'SIOCGSPLXFQDN'
Requests the fully qualified domain name for a given server and group name in a sysplex. This is a special purpose command to support applications that have registered with WorkLoad Manager (WLM) for connection optimization services by way of the DNS. When IOCTL is issued, **REQARG** and **RETARG** must use the address structure sysplexFqDn, which contains the pointer for sysplexFqDnData structure. The fully qualified domain name is returned in the domainName field of sysplexFqDnData. The group name and the server name can be passed using the groupName

and serverName fields of sysplexFqDnData structure. Their structures are defined in the EZBZSDNP MACRO file.

REQARG and RETARG

Point to arguments that are passed between the calling program and IOCTL. The length of the argument is determined by the **COMMAND** request. **REQARG** is an input parameter and is used to pass arguments to IOCTL. **RETARG** is an output parameter and is used for arguments returned by IOCTL.

For the lengths and meanings of **REQARG** and **RETARG** for each **COMMAND** type, see Table 16.

Table 16. IOCTL Macro Arguments

COMMAND/CODE	SIZE	REQARG	SIZE	RETARG
FIONBIO X'8004A77E'	4	Set socket mode to: X'00'=blocking; X'01'=nonblocking	0	Not used
FIONREAD X'4004A77F'	0	not used	4	Number of characters available for read
SIOCATMARK X'4004A707'	0	Not used	4	X'00'= at OOB data X'01'= not at OOB data
SIOCGIFADDR X'C020A70D'	32	First 16 bytes- interface name Last 16 bytes- not used	32	Network interface address (See Figure 44 on page 264 for format.)
SIOCGIFBRDADDR X'C020A712'	32	First 16 bytes- interface name Last 16 bytes- not used	32	Network interface address (See Figure 44 on page 264 for format.)
SIOCGIFCONF X'C008A714'	8	First 4 bytes - size of return buffer, Last 4 bytes - address of return buffer	See note.	
Note: The second 4-byte in the RETARG is the address to the user buffer containing an array of 32-byte Socket Name Structures (See Figure 45 on page 265 for format). Each interface is assigned a 32-byte array element and REQARG should be set to the number of interfaces times 32. TCP/IP 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 44 on page 264 for format.)
SIOCGMONDATA X'C018D902'	—	See MONDATAIN structure in macro EZBZMONP	—	See MONDATAOUT structure in macro EZBZMONP
SIOCGSPLXFQDN X'C018D905'	408 (see 266)	See sysplexFqDn and sysplexFqDnData in macro EZBZSDSP	408 (see 266)	See sysplexFqDn and sysplexFqDnData in macro EZBZSDSP
Note: Both REQARG and RETARG must contain both sysplexFqDn and sysplexFqDnData.				

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

See “Appendix B. Return Codes” on page 547, for information about **ERRNO** return codes.

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

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

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

LISTEN

Only servers use the LISTEN macro. The LISTEN macro:

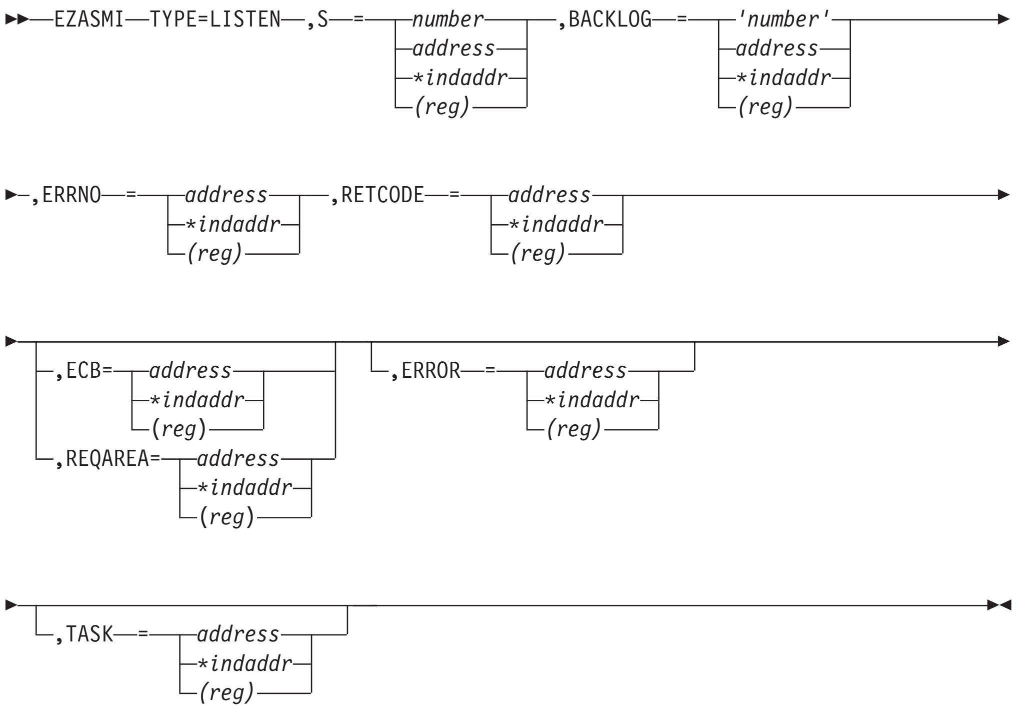
- Completes the bind, if BIND has not already been called for the socket. If the BIND has already been called for in the socket, the LISTEN macro uses what was specified in the BIND call.
- Creates a connection-request queue of a specified number of entries for incoming connection requests.

The LISTEN macro is typically used by a concurrent server to receive connection requests from clients. When a connection request is received, a new socket is created by a later ACCEPT macro. The original socket continues to listen for additional connection requests.

Note: Concurrent servers and iterative servers use this macro. An iterative server handles one client at a time. A concurrent server receives connection requests from multiple clients and creates subtasks that process the client requests. When a subtask is created, the concurrent server gets a new socket, passes the new socket to the subtask, and dissociates itself from the connection. The TCP/IP Listener program is an example of a concurrent server.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space



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

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

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

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

Keyword **Description**

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

BACKLOG Input parameter. A value (enclosed in single quotation marks) or the address of a fullword binary number specifying the number of messages that can be backlogged.

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

See “Appendix B. Return Codes” on page 547, for information about **ERRNO** return codes.

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

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

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

READ

The READ macro reads data on a socket and stores it in a buffer. The READ macro applies only to connected sockets.

For datagram sockets, the READ call returns the entire datagram that was sent. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard extra bytes.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

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

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

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

Keyword **Description**

S
Input parameter. A value, or the address of a halfword binary number specifying the socket that is going to read the data.

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

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

ALET
Optional input parameter. A fullword binary field containing the **ALET** or **BUF**. The default is 0 (primary address space).

If a nonzero ALET is specified, the ALET must represent a valid entry in the dispatchable unit access list (DU-AL) for the task issuing this call. Note that ALETs can only be specified for synchronous socket calls (for example, ECB/REQAREA not specified). An exception to this is an ALET representing a SCOPE=COMMON data space.

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

See “Appendix B. Return Codes” on page 547, for information about **ERRNO** return codes.

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

Value **Description**

0
A zero return code indicates that the connection is closed and no data is available.

>0 A positive value indicates the number of bytes copied into the buffer.

–1 Check **ERRNO** for an error code.

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

READ returns up to the number of bytes specified by **NBYTE**. If less than the number of bytes requested is available, the READ macro returns the number currently available.

If data is not available for the socket and the socket is in blocking mode, the READ macro blocks the caller until data arrives. If data is not available, and the socket is in nonblocking mode, READ returns a -1 and sets **ERRNO** to 35 (EWOULDBLOCK). See “IOCTL” on page 262 or “FCNTL” on page 234 for a description of how to set the nonblocking mode.

READV

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode

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

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

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

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

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

Keyword
Description

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

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

Fullword 1
Input parameter. A buffer to be filled by the completion of the call.

Fullword 2
Input parameter. The **ALET** for this buffer. If the buffer is in the primary address space, this should be zeros.

If a nonzero ALET is specified, the ALET must represent a valid entry in the dispatchable unit access list (DU-AL) for the task issuing this call. Note that ALETs can only be specified for synchronous socket calls (for example, ECB/REQAREA not specified). An exception to this is an ALET representing a SCOPE=COMMON data space.

Fullword 3
Input parameter. The length of the data buffer referred to in Fullword 1.

IOVCNT
Input parameter. A fullword binary field specifying the number of data buffers provided for this call. The limit is 120.

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

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

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

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

RECV

The RECV macro receives data on a socket and stores it in a buffer. The RECV macro applies only to connected sockets. RECV can read the next message, but leave the data in a buffer, and can read out-of-band data. RECV gives you the option of setting **FLAGS** with the **FLAGS** parameter.

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

For stream sockets, the data is processed like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and Application A sends 1000 bytes, each call to RECV can return one byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place RECV in a loop that repeats the call until all data has been received.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

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

►─,RETCODE──=──┬─address──┬─┬──────────────────────────────┬─────────────────────►
               ├─*indaddr─┤ └─,FLAGS──=──┬─'MSG_OOB'──┬────┘
               └─(reg)────┘              ├─'MSG_PEEK'─┤
                                         ├─address────┤
                                         ├─*indaddr───┤
                                         └─(reg)──────┘

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

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

Keyword **Description**

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

NBYTE
Input parameter. A fullword binary number set to the size of **BUF**.

RECV does not receive more than the number of bytes of data in **NBYTE** even if more data is available.

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

ALET
Optional input parameter. A fullword binary field containing the **ALET** of **BUF**. The default is 0 (primary address space).

If a nonzero ALET is specified, the ALET must represent a valid entry in the dispatchable unit access list (DU-AL) for the task issuing this call. Note that ALETs can only be specified for synchronous socket calls (for example, ECB/REQAREA not specified). An exception to this is an ALET representing a SCOPE=COMMON data space.

ERRNO
Ouput parameter. A fullword binary field. If **RETCODE** is negative, this field contains an error number. See “Appendix B. Return Codes” on page 547, 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.

FLAGS
Input parameter. **FLAGS** can be a literal value or a fullword binary field:

Literal Value	Binary Value	Description
'MSG_OOB'	1	Receive out-of-band data. (Stream sockets only.)
'MSG_PEEK'	2	Peek at the data, but do not destroy the data.

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

If data is not available for the socket and the socket is in blocking mode, the RECV macro blocks the caller until data arrives. If data is not available and the socket is in nonblocking mode, RECV returns a –1 and sets **ERRNO** to 35 (EWOULDBLOCK). See “FCNTL” on page 234 or “IOCTL” on page 262 for a description of how to set nonblocking mode.

RECVFROM

The RECVFROM macro receives data for a socket and stores it in a buffer. RECVFROM returns the length of the incoming message or data stream.

If data is not available for the socket designated by descriptor S, and socket S is in blocking mode, the RECVFROM call blocks the caller until data arrives.

If data is not available and socket S is in nonblocking mode, RECVFROM returns a –1 and sets **ERRNO** to 35 (EWOULDBLOCK). Because RECVFROM returns the socket address in the **NAME** structure, it applies to any datagram socket, whether connected or unconnected. See “FCNTL” on page 234 or “IOCTL” on page 262 for a description of how to set nonblocking mode. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard extra bytes.

For stream sockets, the data is processed as streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and Application A sends 1000 bytes, each call to this function can return one byte, or 10 bytes, or the entire 1000 bytes. Applications using stream sockets should place RECVFROM in a loop that repeats until all of the data has been received.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

```
►─,BUF──=──┬─address──┬─,NAME──=──┬─address──┬──┬──────────────────────────────┬──────►
           ├─*indaddr─┤           ├─*indaddr─┤  └─,ALET──=──┬─address──┬──────┘
           └─(reg)────┘           └─(reg)────┘              ├─*indaddr─┤
                                                            └─(reg)────┘

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

►──┬────────────────────────────────┬──┬─────────────────────────────┬────────────────►
   └─,FLAGS──=──┬─'MSG_OOB'──┬──────┘  ├─,ECB=──┬─address──┬─────────┤
                ├─'MSG_PEEK'─┤         │        ├─*indaddr─┤         │
                ├─address────┤         │        └─(reg)────┘         │
                ├─*indaddr───┤         └─,REQAREA=──┬─address──┬─────┘
                └─(reg)──────┘                      ├─*indaddr─┤
                                                    └─(reg)────┘

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

Keyword **Description**

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

NBYTE
Input parameter. A value, or the address of a fullword binary number specifying the length of the input buffer. **NBYTE** must first be initialized to the size of the buffer associated with **NAME**. On return the **NBYTE** contains the number of bytes of data received.

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

NAME
Initially, the application provides a pointer to a structure that will contain the peer socket name on completion of the call.

If the **NAME** parameter value is nonzero, the source address of the message is filled.

Field **Description**

FAMILY
Output parameter. A halfword binary number specifying the addressing family. The value is always 2, indicating AF_INET.

PORT
Output parameter. A halfword binary number specifying the port number of the sending socket.

IP-ADDRESS
Output parameter. A fullword binary number specifying the 32-bit internet address of the sending socket.

RESERVED
Output parameter. An eight-byte reserved field. This field is required, but is not used.

ALET
Optional input parameter. A fullword binary field containing the **ALET** of **BUF**. The default is 0 (primary address space).

If a nonzero ALET is specified, the ALET must represent a valid entry in the dispatchable unit access list (DU-AL) for the task issuing this call. Note that ALETs can only be specified for synchronous socket calls (for example, ECB/REQAREA not specified). An exception to this is an ALET representing a SCOPE=COMMON data space.

ERRNO
Output parameter. A fullword binary field. If **RETCODE** is negative, this field contains an error number. See "Appendix B. Return Codes" on page 547, for information about **ERRNO** return codes.

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

Value **Description**

0 A zero return code indicates that the connection is closed and no data is available.

>0 A positive value indicates the number of bytes transferred by the RECVFROM call.

–1 Check **ERRNO** for an error code.

FLAGS
Input parameter. **FLAGS** can be a literal value or a fullword binary field:

Literal Value	Binary Value	Description
'MSG_OOB'	1	Receive out-of-band data. (Stream sockets only.)
'MSG_PEEK'	2	Peek at the data, but do not destroy the data.

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

RECVMSG

The RECVMSG macro receives messages on a socket with descriptor *s* and stores them in an array of message headers. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard extra bytes.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

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

►──┬─────────────────────────────────┬──┬────────────────────────────────┬──────────────►
   └─,FLAGS──=──┬─'MSG_OOB'──┬───────┘  ├─,ECB=──┬─address──┬────────────┤
                ├─'MSG_PEEK'─┤          │        ├─*indaddr─┤            │
                ├─address────┤          │        └─(reg)────┘            │
                ├─*indaddr───┤          └─,REQAREA=──┬─address──┬────────┘
                └─(reg)──────┘                       ├─*indaddr─┤
                                                     └─(reg)────┘

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

Keyword **Description**

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

MSG
On input, this is a pointer to a message header into which the message is received on completion of the call.

Field **Description**

NAME On input, a pointer to a buffer where the sender's address will be stored on completion of the call.

NAMELEN
On input, a pointer to the size of the **NAME**.

IOV On input, a pointer an array of three fullword structures with the number of structures equal to the value in IOVCNT and the format of the structures as follows:

Fullword 1
Input parameter. The address of a data buffer.

Fullword 2
Input parameter. The **ALET** for this buffer. If the buffer is in the primary address space, this should be zeros.

If a nonzero ALET is specified, the ALET must represent a valid entry in the dispatchable unit access list (DU-AL) for the task issuing this call. Note that ALETs can only be specified for synchronous socket calls (for example, ECB/REQAREA not specified). An exception to this is an ALET representing a SCOPE=COMMON data space.

Fullword 3
Input parameter. The length of the data buffer referenced in Fullword 1.

IOVCNT
On input, a pointer to a fullword binary field specifying the number of data buffers provided for this call.

ACCRIGHTS
On input, a pointer to the access rights received. This field is ignored.

ACCRLEN
On input, a pointer to the length of the access rights received. This field is ignored.

FLAGS Input parameter. **FLAGS** can be a literal value or a fullword binary field:

Literal Value	Binary Value	Description
'MSG_OOB'	1	Receive out-of-band data. (Stream sockets only.)
'MSG_PEEK'	2	Peek at the data, but do not destroy the data.

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

RETCODE Output parameter. A fullword binary field with the following values:

Value **Description**

-1 Call returned error. See **ERRNO** field.

0 Connection partner has closed connection.

>0 Number of bytes read.

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

SELECT

In a process where multiple I/O operations can occur it is necessary for the program to be able to wait on one or several of the operations to complete. For example, consider a program that issues a READ to multiple sockets whose blocking mode is set. Because the socket would block on a READ macro, only one socket could be read at a time. Setting the sockets to nonblocking would solve this problem, but would require polling each socket repeatedly until data becomes available. The SELECT macro allows you to test several sockets and to process a later I/O macro only when one of the tested sockets is ready. This ensures that the I/O macro does not block.

To use the SELECT macro as a timer in your program, do either of the following:
- Set the read, write, and except arrays to zeros
- Do not specify MAXSOC.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked

Control parameters:	All parameters must be addressable by the caller and in the primary address space

Testing Sockets

Read, write, and exception operations can be tested. The select () call monitors activity on selected sockets to determine whether:

- A buffer for the specified sockets contains input data. If input data is available for a given socket, a read operation on that socket does not block.
- TCP/IP can accommodate additional output data. If TCP/IP can accept additional output for a socket, a write operation on the socket does not block.
- An exceptional condition occurs on a socket.
- A timeout occurs on the SELECT macro itself. A TIMEOUT period can be specified when the SELECT macro is issued.

Each socket descriptor is represented by a bit in a bit string.

Read Operations

The ACCEPT, READ, READV, RECV, RECVFROM, and RECVMSG macros are read operations. A socket is ready for reading when data is received on it, or when an exception condition occurs.

To determine if a socket is ready for the read operation, set the appropriate bit in RSNDMSK to '1' before issuing the SELECT macro. When the SELECT macro returns, the corresponding bits in the RRETMSK indicate sockets ready for reading.

Write Operations

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

- TCP/IP can accept additional outgoing data.
- A connection request is received in response to an ACCEPT macro.
- A CONNECT call for a nonblocking socket, that has previously returned ERRNO 36 (EINPROGRESS), completes the connection.

The WRITE, WRITEV, SEND, SENDMSG, or SENDTO macros block when the data to be sent exceeds the amount that TCP/IP can accept. To avoid this, you can precede the write operation with a SELECT macro to ensure that the socket is ready for writing. After a socket is selected for WRITE, your program can determine the amount of TCP/IP buffer space available by issuing the GETSOCKOPT macro with the SO_SNDBUF option.

To determine if a socket is ready for the write operation, set the appropriate bit in WSNDMSK to '1'.

Exception Operations

For each socket to be tested, the SELECT macro can check for an exception condition. The exception conditions are:

- The calling program (concurrent server) has issued a GIVESOCKET command and the target subtask has successfully issued the TAKESOCKET call. When this condition is selected, the calling program (concurrent server) should issue CLOSE to dissociate itself from the socket.
- A socket has received out-of-band data. For this condition, a READ macro returns the out-of-band data before the program data.

To determine if a socket has an exception condition, use the ESNDMSK character string and set the appropriate bits to '1'.

Returning the Results

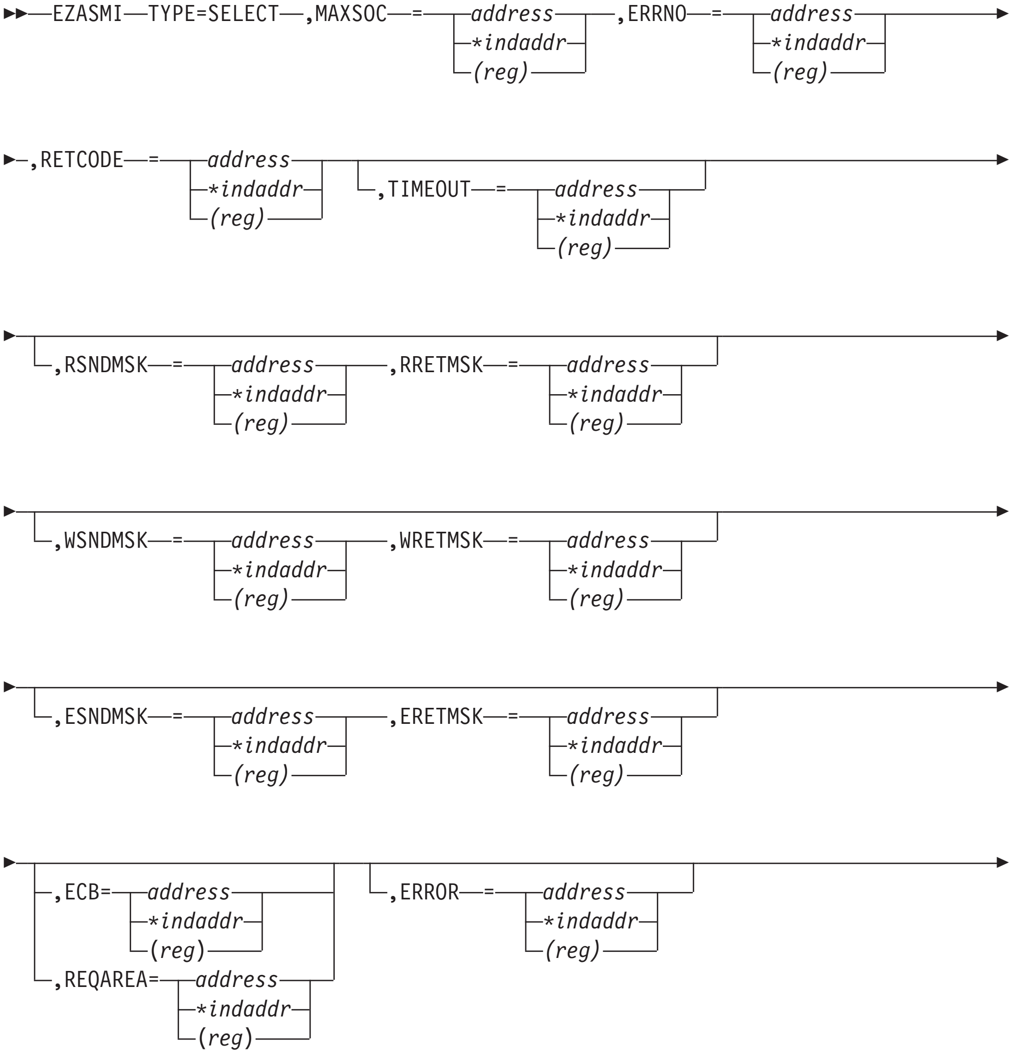
For each event tested by a *x*SNDMSK, a bit string records the results of the check. The bit strings are RRETMSK, WRETMSK, and ERETMSK for read, write, and exceptional events. On return from the SELECT macro, each bit set to ‘1’ in the xRETMSK is a read, write, or exceptional event for the associated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

MAXSOC Parameter

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

TIMEOUT Parameter

If the time in the TIMEOUT parameter elapses before an event is detected, the SELECT call returns and RETCODE is set to 0.



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

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

►─┬────────────────────────────────────────────────────────────────────┬─────────────►
  └─,RSNDMSK──=──┬─address──┬──,RRETMSK──=──┬─address──┬───────────────┘
                 ├─*indaddr─┤               ├─*indaddr─┤
                 └─(reg)────┘               └─(reg)────┘

►─┬────────────────────────────────────────────────────────────────────┬─────────────►
  └─,WSNDMSK──=──┬─address──┬──,WRETMSK──=──┬─address──┬───────────────┘
                 ├─*indaddr─┤               ├─*indaddr─┤
                 └─(reg)────┘               └─(reg)────┘

►─┬────────────────────────────────────────────────────────────────────┬─────────────►
  └─,ESNDMSK──=──┬─address──┬──,ERETMSK──=──┬─address──┬───────────────┘
                 ├─*indaddr─┤               ├─*indaddr─┤
                 └─(reg)────┘               └─(reg)────┘

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

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

Keyword **Description**

MAXSOC Input parameter. A fullword binary field specifying the largest socket descriptor number to be checked plus one. Socket numbering is in the range 0–1999.

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

See “Appendix B. Return Codes” on page 547, for information about **ERRNO** return codes.

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

Value	**Description**
>0	Indicates the number of ready sockets in the three return masks. **Note:** If the number of ready sockets is greater than 65,535, only 65,353 is reported.
=0	Indicates that the SELECT time limit has expired.
–1	Check **ERRNO** for an error code

TIMEOUT Input parameter.

If **TIMEOUT** is not specified, the SELECT call blocks until a socket becomes ready.

If **TIMEOUT** is specified, **TIMEOUT** is the maximum interval for the SELECT call to wait until completion of the call. If you want SELECT to poll the sockets and return immediately, **TIMEOUT** should be specified to point to a zero-valued TIMEVAL structure.

TIMEOUT is specified in the two-word **TIMEOUT** as follows:

- TIMEOUT-SECONDS, word one of **TIMEOUT**, is the seconds component of the time-out value.
- TIMEOUT-MICROSEC, word two of **TIMEOUT**, is the microseconds component of the time-out value (0–999999).

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

For APITYPE=3 with an ECB specified, the SELECT call will return immediately because it is asynchronous; the ECB will be POSTed when the timer pops.

RSNDMSK Input parameter. A bit string sent to request read event status.

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

If this parameter is set to 0, the SELECT will not check for read events. The length of this bit-mask array is dependent on the value in MAXSOC and must be a multiple of 4 bytes. See “Selecting Requests” on page 43 for more information.

RRETMSK
Output parameter. A bit string that returns the status of read events.

- For each socket that is ready for to read, the corresponding bit in the string will be set to 1.
- For sockets to be ignored, the corresponding bit in the string will be set to 0.

WSNDMSK
Input parameter. A bit string sent to request write event status.

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

WRETMSK
Output parameter. A bit string that returns the status of write events.

- For each socket that is ready to write, the corresponding bit in the string will be set to 1.
- For sockets that are not ready to be written, the corresponding bit in the string will be set to 0.

ESNDMSK
Input parameter. A bit string sent to request exception event status. The length of the string should be equal to the maximum number of sockets to be checked.

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

ERETMSK
Output parameter. A bit string that returns the status of exception events. The length of the string should be equal to the maximum number of sockets to be checked.

- For each socket for which exception status has been set, the corresponding bit will be set to 1.
- For sockets that do not have exception status, the corresponding bit will be set to 0.

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

SELECTEX

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

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

- Set the read, write, and except arrays to zeros
- Do not specify MAXSOC.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

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

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

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

►─┬──────────────────────────────────────────────────────────────────┬──────►
  └─,ESNDMSK──=──┬─address───┬─,ERETMSK──=──┬─address───┬──┘
                 ├─*indaddr──┤              ├─*indaddr──┤
                 └─(reg)─────┘              └─(reg)─────┘

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

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

Keyword	Description
MAXSOC	Input parameter. A fullword binary field specifying the largest socket descriptor number being checked.
ERRNO	Output parameter. A fullword binary field. If **RETCODE** is negative, this contains an error number.
RETCODE	Output parameter. A fullword binary field.

Value	Meaning
>0	The number of ready sockets. **Note:** If the number of ready sockets is greater than 65,535, only 65,353 is reported.
0	Either the SELECTEX time limit has expired (ECB value will be 0) or one of the caller's ECBs has been posted (ECB value will be nonzero and the caller's descriptor sets will be set to 0). The caller must initialize the ECB values to zero before issuing the SELECTEX macro.
-1	Check **ERRNO**.

TIMEOUT
Input parameter.

If **TIMEOUT** is not specified, the SELECTEX call blocks until a socket becomes ready or until a user **ECB** is posted.

If a **TIMEOUT** value is specified, **TIMEOUT** is the maximum interval for the SELECTEX call to wait until completion of the call. If you want SELECTEX to poll the sockets and return immediately, **TIMEOUT** should be specified to point to a zero-valued TIMEVAL structure.

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

- TIMEOUT-SECONDS, word one of **TIMEOUT**, is the seconds component of the time-out value.

- TIMEOUT-MICROSEC, word two of **TIMEOUT**, is the microseconds component of the time-out value (0—999999).

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

RSNDMSK Input parameter. The bit-mask array to control checking for read interrupts. If this parameter is not specified or the specified bit-mask is zeros, the SELECT will not check for read interrupts. The length of this bit-mask array is dependent on the value in MAXSOC and must be a multiple of 4 bytes. See "Selecting Requests" on page 43 for more information.

RRETMSK Output parameter. The bit-mask array returned by the SELECT if **RSNDMSK** is specified. The length of this bit-mask array is dependent on the value in MAXSOC and must be a multiple of 4 bytes. See "Selecting Requests" on page 43 for more information.

WSNDMSK Input parameter. The bit-mask array to control checking for write interrupts. If this parameter is not specified or the specified bit-mask is zeros, the SELECT will not check for write interrupts. The length of this bit-mask array is dependent on the value in MAXSOC and must be a multiple of 4 bytes. See "Selecting Requests" on page 43 for more information.

WRETMSK Output parameter. The bit-mask array returned by the SELECT if **WSNDMSK** is specified. The length of this bit-mask array is dependent on the value in MAXSOC and must be a multiple of 4 bytes. See "Selecting Requests" on page 43 for more information.

ESNDMSK Input parameter. The bit-mask array to control checking for exception interrupts. If this parameter is not specified or the specified bit-mask is zeros, the SELECT will not check for exception interrupts. The length of this bit-mask array is dependent on the value in MAXSOC and must be a multiple of 4 bytes. See "Selecting Requests" on page 43 for more information.

ERETMSK Output parameter. The bit-mask array returned by the SELECT if ESNDMSK is specified. The length of this bit-mask array is dependent on the value in MAXSOC and must be a multiple of 4 bytes. See "Selecting Requests" on page 43 for more information.

SELECB Input parameter. An **ECB** or list of **ECB** addresses 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 and you must also add the LIST keyword. The **ECB**s must reside in the caller's home address space.

Note: The maximum number of **ECB**s that can be specified in a list is 1013.

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

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

SEND

The SEND macro sends datagrams on a specified connected socket.

FLAGS allows you to:

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

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

For stream sockets, data is processed as streams of information with no boundaries separating the data. For example, if a program is required to send 1000 bytes, each call to this function can send any number of bytes, up to the entire 1000 bytes, with the number of bytes sent returned in **RETCODE**. Therefore, programs using stream sockets should place this call in a loop, and reissue the call until all data has been sent.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

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

```
►─,RETCODE──=──┬─address──┬──┬────────────────────────────────────┬────────►
               ├─*indaddr─┤  └─,FLAGS──=──┬─'MSG_OOB'─────────┬──┘
               └─(reg)────┘               ├─'MSG_DONTROUTE'─┤
                                          ├─address─────────┤
                                          ├─*indaddr────────┤
                                          └─(reg)───────────┘

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

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

Keyword
Description

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

NBYTE
Input parameter. A value, or the address of a fullword binary number specifying the number of bytes to transmit.

BUF
The address of the data being transmitted. The length of **BUF** must be at least as long as the value of **NBYTE**.

ALET
Optional input parameter. A fullword binary field containing the **ALET** of **BUF**. The default is 0 (primary address space).

If a nonzero ALET is specified, the ALET must represent a valid entry in the dispatchable unit access list (DU-AL) for the task issuing this call. Note that ALETs can only be specified for synchronous socket calls (for example, ECB/REQAREA not specified). An exception to this is an ALET representing a SCOPE=COMMON data space.

ERRNO
Output parameter. A fullword binary field. If **RETCODE** is negative, this field contains an error number. See "Appendix B. Return Codes" on page 547, for information about **ERRNO** return codes.

RETCODE
Output parameter. A fullword binary field.

Value
Description

0 or >0
A successful call. The value is set to the number of bytes transmitted.

–1
Check **ERRNO** for an error code

FLAGS
Input parameter. **FLAGS** can be a literal value or a fullword binary field:

Literal Value	Binary Value	Description
'MSG_OOB'	1	Send out-of-band data. (Stream sockets only.)
'MSG_DONTROUTE'	4	Do not route. Routing is handled by the calling program.

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

SENDMSG

The SENDMSG macro sends messages on a socket with descriptor *s* passed in an array of messages.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

►──┬────────────────────────────────┬──,ERRNO──=──┬─address──┬──────────────────►
   └─,FLAGS──=──┬─'MSG_OOB'──┬──────┘             ├─*indaddr─┤
                ├─'MSG_PEEK'─┤                    └─(reg)────┘
                ├─address────┤
                ├─*indaddr───┤
                └─(reg)──────┘

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

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

Keyword
Description

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

MSG
Input parameter. A pointer to the message header into which the message is received.

Field
Description

NAME
A pointer to a buffer that contains the destination of the data on input.

NAMELEN
A pointer to the size of the address buffer.

IOV
A pointer to an array of three fullword structures with the number of structures equal to the value in IOVCNT and the format of the structures as follows:

Fullword 1
Input parameter. The address of a data buffer.

Fullword 2
Input parameter. The **ALET** for this buffer. If the buffer is in the primary address space, this should be zeros.

If a nonzero ALET is specified, the ALET must represent a valid entry in the dispatchable unit access list (DU-AL) for the task issuing this call. Note that ALETs can only be specified for synchronous socket calls (for example, ECB/REQAREA not specified). An exception to this is an ALET representing a SCOPE=COMMON data space.

Fullword 3
Input parameter. The length of the data buffer referenced in Fullword 1.

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

ACCRIGHTS
A pointer to the access rights sent. This field is ignored.

ACCRLEN
A pointer to the length of the access rights sent. This field is ignored.

FLAGS Input parameter. **FLAGS** can be a literal value or a fullword binary field:

Literal Value	Binary Value	Description
'MSG_OOB'	1	Send out-of-band data. (Stream sockets only.)
'MSG_DONTROUTE'	4	Do not route. Routing is handled by the calling program.

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

RETCODE Output parameter. A fullword binary field.

Value **Description**

0 or >0
A successful call. The value is set to the number of bytes transmitted.

–1 Check **ERRNO** for an error code

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

SENDTO

SENDTO is similar to SEND, except that it includes the destination address parameter. You can use the destination address on the SENDTO macro to send datagrams on a UDP socket that is connected or not connected.

Use the **FLAGS** parameter to:

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

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

For stream sockets, data is processed as streams of information with no boundaries separating the data. For example, if a program is required to send 1000 bytes, each SENDTO macro call can send any number of bytes, up to the entire 1000 bytes, with the number of bytes sent returned in RETCODE. Therefore, programs using stream sockets should place SENDTO in a loop that repeats the macro until all data has been sent.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

```
►─,BUF──=──┬─address──┬─,ALET──=──┬─address──┬─,NAME──=──┬─address──┬───────────►
           ├─*indaddr─┤           ├─*indaddr─┤           ├─*indaddr─┤
           └─(reg)────┘           └─(reg)────┘           └─(reg)────┘

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

►──┬────────────────────────────────────┬──┬─────────────────────────────┬────────►
   └─,FLAGS──=──┬─'MSG_OOB'───────┬─────┘  ├─,ECB=──┬─address──┬─────────┤
                ├─'MSG_DONTROUTE'─┤        │        ├─*indaddr─┤         │
                ├─address─────────┤        │        └─(reg)────┘         │
                ├─*indaddr────────┤        └─,REQAREA=──┬─address──┬─────┘
                └─(reg)───────────┘                     ├─*indaddr─┤
                                                        └─(reg)────┘

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

Keyword
Description

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

NBYTE
Input parameter. A value, or the address of a fullword binary number specifying the number of bytes to transmit.

BUF
Input parameter. The address of the data being transmitted. The length of **BUF** must be at least as long as the value of **NBYTE**.

ALET
Optional input parameter. A fullword binary field containing the **ALET** of **BUF**. The default is 0 (primary address space).

If a nonzero ALET is specified, the ALET must represent a valid entry in the dispatchable unit access list (DU-AL) for the task issuing this call. Note that ALETs can only be specified for synchronous socket calls (for example, ECB/REQAREA not specified). An exception to this is an ALET representing a SCOPE=COMMON data space.

NAME
Input parameter. The address of the target.

Field **Description**

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

PORT
A halfword binary field containing the port number bound to the socket.

IP-ADDRESS
A fullword binary field containing the 32-bit internet address of the socket.

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

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

See “Appendix B. Return Codes” on page 547, for information about **ERRNO** return codes.

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

Value Description

0 or >0
A successful call. The value is set to the number of bytes transmitted.

–1 Check **ERRNO** for an error code

FLAGS Input parameter. **FLAGS** can be a literal value or a fullword binary field:

Literal Value	Binary Value	Description
'MSG_OOB'	1	Send out-of-band data. (Stream sockets only.)
'MSG_DONTROUTE'	4	Do not route. Routing is handled by the calling program.

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

SETSOCKOPT

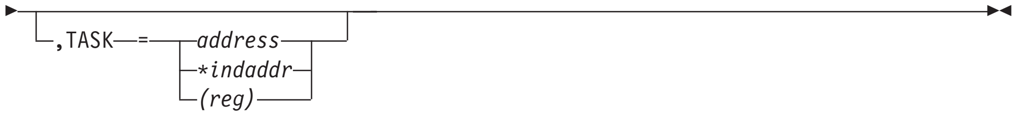
The SETSOCKOPT macro sets the options associated with a socket. SETSOCKOPT can be called only for sockets in the AF_INET domain. It is not supported in the AF_IUCV domain.

The **OPTVAL** and **OPTLEN** parameters are used to pass data used by the particular set command. The **OPTVAL** parameter points to a buffer containing the data needed by the set command. The **OPTLEN** parameter must be set to the size of the data pointed to by **OPTVAL**.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

►─,OPTNAME──=──┬─'SO_BROADCAST'─┬─,OPTVAL──=──┬─address──┬──────────────────────────►
               ├─'SO_KEEPALIVE'─┤             ├─*indaddr─┤
               ├─'SO_LINGER'────┤             └─(reg)────┘
               ├─'SO_OOBINLINE'─┤
               ├─'SO_SNDBUF'────┤
               ├─'SO_RCVBUF'────┤
               ├─'SO_REUSEADDR'─┤
               ├─address────────┤
               ├─*indaddr───────┤
               └─(reg)──────────┘

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

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



Keyword **Description**

S A value, or the address of a halfword binary number specifying the socket sending the data.

OPTLEN Input parameter. A fullword binary number specifying the length of the field specified by **OPTVAL**.

OPTNAME Input parameter. Indicates the following values:

Value **Description**

SO_BROADCAST
Sets the message broadcast capability for the socket. If SO_BROADCAST is set, the program can send broadcast messages over the socket to destinations that can receive datagram messages. SO_BROADCAST has no meaning for stream sockets.

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 processes data that has not been transmitted when a CLOSE macro is issued for the socket. This option has meaning only for stream sockets.

- When **SO_LINGER** is set and CLOSE is called, the calling program is blocked until the data is successfully transmitted or the connection has timed out.
- When SO_LINGER is not set, the CLOSE macro returns without blocking the caller, and TCP/IP continues to attempt to send data for a specified time. This usually allows sufficient time to complete the data transfer. Use of the SO_LINGER option does not guarantee successful completion because TCP/IP only waits the amount of time specified in **OPTVAL** for SO_LINGER.

The default is DISABLED.

SO_OOBINLINE
Sets the ability to receive out-of-band data. This option has meaning only for stream sockets.

- When SO_OOBINLINE is set, out-of-band data is placed in the normal data input queue as it is received, and is available to a RECV or a RECVFROM call even if the OOB flag is not set in the RECV or the RECVFROM macro.
- When SO_OOBINLINE is turned off, out-of-band data is placed in the priority data input queue as it is received and is available to a RECV or a RECVFROM only when the OOB flag is set in the RECV or the RECVFROM macro.

The default is DISABLED.

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

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

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

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

SO_REUSEADDR

Toggles local address reuse. The default is disabled. This alters the normal algorithm used in the bind() call.

The normal bind() call algorithm allows each internet address and port combination to be bound only once. If the address and port have been bound already, a subsequent bind() will fail and result error EADDRINUSE.

After the 'SO_REUSEADDR' option is active, the following situations are supported:

- A server can bind() the same port multiple times as long as every invocation uses a different local IP address, and the wildcard address INADDR_ANY is used only one time per port.

- A server with active client connections can be restarted and can bind to its port, without having to close all of the client connections.
- For datagram sockets, multicasting is supported so multiple bind() calls can be made to the same class D address and port number.

OPTVAL
Input parameter. Contains data about the option specified in **OPTNAME**.

- **OPTVAL** is a 32-bit binary number for all values of **OPTNAME**, except SO_LINGER. Set **OPTVAL** to a nonzero positive value to enable the option. set **OPTVAL** to zero to disable the option.
- For SO_LINGER, **OPTVAL** is:

```
ONOFF    DS   F          ON OR OFF
LINGER   DS   F          TIME IN SECONDS
```

Set ONOFF to a nonzero value to enable the option and set it to zero to disable the option. Set the LINGER value to the time in seconds that TCP/IP lingers after the CLOSE macro is issued.

ERRNO
Output parameter.

A fullword binary field. If **RETCODE** is negative, **OPTVAL** contains an error number.See “Appendix B. Return Codes” on page 547, for information about **ERRNO** return codes.

RETCODE
Output parameter.

A fullword binary field that returns one of the following:

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

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

The **OPTVAL** and **OPTLEN** parameters are used to pass data used by the particular set command. The **OPTVAL** parameter points to a buffer containing the data needed by the set command. It is optional and can be set to the NULL pointer, if data is not needed by the command. The **OPTLEN** parameter must be set to the size of the data pointed to by **OPTVAL**.

SHUTDOWN

One way to terminate a network connection is to issue a CLOSE macro that attempts to complete all outstanding data transmission requests prior to breaking the connection. The SHUTDOWN macro can be used to close one-way traffic while completing data transfer in the other direction. The HOW parameter determines the direction of the traffic to shutdown.

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

If you issue SHUTDOWN for a socket that currently has outstanding socket calls pending, see Table 3 on page 35 to determine the effects of this operation on the outstanding socket calls.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

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

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

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

Keyword **Description**

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

HOW
Input parameter. A fullword binary field specifying the shutdown method.

Value	**Description**
0	Ends further receive operations.
1	Ends further send operations.
2	Ends further send and receive operations.

ERRNO
Output parameter. A fullword binary field. If **RETCODE** is negative, this field contains an error number. See "Appendix B. Return Codes" on page 547, for information about **ERRNO** return codes.

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

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

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

SOCKET

The SOCKET macro creates an endpoint for communication and returns a socket descriptor representing the endpoint. Different types of sockets provide different communication services.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

```
►►──EZASMI──TYPE=SOCKET──,AF──=──┬─'INET'───┬──,SOCTYPE──=──┬─'STREAM'───┬──────────►
                                 ├─address──┤               ├─'DATAGRAM'─┤
                                 ├─*indaddr─┤               ├─'RAW'──────┤
                                 └─(reg)────┘               ├─address────┤
                                                            ├─*indaddr───┤
                                                            └─(reg)──────┘

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

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

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

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

Keyword **Description**

AF Input parameter. Specifies the literal INET, which indicates the internet or TCP/IP. AF can also indicate a fullword binary number specifying the address family. For TCP/IP the value is always 2, indicating AF_INET.

Note: IUCV sockets are not supported by the macro API.

SOCTYPE Input parameter. A fullword binary field set to the type of socket required. The types are:

Value **Description**

1 or 'STREAM'
Stream sockets provide sequenced, two-way byte streams that are reliable and connection-oriented. They support a mechanism for out-of-band data. This is the normal type for TCP/IP.

2 or 'DATAGRAM'
Datagram sockets provide datagrams, which are connectionless messages of a fixed maximum length whose reliability is not guaranteed. Datagrams can be corrupted, received out of order, lost, or delivered multiple times. This type is supported only in the AF_INET domain.

3 or 'RAW'
Raw sockets provide the interface to internal protocols (such as IP and ICMP).

Note: For SOCK_RAW sockets, the port number must be 0.

ERRNO Output parameter. A fullword binary field. If **RETCODE** is negative, this field contains an error number. See “Appendix B. Return Codes” on page 547, for information about **ERRNO** return codes.

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

Value **Description**

> or = 0
Contains the new socket descriptor

–1 Check **ERRNO** for an error code

NS Optional input. A value or the address of a halfword binary number specifying the socket number for the new socket. If a socket number is not specified, the interface assigns one.

PROTO Input parameter. A fullword binary number specifying the protocol supported. **PROTO** only applies to raw sockets and should be set to 0 for TCP/IP. **PROTO** numbers are found in the *hlq*.etc.proto data set.

ECB or REQAREA

Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA

A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

ERROR

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

TASK

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

PROTO specifies a particular protocol to be used with the socket. In most cases, a single protocol exists to support one type of socket in a domain (not true with raw sockets). If **PROTO** is set to 0, the system selects the default protocol number for the domain and socket type requested. The **PROTO** defaults are TCP for stream sockets and UDP for datagram sockets. There is no default for raw sockets.

SOCK_STREAM sockets model duplex byte streams. They provide reliable, flow-controlled connections between peer applications. Stream sockets are either active or passive. Active sockets are used by clients who initiate connection requests with CONNECT. By default, SOCKET creates active sockets. Passive sockets are used by servers to accept connection requests with the CONNECT macro. An active socket is transformed into a passive socket by binding a name to the socket with the BIND macro and by indicating a willingness to accept connections with the LISTEN macro. Once a socket is passive, it cannot be used to initiate connection requests.

In the AF_INET domain, the BIND macro, applied to a stream socket, lets the application specify the networks from which it is willing to accept connection requests. The application can fully specify the network interface by setting the internet address field in the address structure to the internet address of a network interface. Alternatively, the application can set the address in the name structure to zeros to indicate that it wants to receive connection requests from any network.

Once a connection has been established between stream sockets, the data transfer macros READ, WRITE, SEND, RECV, SENDTO, and RECVFROM can be used. Usually, the READ-WRITE or SEND-RECV pairs are used for sending data on stream sockets.

SOCK_DGRAM sockets are used to model datagrams. They provide connectionless message exchange without guarantees of reliability. Messages sent have a maximum size. Datagram sockets are not supported in the AF_IUCV domain.

The active or passive concepts for stream sockets do not apply to datagram sockets. Servers must still call BIND to name a socket and to specify from which network interfaces it wants to receive datagrams. Wildcard addressing, as described for stream sockets, also applies to datagram sockets. Because datagram sockets are connectionless, the LISTEN macro has no meaning for them and must not be used.

After an application receives a datagram socket, it can exchange datagrams using the SENDTO and RECVFROM macros. If the application goes one step further by calling CONNECT and fully specifying the name of the peer with which all messages are exchanged, then the other data transfer macros READ, WRITE, SEND, and RECV can be used as well. For more information about placing a socket into the connected state, see “CONNECT” on page 338.

Datagram sockets allow message broadcasting to multiple recipients. Setting the destination address to a broadcast address depends on the network interface (address class and whether subnets are used).

SOCK_RAW sockets supply an interface to lower layer protocols, such as IP. You can use this interface to bypass the transport layer when you need direct access to lower layer protocols. Raw sockets are also used to test new protocols. Raw sockets are not supported in the AF_IUCV domain.

Raw sockets are connectionless and data transfer is the same as for datagram sockets. You can also use the CONNECT macro to specify a peer socket in the same way that is previously described for datagram sockets.

Outgoing datagrams have an IP header prefixed to them. Your program receives incoming datagrams with the IP header intact. You can set and inspect IP options by using the SETSOCKOPT and GETSOCKOPT macros.

Use the CLOSE macro to deallocate sockets.

TAKESOCKET

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

Notes:

1. When TAKESOCKET is issued, a new socket descriptor is returned in **RETCODE**. You should use this new socket descriptor in later macros such as GETSOCKOPT, which require the S (socket descriptor) parameter.
2. Both concurrent servers and iterative servers use this interface. An iterative server handles one client at a time. A concurrent server receives connection requests from multiple clients and creates subtasks that process the client requests. When a subtask is created, the concurrent server gets a new socket, passes the new socket to the subtask, and dissociates itself from the connection. The TCP/IP Listener program is an example of a concurrent server.

The following requirements apply to this call:

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

Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

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

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

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

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

Keyword **Description**

CLIENT
Input parameter. The client data returned by the GETCLIENTID macro.

Field **Description**

DOMAIN
Input parameter. A fullword binary number set to the domain of the program that is giving the socket. For TCP/IP the value is always 2, indicating AF_INET.

NAME An eight-byte character field set to the MVS address space identifier of the program giving the socket.

TASK Input parameter. Specifies an eight-byte character field.

This field must match the value of the SUBTASK parameter on the INITAPI for the MVS task that issued the GIVESOCKET request.

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

SOCRECV Input parameter. A halfword binary field containing the socket descriptor number assigned by the application that called GIVESOCKET.

ERRNO Output parameter. A fullword binary field. If **RETCODE** is negative, this field contains an error number. See "Appendix B. Return Codes" on page 547, for information about **ERRNO** return codes.

RETCODE Output parameter. A fullword binary field.

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

NS Input parameter. A value or a halfword binary number specifying the socket descriptor number for the new socket. If a number is not specified, the interface assigns one.

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB
A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA
A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA
A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

TASK

The TASK macro allocates a task storage area addressable to all socket users communicating across a particular connection. Most commonly this is done by assigning one connection to each MVS subtask. If more than one module is using sockets within a connection or task, it is your responsibility to provide the task storage address to each user. Each program using sockets should define task storage using the instruction EZASMI TYPE=TASK with STORAGE=DSECT.

If this macro is not named, the default name EZASMTIE is assumed.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

```
►►──EZASMI──TYPE=TASK──,STORAGE──=──┬─DSECT─┬──────────────────────────────►◄
                                    └─CSECT─┘
```

Keyword
Description

STORAGE
Input parameter. Defines one of the following storage definitions:

Keyword
Description

DSECT
Generates a DSECT.

CSECT
Generates an in-line storage definition that can be used within a CSECT or as a part of a larger DSECT.

TERMAPI

The TERMAPI macro ends the session created by the INITAPI macro.

Note: The INITAPI and TERMAPI macros must be issued under the same task.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked

Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

Keyword **Description**

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

WRITE

The WRITE macro writes data on a connected socket. The WRITE macro is similar to the SEND macro except that it does not have the control flags that can be used with SEND.

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

For stream sockets, the data is processed as streams of information with no boundaries separating the data. For example, if you want to send 1000 bytes of data, each call to the write macro can send one byte, ten bytes, or the entire 1000 bytes. You should place the WRITE macro in a loop that cycles until all of the data has been sent.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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

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

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

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

Keyword **Description**

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

NBYTE Input parameter. A value, or the address of a fullword binary field specifying the number of bytes of data to transmit.

BUF The address of the data being transmitted. The length of **BUF** must be at least as long as the value of **NBYTE**.

ALET Optional input parameter. A fullword binary field containing the **ALET** of **BUF**. The default is 0 (primary address space).

If a nonzero ALET is specified, the ALET must represent a valid entry in the dispatchable unit access list (DU-AL) for the task issuing this call. Note that ALETs can only be specified for synchronous socket calls (for example, ECB/REQAREA not specified). An exception to this is an ALET representing a SCOPE=COMMON data space.

ERRNO Output parameter. A fullword binary field. If RETCODE is negative, this field contains an error number. See “Appendix B. Return Codes” on page 547, for information about **ERRNO** return codes.

RETCODE Output parameter. A fullword binary field.

Value	**Description**
>0	A successful call. The value is set to the number of bytes transmitted.
0	Connection partner has closed connection.
–1	Check **ERRNO** for an error code

ECB or REQAREA
Input parameter. This parameter is required if you are using APITYPE=3. It points to a 104-byte field containing:

For ECB

A four-byte **ECB** posted by TCP/IP when the macro completes.

For REQAREA

A four-byte user token (set by you) that is presented to your exit when the response to this function request is complete.

For ECB/REQAREA

A 100-byte storage field used by the interface to save the state information.

Note: This storage must not be modified until the macro function has completed and the **ECB** has been posted or the asynchronous exit has been driven.

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

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

This macro writes up to **NBYTE** bytes of data. If there is not enough available buffer space for the socket data to be transmitted, and the socket is in blocking mode, WRITE blocks the caller until additional buffer space is available. If the socket is in nonblocking mode, WRITE returns a -1 and sets **ERRNO** to 35 (EWOULDBLOCK). See “FCNTL” on page 234 or “IOCTL” on page 262 for a description of how to set the nonblocking mode.

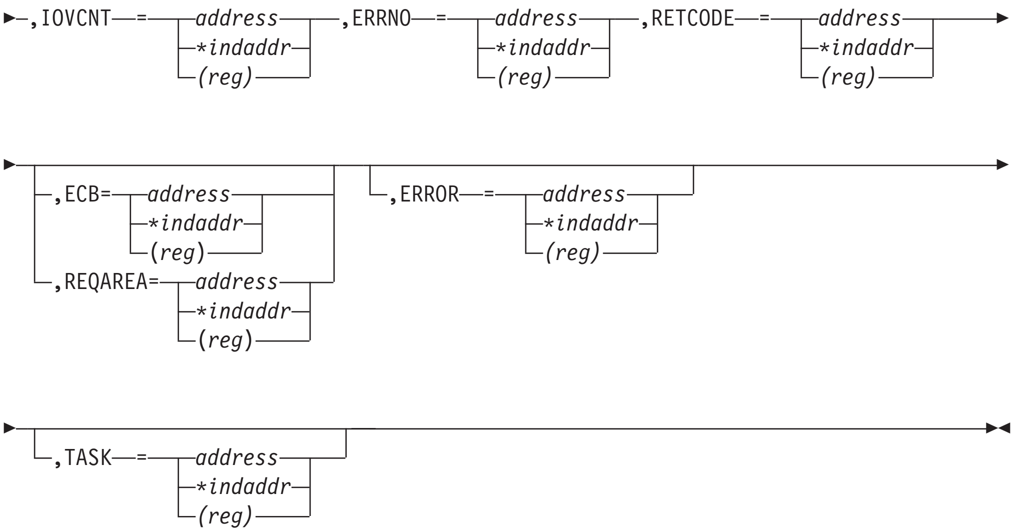
WRITEV

The WRITEV function writes data on a socket from a set of buffers.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

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



Keyword **Description**

S
Input parameter. A value, or the address of a halfword binary number specifying the descriptor of the socket from which the data is to be written.

IOV
Input parameter. An array of three fullword structures with the number of structures equal to the value in IOVCNT and the format of the structures as follows:

Fullword 1
Input parameter. The address of a data buffer.

Fullword 2
Input parameter. The **ALET** for this buffer. If the buffer is in the primary address space, this should be zeros.

If a nonzero ALET is specified, the ALET must represent a valid entry in the dispatchable unit access list (DU-AL) for the task issuing this call. Note that ALETs can only be specified for synchronous socket calls (for example, ECB/REQAREA not specified). An exception to this is an ALET representing a SCOPE=COMMON data space.

Fullword 3
Input parameter. The length of the data buffer referenced in Fullword 1.

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

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

RETCODE
Output parameter. A fullword binary field.

Value	Description
>0	A successful call. The value is set to the number of bytes transmitted.
0	Connection partner has closed connection.

–1 Check **ERRNO** for an error code

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

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

Macro Interface Assembler Language Sample Programs

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

The following sample programs are included:

Program	**Description**
EZASOKAS	Sample macro interface server program.
EZASOKAC	Sample macro interface client program.

EZASOKAS Sample Program

The EZASOKAS program is a server program that shows you how to use the following calls provided by the macro socket interface:

- INITAPI
- SOCKET
- GETHOSTID
- BIND
- LISTEN
- ACCEPT
- READ
- WRITE
- CLOSE
- TERMAPI

```
EZASOKAS CSECT
EZASOKAS AMODE ANY
EZASOKAS RMODE ANY
*        PRINT NOGEN
************************************************************************
*                                                                      *
*   MODULE NAME:  EZASOKAS Sample server program                       *
*                                                                      *
*   LANGUAGE:  Assembler                                               *
*                                                                      *
*   ATTRIBUTES: NON-REUSABLE                                           *
*                                                                      *
*   REGISTER USAGE:                                                    *
*        R1  =                                                         *
*        R2  =                                                         *
*        R3  = BASE REG 1                                              *
*        R4  = BASE REG 2 (UNUSED)                                     *
*        R5  = FUTURE BASE REG?                                        *
*        R6  = TEMP                                                    *
*        R7  = RETURN REG                                              *
*        R8  =                                                         *
*        R9  = A(WORK AREA)                                            *
*        R10 =                                                         *
*        R11 =                                                         *
*        R12 =                                                         *
*        R13 = SAVE AREA                                               *
*        R14 =                                                        *
*        R15 =                                                        *
*                                                                     *
*   INPUT: NONE                                                       *
*   OUTPUT: WTO results of each test case                             *
*                                                                     *
***********************************************************************
         GBLB  &amp;amp;TRACE  ASSEMBLER VARIABLE TO CONTROL TRACE GENERATION
&amp;amp;TRACE   SETB  1       1=TRACE ON  0=TRACE OFF
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
*---------------------------------------------------------------------*
* START OF EXECUTABLE CODE                                            *
*---------------------------------------------------------------------*
         USING *,R3,R4            TELL ASSEMBLER OF OTHERS
         SAVE (14,12),T,*
         LR    R3,R15             COPY EP REG TO FIRST BASE
         LA    R5,2048            GET R5 HALFWAY THERE
         LA    R5,2048(R5)        GET R5 THERE
         LA    R4,0(R5,R3)        GET R4 THERE
         LA    R12,12
         ST    R1,PARMADDR        SAVE ADDRESS OF PARAMETER LIST
         L     R1,0(R1)           GET POINTER
         LH    R1,0(R1)           GET LENGTH
*        STC   R1,TRACE           USE IT AS FLAG
         L     R7,=A(SOCSAVE)     GET NEW SAVE AREA
         ST    R7,8(R13)          SAVE ADDRESS OF NEW SAVE AREA
         ST    R13,4(R7)          COMPLETE SAVE AREA CHAIN
         LR    R13,R7             NOW SWAP THEM
         L     R9,=A(MYCB)        POINT TO THE CONTROL BLOCK
         USING MYCB,R9            TELL ASSEMBLER
*---------------------------------------------------------------------*
*   BUILD MESSAGE FOR CONSOLE
*---------------------------------------------------------------------*
*                                 INITIALIZE MESSAGE TEXT FIELDS
LOOP     EQU   *
         MVC   MSGNUM(8),SUBTASK  WHO I AM
         MVC   TYPE,MSGSTART      MOVE 'STARTED' TO MESSAGE
*
         MVC   MSGRSLT1,MSGSUCC   ...SUCCESSFUL TEXT
         MVC   MSGRSLT2,BLANK35
*
         STM   R14,R12,12(R13)    JUST FOR DEBUGGING
         BAL   R14,WTOSUB         --&amp;gt; DO STARTING WTO
***********************************************************************
*                                                                     *
*        Issue INITAPI to connect to interface                        *
*                                                                     *
***********************************************************************
         POST  ECB,1              NEXT IS ALWAYS SYNCH
         MVI   SYNFLAG,1          MOVE A 1 FOR ASYNC
         MVC   TYPE,MINITAPI      MOVE 'INITAPI' TO MESSAGE
*
```

```
         EZASMI TYPE=INITAPI,     Issue INITAPI Macro                  X
               SUBTASK=SUBTASK,   SPECIFY SUBTASK IDENTIFIER           X
               MAXSOC=MAXSOC,     SPECIFY MAXIMUM NUMBER OF SOCKETS    X
               MAXSNO=MAXSNO,     (HIGHEST SOCKET NUMBER ASSIGNED)     X
               ERRNO=ERRNO,       (Specify ERRNO field)                X
               RETCODE=RETCODE,   (Specify RETCODE field)              X
               APITYPE=APITYPE,   (SPECIFY APITYPE FIELD)              X
               ERROR=ERROR,       ABEND IF ERROR ON MACRO              X
               ASYNC=('EXIT',MYEXIT)  (SPECIFY AN EXIT)
*              IDENT=IDENT,       TCP ADDR SPACE AND MY ADDR SPACE
*              ASYNC=('ECB')      (SPECIFY ECBS)
*
         BAL   R14,RCCHECK        --&amp;gt; DID IT WORK?
***********************************************************************
*                                                                     *
*        Issue SOCKET Macro to obtain a socket descriptor             *
*                  *** INET and STREAM ***                            *
*                                                                     *
***********************************************************************
         MVC   TYPE,MSOCKET       MOVE 'SOCKET' TO MESSAGE
*
         EZASMI TYPE=SOCKET,      Issue SOCKET Macro                   X
               AF='INET',         INET or IUCV                         X
               SOCTYPE='STREAM',  STREAM(TCP) DATAGRAM(UDP) or RAW     X
               ERRNO=ERRNO,       (Specify ERRNO field)                X
               RETCODE=RETCODE,   (Specify RETCODE field)              X
               REQAREA=REQAREA,   IN CASE WE ARE DOING EXITS OR ECBS   X
               ERROR=ERROR        Abend if Macro error
*
         BAL   R14,RCCHECK        CHECK FOR SUCCESSFUL CALL
*
*---------------------------------------------------------------------*
*        Get socket descriptor number
*---------------------------------------------------------------------*
         STH   R8,S               SAVE RETCODE (=SOCKET DESCRIPTOR)
***********************************************************************
*                                                                     *
*        ISSUE GETHOSTID CALL                                         *
*                                                                     *
***********************************************************************
         MVC   TYPE,=CL8'GETHOSTI'     'GETHOSTI' TO MESSAGE
         EZASMI TYPE=GETHOSTID,RETCODE=RETCODE,ERRNO=ERRNO,            X
               REQAREA=REQAREA    IN CASE WE ARE DOING EXITS OR ECBS
         BAL   R14,RCCHECK        CHECK FOR SUCCESSFUL CALL
         ST    R8,ADDR            SAVE OUR ID
***********************************************************************
*                                                                     *
*        Issue BIND socket                                            *
*                                                                     *
***********************************************************************
         MVC   TYPE,MBIND         MOVE 'BIND' TO MESSAGE
         MVC   PORT(2),PORTS      Load STREAM port #
         MVC   ADDRESS(4),ADDR    Load MVS1 internet address
*
         EZASMI TYPE=BIND,        Issue Macro                          X
               S=S,               STREAM                               X
               NAME=NAME,         (SOCKET NAME STRUCTURE)              X
               ERRNO=ERRNO,       (Specify ERRNO field)                X
               RETCODE=RETCODE,   (Specify RETCODE field)              X
               REQAREA=REQAREA,   IN CASE WE ARE DOING EXITS OR ECBS   X
               ERROR=ERROR        Abend if Macro error
*
         BAL   R14,RCCHECK        CHECK FOR SUCCESSFUL CALL
***********************************************************************
*                                                                     *
*        Issue LISTEN - Backlog = 5                                   *
*                                                                     *
```

```
***********************************************************************
         MVC   TYPE,MLISTEN       MOVE 'LISTEN' TO MESSAGE
*
         EZASMI TYPE=LISTEN,      Issue Macro                          X
               S=S,               STREAM                               X
               BACKLOG=BACKLOG,   BACKLOG                              X
               ERRNO=ERRNO,       (Specify ERRNO field)                X
               RETCODE=RETCODE,   (Specify RETCODE field)              X
               REQAREA=REQAREA,   IN CASE WE ARE DOING EXITS OR ECBS   X
               ERROR=ERROR        Abend if Macro error
*
         BAL   R14,RCCHECK        CHECK FOR SUCCESSFUL CALL
```

LISTEN indicates a readiness to accept client connection requests, and creates a connection request queue of length BACKLOG to queue incoming connection requests. Once full, additional connection requests are rejected.

```
***********************************************************************
*                                                                     *
*        Issue ACCEPT - Block until connection from peer              *
*                                                                     *
***********************************************************************
         MVC   TYPE,MACCEPT       MOVE 'ACCEPT' TO MESSAGE
         MVC   PORT(2),PORTS      Load STREAM port #
         MVC   ADDRESS(4),ADDR    Load MVS1 internet address
*
         EZASMI TYPE=ACCEPT,      Issue Macro                          X
               S=S,               STREAM                               X
               NAME=NAME,         (SOCKET NAME STRUCTURE)              X
               ERRNO=ERRNO,       (Specify ERRNO field)                X
               RETCODE=RETCODE,   (Specify RETCODE field)              X
               REQAREA=REQAREA,   IN CASE WE ARE DOING EXITS OR ECBS   X
               ERROR=ERROR        Abend if Macro error
*
         BAL   R14,RCCHECK        CHECK FOR SUCCESSFUL CALL
* Message RESULTS text
         STH   R8,SOCDESCA        SAVE RETCODE (SOCKET DESCRIPTOR)
***********************************************************************
*                                                                     *
*        Issue READ - Read data and store in buffer                   *
*                                                                     *
***********************************************************************
         MVC   TYPE,MREAD         MOVE 'READ  ' TO MESSAGE
*
         EZASMI TYPE=READ,        Issue Macro                          X
               S=SOCDESCA,        ACCEPT SOCKET                        X
               NBYTE=NBYTE,       SIZE OF BUFFER                       X
               BUF=BUF,           (BUFFER)                             X
               ERRNO=ERRNO,       (Specify ERRNO field)                X
               RETCODE=RETCODE,   (Specify RETCODE field)              X
               REQAREA=REQAREA,   IN CASE WE ARE DOING EXITS OR ECBS   X
               ERROR=ERROR        Abend if Macro error
*
         BAL   R14,RCCHECK        CHECK FOR SUCCESSFUL CALL
         MVC   MSGRSLT1,MSGBUFF
         MVC   MSGRSLT2,BUF
         BAL   R14,WTOSUB         --&amp;gt; PRINT IT
*
*
***********************************************************************
*                                                                     *
*        Issue WRITE - Write data from buffer                         *
*                                                                     *
***********************************************************************
         MVC   TYPE,MWRITE        MOVE 'WRITE ' TO MESSAGE
*
         EZASMI TYPE=WRITE,       Issue Macro                          X
```

```
               S=SOCDESCA,        ACCEPT Socket                        X
               NBYTE=NBYTE,       SIZE OF BUFFER                       X
               BUF=BUF,           (BUFFER)                             X
               ERRNO=ERRNO,       (Specify ERRNO field)                X
               RETCODE=RETCODE,   (Specify RETCODE field)              X
               REQAREA=REQAREA,   IN CASE WE ARE DOING EXITS OR ECBS   X
               ERROR=ERROR        Abend if Macro error
*
         BAL   R14,RCCHECK        CHECK FOR SUCCESSFUL CALL
************************************************************************
*                                                                      *
*        Issue CLOSE for ACCEPT socket                                 *
*                                                                      *
************************************************************************
         MVC   TYPE,MCLOSE        MOVE 'CLOSE' TO MESSAGE
*
         EZASMI TYPE=CLOSE,       Issue Macro                          X
               S=SOCDESCA,        ACCEPT                               X
               ERRNO=ERRNO,       (Specify ERRNO field)                X
               RETCODE=RETCODE,   (Specify RETCODE field)              X
               REQAREA=REQAREA,   IN CASE WE ARE DOING EXITS OR ECBS   X
               ERROR=ERROR        Abend if Macro error
*
         MVC   MSGRSLT2,BLANK35
         BAL   R14,RCCHECK        CHECK FOR SUCCESSFUL CALL
*
************************************************************************
*                                                                      *
*        Terminate Connection to API                                   *
*                                                                      *
************************************************************************
         MVC   TYPE,MTERMAPI      MOVE 'TERMAPI' TO MESSAGE
*
         POST  ECB,1              FOLLOWING IS ALWAYS SYNCH
         EZASMI TYPE=TERMAPI      Issue EZASMI Macro for Termapi
*----------------------------------------------------------------------*
* Message RESULTS text
         MVC   MSGRSLT2,BLANK35
*
         BAL   R14,RCCHECK        --&amp;gt; CHECK RC
*----------------------------------------------------------------------*
*        Issue console message for task termination
*----------------------------------------------------------------------*
         MVC   TYPE,MSGEND        Move 'ENDED' to message
*
         MVC   MSGRSLT1,MSGSUCC   ...SUCCESSFUL text
         MVC   MSGRSLT2,BLANK35
*
         BAL   R14,WTOSUB
         LA    R14,1              CONSTANT
         AH    R14,APITYPE        ADD
         STH   R14,APITYPE        STORE
         CH    R14,=H'3'          COMPARE
*        BE    LOOP               --&amp;gt; LETS DO IT AGAIN!
*----------------------------------------------------------------------*
*        Return to Caller
*----------------------------------------------------------------------*
         L     R13,4(R13)
         RETURN (14,12),T,RC=0
WTOSUB   EQU   *
         LR    R7,R14             COPY RETURN REG
         MVC   MSGCMD(8),TYPE
         WTO   TEXT=MSG           WRITE MESSAGE TO OPERATOR
         BR    R7                 --&amp;gt; RETURN TO CALLER
         CNOP  2,4
*        USES R6,R7,R8         RETCODE RETURNED IN R8
RCCHECK  EQU   *
```

```
         LR    R7,R14             COPY TO REAL RETURN REG
         MVC   MSGRSLT1,MSGSUCC   ...SUCCESS TEXT
         L     R6,RETCODE
         LTR   R6,R6
         BM    NOWAIT
         CLI   SYNFLAG,0          PLAIN CASE?
         BE    NOWAIT             --&amp;gt; SKIP IT
         MVC   KEY+14(8),SUBTASK
         MVC   KEY+23(8),TYPE
KEY      WTO   'WAIT: XXXXXXXX XXXXXXXX'
         WAIT  ECB=ECB
NOWAIT   EQU   *
*        LA    R15,ECB
*        ST    R15,ECB
         ST    R9,ECB             MAKE THIS THE TOKEN AGAIN
         L     R6,RETCODE         CHECK FOR SUCCESSFUL CALL
         LTR   R8,R6              SAVE A COPY
*
         BNL   CONT00
*
         MVC   MSGRSLT1,MSGFAIL   ...FAIL TEXT
CONT00   EQU   *
*
*----------------------------------------------------------------------*
*        FORMAT THE RETCODE= -XXXXXXX ERRNO= XXXXXXX MSG RESULTS
*        ***&amp;gt; R6 = RETCODE VALUE ON ENTRY
*----------------------------------------------------------------------*
         MVC   MSGRTCT,MSGRETC    ' RETCODE= '
         MVI   MSGRTCS,C'+'
         LTR   R6,R6
         BNM   NOTM               --&amp;gt;
         MVI   MSGRTCS,C'-'       MOVE SIGN WHICH IS ALWAYS MINUS
NOTM     EQU   *
         MVC   MSGERRT,MSGERRN    ' ERRNO= '
*
         CVD   R6,DWORK           CONVERT IT TO DECIMAL
         UNPK  MSGRTCV,DWORK+4(4) UNPACK IT
         OI    MSGRTCV+6,X'F0'    CORRECT THE SIGN
*
         L     R6,ERRNO           GET ERRNO VALUE
         CVD   R6,DWORK           CONVERT IT TO DECIMAL
         UNPK  MSGERRV,DWORK+4(4) UNPACK IT
         OI    MSGERRV+6,X'F0'    CORRECT THE SIGN
*
         MVC   MSGRSLT2(35),MSGRTCD
*
         SR    R6,R6              CLEAR OUT...
         ST    R6,RETCODE            RETCODE AND...
         ST    R6,ERRNO                 ERRNO
*
*
         CLI   TRACE,0
         BNE   NOTRACE
         LR    R14,R7             GIVE HIM RETURN REG
         B     WTOSUB             --&amp;gt; DO WTO
NOTRACE  EQU   *
         BR    R7                 --&amp;gt; RETURN TO CALLER
SYNFLAG  DC    H'0'               DEFAULT TO SYN
TRACE    DC    H'0'               DEFAULT TO TRACE
MYEXIT   DC   A(MYEXIT1,SUBTASK)
MYEXIT1  SAVE (14,12),T,*
         LR   R2,R15
         USING MYEXIT1,R2
         LM   R8,R9,0(R1)            GET TWO TOKENS
         MVC  EXKEY+14(8),0(R8)      TELL WHO
         MVC  EXKEY+23(8),TYPE       TELL WHAT
EXKEY    WTO 'EXIT: XXXXXXXX XXXXXXXX'
```

```
         POST ECB,1
         RETURN (14,12),T,RC=0
         DROP  R2
*----------------------------------------------------------------------*
*        ABEND PROGRAM AND GET DUMP
*----------------------------------------------------------------------*
ERROR    ABEND 1,DUMP
*----------------------------------------------------------------------*
* CONSTANTS USED TO RUN PROGRAM                                        *
*----------------------------------------------------------------------*
EZASMGW  EZASMI TYPE=GLOBAL,      Storage definition for GWA           X
               STORAGE=CSECT
*---------------------*
* INITAPI macro parms *
*---------------------*
SUBTASK  DC    CL8'EZASOKAS'      SUBTASK PARM VALUE
MAXSOC   DC    H'10'              MAXSOC PARM VALUE
APITYPE  DC    H'2'               OR A 3
MAXSNO   DC    F'0'               (HIGHEST SOCKET DESCRIPTOR AVAILABLE)
IDENT    DC    0CL16' '
         DC    CL8'       '       NAME OF TCP TO WHICH CONNECTING
         DC    CL8'SOC401CB'      MY ADDR SPACE NAME
*----------------------------------------------------------------------*
* SOCKET macro parms *
*--------------------*
S        DC    H'0'               SOCKET DESCRIPTOR FOR STREAM
*----------------------------------------------------------------------*
* BIND MACRO PARMS   *
*--------------------*
         CNOP  0,4
NAME     DC    0CL16' '           SOCKET NAME STRUCTURE
         DC    AL2(2)             FAMILY
PORT     DC    H'0'
ADDRESS  DC    F'0'
         DC    XL8'00'            RESERVED
ADDR     DC    AL1(14),AL1(0),AL1(0),AL1(0) Internet Address
PORTS    DC    H'11007'
*----------------------------------------------------------------------*
* LISTEN PARMS       *
*--------------------*
BACKLOG  DC    F'5'               BACKLOG
*----------------------------------------------------------------------*
* READ MACRO PARMS   *
*--------------------*
NBYTE    DC    F'50'              SIZE OF BUFFER
SOCDESCA DC    H'0'               SOCKET DESCRIPTOR FROM ACCEPT
BUF      DC    CL50' THIS SHOULD NEVER APPEAR!!! &amp;colon;-('
*----------------------------------------------------------------------*
* WTO FRAGMENTS *
*---------------*
MINITAPI DC    CL8'INITAPI'
MSOCKET  DC    CL8'SOCKET'
MBIND    DC    CL8'BIND'
MACCEPT  DC    CL8'ACCEPT'
MLISTEN  DC    CL8'LISTEN'
MREAD    DC    CL8'READ'
MWRITE   DC    CL8'WRITE'
MCLOSE   DC    CL8'CLOSE'
MTERMAPI DC    CL8'TERMAPI'
MSGSTART DC    CL8' STARTED'
MSGEND   DC    CL8' ENDED  '
MSGBUFF  DC    CL10' BUFFER:  '                    ...
MSGSUCC  DC    CL10' SUCCESS  '     Command results...
MSGFAIL  DC    CL10' FAIL: ( '                    ...
MSGRETC  DC    CL10' RETCODE= '                    ...
MSGERRN  DC    CL10' ERRNO=   '                    ...
BLANK35  DC    CL35' '
```

```
*----------------------------------------------------------------------*
* ERROR NUMBER / RETURN CODE FIELDS *
*-----------------------------------*
*----------------------------------------------------------------------*
* MESSAGE AREA *
*--------------*
MSG      DC    0F'0'              MESSAGE AREA
         DC    AL2(MSGE-MSGNUM)   LENGTH OF MESSAGE
MSGNUM   DC    CL10'EZASOKAS&amp;colon;'    'EZASOKASXX&amp;colon;'
MSGCMD   DC    CL8' '             COMMAND ISSUED
MSGRSLT1 DC    CL10' '            COMMAND RESULTS (SUCC, PASS, FAIL)
MSGRSLT2 DC    CL35' '            RETURNED VALUES
MSGE     EQU   *                  End of message
*----------------------------------------------------------------------*
* MESSAGE RESULTS AREAS (fill in and move to MSGRSLT2) *
*------------------------------------------------------*
MSGRTCD  DC    0CL35' '           GENERAL RETURNED VALUE
MSGRTCT  DC    CL10' RETCODE='    ' RETCODE= '
MSGRTCS  DC    CL1' '             '-' (NEGATIVE SIGN)
MSGRTCV  DC    CL7' '             RETURNED VALUE (RETCODE)
MSGERRT  DC    CL10' ERRNO='      ' ERRNO=   '
MSGERRV  DC    CL7' '             RETURNED VALUE (ERRNO)
*----------------------------------------------------------------------*
PARMADDR DC    A(0)               PARM ADDRESS SAVE AREA
DWORK    DC    D'0'               WORK AREA
         LTORG ,
*----------------------------------------------------------------------*
*----------------------------------------------------------------------*
* REG/SAVEAREA *
*--------------*
SOCSAVE  DC    9D'0'              SAVE AREA
         CNOP  0,8
MYCB     EQU   *                  MY CONTROL BLOCK
REQAREA  EQU   *
ECB      DC    A(ECB)             SELF POINTER
         DC    CL100'WORK AREA'
MYTIE    EZASMI TYPE=TASK,STORAGE=CSECT     TIE
TYPE     DC    CL8'TYPE'
ERRNO    DC    F'0'
RETCODE  DC    F'0'
MYNEXT   DC    A(MYCB)            NEXT IN CHAIN FOR MULTIPLES
         CNOP  0,8
MYLEN    EQU   *-MYCB
MYCB2    EQU   *
         ORG   *+MYLEN
         CNOP  0,8
         DC    CL8'&amp;amp;SYSDATE'
         DC    CL8'&amp;amp;SYSTIME'
         END
```

EZASOKAC Sample Program

The EZASOKAC program is a client module that shows you how to use the following calls provided by the macro socket interface:

- INITAPI
- SOCKET
- CONNECT
- GETPEERNAME
- WRITE
- SHUTDOWN
- WRITE
- READ
- TERMAPI

```
EZASOKAC CSECT
EZASOKAC AMODE 24
EZASOKAC RMODE 24
         PRINT NOGEN
***********************************************************************
*                                                                     *
*   MODULE NAME:  EZASOKAC - THIS IS A VERY SIMPLE CLIENT             *
*                                                                     *
*   LANGUAGE:  ASSEMBLER                                              *
*                                                                     *
*   ATTRIBUTES: NON-REUSEABLE                                         *
*                                                                     *
*   REGISTER USAGE:                                                   *
*        R1  =                                                        *
*        R2  =                                                        *
*        R3  = BASE REG 1                                             *
*        R4  = BASE REG 2 (UNUSED)                                    *
*        R5  = FUTURE BASE?                                           *
*        R6  = TEMP                                                   *
*        R7  = RETURN REG                                             *
*        R8  =                                                        *
*        R9  = A(WORK AREA)                                           *
*        R10 =                                                        *
*        R11 =                                                        *
*        R12 =                                                        *
*        R13 = SAVE AREA                                              *
*        R14 =                                                        *
*        R15 =                                                        *
*                                                                     *
*   INPUT: ANY PARM TURNS TRACE OFF, NO PARM IS NOISY MODE            *
*   OUTPUT: WTO RESULTS OF EACH TEST CASE IF TRACING                  *
*           RETURN CODE IS 0 WHETHER IT CONNECTS OR NOT!              *
*                                                                     *
***********************************************************************
         GBLB  &TRACE  ASSEMBLER VARIABLE TO CONTROL TRACE GENERATION
&TRACE   SETB  1       1=TRACE ON  0=TRACE OFF
R0       EQU   0
R1       EQU   1
R2       EQU   2
R3       EQU   3
R4       EQU   4
R5       EQU   5
R6       EQU   6
R7       EQU   7
R8       EQU   8
R9       EQU   9
R10      EQU   10
R11      EQU   11
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
*---------------------------------------------------------------------*
* START OF EXECUTABLE CODE                                            *
*---------------------------------------------------------------------*
         USING *,R3,R4            TELL ASSEMBLER OF OTHERS
         SAVE (14,12),T,*
         LR    R3,R15             COPY EP REG TO FIRST BASE
         LA    R5,2048            GET R5 HALFWAY THERE
         LA    R5,2048(R5)        GET R5 THERE
         LA    R4,0(R5,R3)        GET R4 THERE
         LA    R12,12             JUST FOR FUN!
         ST    R1,PARMADDR        SAVE ADDRESS OF PARAMETER LIST
         L     R1,0(R1)           GET POINTER
         LH    R1,0(R1)           GET LENGTH
*        STC   R1,TRACE           USE IT AS FLAG
         L     R7,=A(SOCSAVE)     GET NEW SAVE AREA
         ST    R7,8(R13)          SAVE ADDRESS OF NEW SAVE AREA
         ST    R13,4(R7)          COMPLETE SAVE AREA CHAIN
         LR    R13,R7             NOW SWAP THEM
         L     R9,=A(MYCB)        POINT TO THE CONTROL BLOCK
         USING MYCB,R9            TELL ASSEMBLER
*---------------------------------------------------------------------*
*   BUILD MESSAGE FOR CONSOLE
*---------------------------------------------------------------------*
*                                 INITIALIZE MESSAGE TEXT FIELDS
LOOP     EQU   *
         MVC   MSGNUM(8),SUBTASK  WHO I AM
         MVC   TYPE,MSGSTART      MOVE 'STARTED' TO MESSAGE
*
         MVC   MSGRSLT1,MSGSUCC   ...SUCCESSFUL TEXT
         MVC   MSGRSLT2,BLANK35
*
         STM   R14,R12,12(R13)    JUST FOR DEBUGGING
         BAL   R14,WTOSUB         --> DO STARTING WTO
***********************************************************************
*                                                                     *
*        Issue INITAPI to connect to interface                        *
*                                                                     *
***********************************************************************
         MVC   TYPE,MINITAPI      MOVE 'INITAPI' TO MESSAGE
*
         POST  ECB,1              FOLLOWING IS SYNC ONLY
         MVI   SYNFLAG,0          MOVE A 1 FOR ASYNCH
         EZASMI TYPE=INITAPI,     ISSUE INITAPI MACRO                  X
               SUBTASK=SUBTASK,   SPECIFY SUBTASK IDENTIFIER           X
               MAXSOC=MAXSOC,     SPECIFY MAXIMUM NUMBER OF SOCKETS    X
               MAXSNO=MAXSNO,     (HIGHEST SOCKET NUMBER ASSIGNED)     X
               ERRNO=ERRNO,       (Specify ERRNO field)                X
               RETCODE=RETCODE,   (Specify RETCODE field)              X
               APITYPE=APITYPE,   (SPECIFY APITYPE FIELD)              X
               ERROR=ERROR        Abend if error on macro
*              IDENT=IDENT,       TCP ADDR SPACE AND MY ADDR SPACE
*
*              ASYNC=('ECB'),     (SPECIFY TO USE ECBS)
*              ASYNC=('EXIT',MYEXIT)  (SPECIFY TO USE EXITS)
         BAL   R14,RCCHECK        --> CHECK RESULTS
***********************************************************************
*                                                                     *
*        Issue SOCKET Macro to obtain a socket descriptor             *
*                  *** INET and STREAM ***                            *
*                                                                     *
***********************************************************************
         MVC   TYPE,MSOCKET       MOVE 'SOCKET' TO MESSAGE
*
         EZASMI TYPE=SOCKET,      Issue SOCKET Macro                   X
               AF='INET',         INET or IUCV                         X
               SOCTYPE='STREAM',  STREAM(TCP) DATAGRAM(UDP) or RAW     X
               ERRNO=ERRNO,       (Specify ERRNO field)                X
               RETCODE=RETCODE,   (Specify RETCODE field)              X
               REQAREA=REQAREA,   FOR EXITS (AND ECBS)                 X
               ERROR=ERROR        Abend if Macro error
*
         BAL   R14,RCCHECK        --> CHECK RESULTS
         STH   R8,S               SAVE RETCODE (=SOCKET DESCRIPTOR)
         LTR   R8,R8              CHECK IT
         BM    DOSHUTDO           --> WE ARE DONE!
***********************************************************************
*                                                                     *
*        ISSUE GETHOSTID CALL                                         *
*                                                                     *
***********************************************************************
         MVC    TYPE,=CL8'GETHOSTI'
         POST   ECB,1             FOLLOWING IS SYNC ONLY
         EZASMI TYPE=GETHOSTID,RETCODE=RETCODE,ERRNO=ERRNO
         BAL   R14,RCCHECK        --> CHECK RESULTS
         ST    R8,ADDR
***********************************************************************
*                                                                     *
*        Issue CONNECT Socket                                         *
*                                                                     *
***********************************************************************
         MVC   TYPE,MCONNECT      MOVE 'CONNECT' TO MESSAGE
         MVC   PORT(2),PORTS      Load STREAM port #
*
*
         MVC   ADDRESS(4),ADDR    LOAD OUR INTERNET ADDRESS
*
         EZASMI TYPE=CONNECT,     Issue Macro                          X
               S=S,               STREAM                               X
               NAME=NAME,         SOCKET NAME STRUCTURE                X
               ERRNO=ERRNO,       (Specify ERRNO field)                X
               RETCODE=RETCODE,   (Specify RETCODE field)              X
               REQAREA=REQAREA,   FOR EXITS (AND ECBS)                 X
               ERROR=ERROR        Abend if Macro error
*
         BAL   R14,RCCHECK        --> CHECK RC
         LTR   R8,R8              RECHECK IT
         BM    DOSHUTDO           --> WE ARE DONE
***********************************************************************
*                                                                     *
*        Issue GETPEERNAME                                            *
*                                                                     *
***********************************************************************
         MVC   TYPE,MGETPEER      MOVE 'GTPEERN' TO MESSAGE
*
         EZASMI TYPE=GETPEERNAME, Issue Macro                          X
               S=S,               STREAM                               X
               NAME=NAME,         (SOCKET NAME STRUCTURE)              X
               ERRNO=ERRNO,       (Specify ERRNO field)                X
               RETCODE=RETCODE,   (Specify RETCODE field)              X
               REQAREA=REQAREA,   FOR EXITS (AND ECBS)                 X
               ERROR=ERROR        Abend if Macro error
*
         BAL   R14,RCCHECK        --> CHECK RC
***********************************************************************
*                                                                     *
*        Issue WRITE - Write data from buffer                         *
*                                                                     *
***********************************************************************
         MVC   TYPE,MWRITE        MOVE 'WRITE ' TO MESSAGE
*
         EZASMI TYPE=WRITE,       Issue Macro                          X
               S=S,               STREAM SOCKET                        X
               NBYTE=NBYTE,       SIZE OF BUFFER                       X
               BUF=BUF,           BUFFER                               X
               ERRNO=ERRNO,       (Specify ERRNO field)                X
               RETCODE=RETCODE,   (Specify RETCODE field)              X
               REQAREA=REQAREA,   FOR EXITS (AND ECBS)                 X
               ERROR=ERROR        Abend if Macro error
*
         BAL   R14,RCCHECK        --> CHECK RC
***********************************************************************
*                                                                     *
*        Issue SHUTDOWN - HOW = 1 (end communication TO socket)       *
*                                                                     *
***********************************************************************
DOSHUTDO EQU   *
         MVC   HOW(4),=F'1'
*
         BAL   R14,SHUTSUB        --> SHUTDOWN
```

```
*
         BAL   R14,RCCHECK        --> CHECK RC
***********************************************************************
*                                                                     *
*        Issue READ - Read data and store in buffer                   *
*                                                                     *
***********************************************************************
         MVC   TYPE,MREAD         MOVE 'READ  ' TO MESSAGE
*
         EZASMI TYPE=READ,        Issue Macro                          X
               S=S,               STREAM SOCKET                        X
               NBYTE=NBYTE,       SIZE OF BUFFER                       X
               BUF=BUF2,          (BUFFER)                             X
               ERRNO=ERRNO,       (Specify ERRNO field)                X
               RETCODE=RETCODE,   (Specify RETCODE field)              X
               REQAREA=REQAREA,   FOR EXITS (AND ECBS)                 X
               ERROR=ERROR        Abend if Macro error
*
         BAL   R14,RCCHECK        --> CHECK RC
         MVC   MSGRSLT1,MSGBUFF   TITLE
         MVC   MSGRSLT2,BUF2      MOVE THE DATA
         BAL   R14,WTOSUB         --> PRINT IT
***********************************************************************
*                                                                     *
*        Issue SHUTDOWN - HOW = 0 (end communication FROM socket)     *
*                                                                     *
***********************************************************************
         MVC   HOW(4),=F'0'
*
         BAL   R14,SHUTSUB        --> SHUTDOWN
*
         BAL   R14,RCCHECK        --> CHECK RC
***********************************************************************
*                                                                     *
*        Terminate Connection to API                                  *
*                                                                     *
***********************************************************************
         MVC   TYPE,MTERMAPI      MOVE 'TERMAPI' TO MESSAGE
*
         POST  ECB,1              FOLLOWING IS SYNC ONLY
         EZASMI TYPE=TERMAPI      Issue EZASMI Macro for Termapi
*
         BAL   R14,RCCHECK        --> CHECK RC
*---------------------------------------------------------------------*
*        Issue console message for task termination
*---------------------------------------------------------------------*
         MVC   TYPE,MSGEND        Move 'ENDED' to message
*
         MVC   MSGRSLT1,MSGSUCC   ...SUCCESSFUL text
         MVC   MSGRSLT2,BLANK35
         BAL   R14,WTOSUB         --> DO WTO
         LA    R14,1              CONSTANT
         AH    R14,APITYPE        ADD
         STH   R14,APITYPE        STORE
         CH    R14,=H'3'          COMPARE
*        BE    LOOP               --> LETS DO IT AGAIN!
*
*---------------------------------------------------------------------*
*        Return to Caller
*---------------------------------------------------------------------*
         L     R13,4(R13)
         RETURN (14,12),T,RC=0
WTOSUB   EQU   *
         LR    R7,R14             SAVE RETURN REG
         MVC   MSGCMD,TYPE        COPY COMMAND
         WTO   TEXT=MSG
         BR    R7                 --> RETURN
```

```
*
SHUTSUB  EQU   *
         LR    R7,R14
         MVC   TYPE,MSHUTDOW      MOVE 'SHUTDOW' TO MESSAGE
*
         EZASMI TYPE=SHUTDOWN,    Issue Macro                          X
               S=S,               STREAM                               X
               HOW=HOW,           End communication in both directions X
               ERRNO=ERRNO,       (Specify ERRNO field)                X
               RETCODE=RETCODE,   (Specify RETCODE field)              X
               REQAREA=REQAREA,   FOR EXITS (AND ECBS)                 X
               ERROR=ERROR        Abend if Macro error
*
         BR    R7                 --> RETURN TO CALLER
*---------------------------------------------------------------------*
*        ABEND PROGRAM AND GET DUMP TO DEBUG!
ERROR    ABEND 1,DUMP
         CNOP  2,4
*        USES R6,R7,R8         RETCODE RETURNED IN R8
RCCHECK  EQU   *
         LR    R7,R14             COPY TO REAL RETURN REG
         MVC   MSGRSLT1,MSGSUCC   ...SUCCESS TEXT
         L     R6,RETCODE
         LTR   R6,R6
         BM    NOWAIT
         CLI   SYNFLAG,0          PLAIN CASE?
         BE    NOWAIT             --> SKIP IT
         MVC   KEY+14(8),SUBTASK
         MVC   KEY+23(8),TYPE
KEY      WTO   'WAIT: XXXXXXXX XXXXXXXX'
         WAIT  ECB=ECB
NOWAIT   EQU   *
*        LA    R15,ECB
*        ST    R15,ECB
         ST    R9,ECB             MAKE THIS THE TOKEN AGAIN
         L     R6,RETCODE         CHECK FOR SUCCESSFUL CALL
         LTR   R8,R6              SAVE A COPY
*
         BNL   CONT00
*
         MVC   MSGRSLT1,MSGFAIL   ...FAIL TEXT
CONT00   EQU   *
*
*---------------------------------------------------------------------*
*        FORMAT THE RETCODE= -XXXXXXX ERRNO= XXXXXXX MSG RESULTS
*        ***> R6 = RETCODE VALUE ON ENTRY
*---------------------------------------------------------------------*
         MVC   MSGRTCT,MSGRETC    ' RETCODE= '
         MVI   MSGRTCS,C'+'
         LTR   R6,R6
         BNM   NOTM               -->
         MVI   MSGRTCS,C'-'       MOVE SIGN WHICH IS ALWAYS MINUS
NOTM     EQU   *
         MVC   MSGERRT,MSGERRN    ' ERRNO= '
*
         CVD   R6,DWORK           CONVERT IT TO DECIMAL
         UNPK  MSGRTCV,DWORK+4(4) UNPACK IT
         OI    MSGRTCV+6,X'F0'    CORRECT THE SIGN
*
         L     R6,ERRNO           GET ERRNO VALUE
         CVD   R6,DWORK           CONVERT IT TO DECIMAL
         UNPK  MSGERRV,DWORK+4(4) UNPACK IT
         OI    MSGERRV+6,X'F0'    CORRECT THE SIGN
*
         MVC   MSGRSLT2(35),MSGRTCD
*
         SR    R6,R6              CLEAR OUT...
         ST    R6,RETCODE            RETCODE AND...
         ST    R6,ERRNO                 ERRNO
*
*
         CLI   TRACE,0
         BNE   NOTRACE
         LR    R14,R7             GIVE HIM RETURN REG
         B     WTOSUB             --> DO WTO
NOTRACE  EQU   *
         BR    R7                 --> RETURN TO CALLER
SYNFLAG  DC    H'0'               DEFAULT TO SYN
TRACE    DC    H'0'               DEFAULT TO TRACE
MYEXIT   DC   A(MYEXIT1,SUBTASK)
MYEXIT1  SAVE (14,12),T,*
         LR   R2,R15
         USING MYEXIT1,R2
         LM   R8,R9,0(R1)            GET TWO TOKENS
         MVC  EXKEY+14(8),0(R8)      TELL WHO
         MVC  EXKEY+23(8),TYPE       TELL WHAT
EXKEY    WTO 'EXIT: XXXXXXXX XXXXXXXX'
         POST ECB,1
         RETURN (14,12),T,RC=0
         DROP  R2
*---------------------------------------------------------------------*
* ELEMENTS USED TO RUN PROGRAM                                       *
*---------------------------------------------------------------------*
EZASMGW  EZASMI TYPE=GLOBAL,      STORAGE DEFINITION FOR GWA           X
               STORAGE=CSECT
*---------------------*
* INITAPI macro parms *
*---------------------*
SUBTASK  DC    CL8'EZASOKAC'      SUBTASK PARM VALUE
IDENT    DC    0CL16' '
         DC    CL8'TCPV32'        DEFAULT TO FIRST ONE AVAILABLE
         DC    CL8'EZASOKAC'      MY ADDR SPACE NAME OR JOBNAME
MAXSNO   DC    F'0'               (HIGHEST SOCKET DESCRIPTOR AVAILABLE)
MAXSOC   DC    H'10'              MAXSOC PARM VALUE
APITYPE  DC    H'2'               OR PUT A 3 HERE
*---------------------------------------------------------------------*
* SOCKET macro parms *
*--------------------*
S        DC    H'0'               SOCKET DESCRIPTOR FOR STREAM
*---------------------*
* CONNECT MACRO PARMS *
*---------------------*
         CNOP  0,4
NAME     DC    0CL16' '           SOCKET NAME STRUCTURE
         DC    AL2(2)             FAMILY
PORT     DC    H'0'
ADDRESS  DC    F'0'
         DC    XL8'0'             RESERVED
ADDR     DC    AL1(14),AL1(0),AL1(0),AL1(0) Internet Address
PORTS    DC    H'11007'
*ORTS    DC    H'43'
*---------------------*
* WRITE MACRO PARMS   *
*---------------------*
NBYTE    DC    F'50'              SIZE OF BUFFER
BUF      DC    CL50' THIS IS FROM EZASOKAC!' BUFFER FOR WRITE
*------------------------*
* SHUTDOWN MACRO PARMS   *
*------------------------*
HOW      DC    F'2'               END COMMUNICATION TO- AND FROM-SOCKET
*---------------------*
* READ MACRO PARMS    *
*---------------------*
BUF2     DC    CL50'BUF2'         BUFFER FOR READ
```

```
*----------------*
*----------------*
MINITAPI DC    CL8'INITAPI'
MSOCKET  DC    CL8'SOCKET'
MCONNECT DC    CL8'CONNECT'
MGETPEER DC    CL8'GETPEERN'
MREAD    DC    CL8'READ'
MWRITE   DC    CL8'WRITE'
MSHUTDOW DC    CL8'SHUTDOWN'
MTERMAPI DC    CL8'TERMAPI'
MSGSTART DC    CL8' STARTED'
MSGEND   DC    CL8' ENDED  '
MSGSUCC  DC    CL10' SUCCESS  '     Command results...
MSGFAIL  DC    CL10' FAIL: ( '                    ...
MSGRETC  DC    CL10' RETCODE= '                    ...
MSGERRN  DC    CL10' ERRNO=   '                    ...
MSGBUFF  DC    CL10' BUFFER:  '                    ...
BLANK35  DC    CL35' '
*----------------------------------------------------------------------*
* MESSAGE AREA *
*--------------*
MSG      DC    0F'0'              MESSAGE AREA
         DC    AL2(MSGE-MSGNUM)   LENGTH OF MESSAGE
MSGNUM   DC    CL10'EZASOKAC:'    'EZASOKAC: '
MSGCMD   DC    CL8' '             COMMAND ISSUED
MSGRSLT1 DC    CL10' '            COMMAND RESULTS (SUCC, PASS, FAIL)
MSGRSLT2 DC    CL35' '            RETURNED VALUES
MSGE     EQU   *                  End of message
*----------------------------------------------------------------------*
* MESSAGE RESULTS AREAS (fill in and move to MSGRSLT2) *
*------------------------------------------------------*
*
MSGRTCD  DC    0CL35' '           GENERAL RETURNED VALUE
MSGRTCT  DC    CL10' RETCODE='    ' RETCODE= '
MSGRTCS  DC    CL1' '             '-' (NEGATIVE SIGN)
MSGRTCV  DC    CL7' '             RETURNED VALUE (RETCODE)
MSGERRT  DC    CL10' ERRNO='      ' ERRNO=   '
MSGERRV  DC    CL7' '             RETURNED VALUE (ERRNO)
DWORK    DC    D'0'               WORK AREA
PARMADDR DC    A(0)               PARM ADDRESS SAVE AREA
         LTORG
*----------------------------------------------------------------------*
* REG/SAVEAREA *
*--------------*
SOCSAVE  DC    9D'0'              SAVE AREA
*----------------------------------------------------------------------*
         CNOP  0,8
MYCB     EQU   *                  MY CONTROL BLOCK
REQAREA  EQU   *
ECB      DC    A(ECB)             SELF POINTER
         DC    CL100'WORK AREA'
MYTIE    EZASMI TYPE=TASK,STORAGE=CSECT     TIE
TYPE     DC    CL8'TYPE'
ERRNO    DC    F'0'
RETCODE  DC    F'0'
MYNEXT   DC    A(MYCB)            NEXT IN CHAIN FOR MULTIPLES
         CNOP  0,8
MYLEN    EQU   *-MYCB
MYCB2    EQU   *
         ORG   *+MYLEN
         CNOP  0,8
         DC    CL8'&SYSDATE'
         DC    CL8'&SYSTIME'
         END
```

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

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

- Environmental Restrictions and Programming Requirements
- CALL instruction API
- Understanding COBOL, Assembler, and PL/1 call formats
- Call interface PL/1 sample programs

Environmental Restrictions and Programming Requirements

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

- SRB mode

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

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

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

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

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

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

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

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

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

CALL Instruction Application Programming Interface (API)

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

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

Notes:

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

   ```
   DCL EZASOKET ENTRY OPTIONS(RETCODE,ASM,INTER) EXT;
   ```
4. A C run-time library is required when you use the GETHOSTBYADDR or GETHOSTBYNAME call.
5. The entry point for the CICS Sockets Extended module (EZASOKET) is within the EZACICAL module, therefore EZACICAL should be included explicitly in your linkedit JCL. If not included, you could experience problems, such as the CICS region waiting for the socket calls to complete.

Understanding COBOL, Assembler, and PL/1 Call Formats

This API is invoked by calling the EZASOKET program and performs the same functions as the C language calls. The parameters look different because of the differences in the programming languages.

COBOL Language Call Format

```
►►──CALL 'EZASOKET' USING SOC-FUNCTION──parm1, parm2, ...──ERRNO RETCODE.────────►◄
```

SOC-FUNCTION
A 16-byte character field, left-justified and padded on the right with blanks. Set to the name of the call. SOC-FUNCTION is case specific. It must be in uppercase.

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

ERRNO
If RETCODE is negative, there is an error number in ERRNO. This field is used in most, but not all, of the calls. It corresponds to the value returned by the tcperror() function in C.

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

Assembler Language Call Format

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

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

PL/1 Language Call Format

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

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

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

ERRNO
If RETCODE is negative, there is an error number in ERRNO. This field is used in most, but not all, of the calls. It corresponds to the value returned by the tcperror() function in C.

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

Converting Parameter Descriptions

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

Figure 46 on page 332 shows examples of storage definition statements for COBOL, PL/1, and assembler language programs.

```
VS COBOL II PIC

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

COBOL PIC

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

PL/1 DECLARE STATEMENT

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

ASSEMBLER DECLARATION

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

Figure 46. Storage Definition Statement Examples

Diagnosing Problems in Applications Using the CALL Instruction API

TCP/IP provides a trace facility that can be helpful in diagnosing problems in applications using the CALL instruction API. The trace is implemented using the TCP/IP Component Trace (CTRACE) SOCKAPI trace option. The SOCKAPI trace option allows all Call instruction socket API calls issued by an application to be traced in the TCP/IP CTRACE. The SOCKAPI trace records include information such as the type of socket call, input, and output parameters and return codes. This trace can be helpful in isolating failing socket API calls and in determining the nature of the error or the history of socket API calls that may the cause of an error. For more information on the SOCKAPI trace option, refer to *OS/390 IBM Communications Server: IP Diagnosis*.

Error Messages and Return Codes

For information about error messages, see *TCP/IP for MVS: Messages and Codes*.

For information about error codes that are returned by TCP/IP, see “Appendix B. Return Codes” on page 547.

Code CALL Instructions

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

ACCEPT

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

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

1. Accepts the first connection on a queue of pending connections

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

Notes:

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 47 on page 334 shows an example of ACCEPT call instructions.

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

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

Figure 47. ACCEPT Call Instructions Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

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 eight bytes of binary zeros. This field is required, but not used.

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix B. Return Codes” on page 547, for information about ERRNO return codes.

RETCODE
If the RETCODE value is positive, the RETCODE value is the new socket number.

If the RETCODE value is negative, check the ERRNO field for an error number.

BIND

In a typical server program, the BIND call follows a SOCKET call and completes the process of creating a new socket.

The BIND call can either specify the required port or let the system choose the port. A listener program should always bind to the same well-known port, so that clients know what socket address to use when attempting to connect.

In the AF_INET domain, the BIND call for a stream socket can specify the networks from which it is willing to accept connection requests. The application can fully specify the network interface by setting the ADDRESS field to the internet address of a network interface. Alternatively, the application can use a *wildcard* to specify that it wants to receive connection requests from any network interface. This is done by setting the ADDRESS field to a fullword of zeros.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 48 shows an example of BIND call instructions.

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

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

Figure 48. BIND Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

Parameter Values Set by the Application

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

S
A halfword binary number specifying the socket descriptor for the socket to be bound.

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

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

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

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

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

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

Parameter Values Returned to the Application

ERRNO
A fullword binary field. If RETCODE is negative, this field contains an error number. See “Appendix B. Return Codes” on page 547, for information about ERRNO return codes.

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

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

CLOSE

The CLOSE call performs the following functions:

- The CLOSE call shuts down a socket and frees all resources allocated to it. If the socket refers to an open TCP connection, the connection is closed.
- The CLOSE call is also issued by a concurrent server after it gives a socket to a child server program. After issuing the GIVESOCKET and receiving notification that the client child has successfully issued a TAKESOCKET, the concurrent server issues the close command to complete the passing of ownership. In high-performance, transaction-based systems the timeout associated with the CLOSE call can cause performance problems. In such systems you should consider the use of a SHUTDOWN call before you issue the CLOSE call. See “SHUTDOWN” on page 396 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 393.
2. A concurrent server differs from an iterative server. An iterative server provides services for one client at a time; a concurrent server receives connection requests from multiple clients and creates child servers that actually serve the clients. When a child server is created, the concurrent

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

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 49 shows an example of CLOSE call instructions.

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


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

Figure 49. CLOSE Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

Parameter Values Set by the Application

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

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

Parameter Values Returned to the Application

ERRNO
A fullword binary field. If RETCODE is negative, this field contains an error number. See “Appendix B. Return Codes” on page 547, for information about ERRNO return codes.

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

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

CONNECT

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

Stream Sockets

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

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

UDP Sockets

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

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

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

The blocking mode of the CONNECT call conditions its operation.

- If the socket is in blocking mode, the CONNECT call blocks the calling program until the connection is established, or until an error is received.
- If the socket is in nonblocking mode the return code indicates whether the connection request was successful.
  - A zero RETCODE indicates that the connection was completed.
  - A nonzero RETCODE with an ERRNO 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 379.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode

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

Figure 50 shows an example of CONNECT call instructions.

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


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

Figure 50. CONNECT Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

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 2for 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 eight-byte reserved field. This field is required, but is not used.

Parameter Values Returned to the Application

ERRNO

A fullword binary field. If RETCODE is negative, this field contains an error number. See “Appendix B. Return Codes” on page 547, 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 363 for another way to control a socket’s blocking mode.

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 51 on page 341 shows an example of FCNTL call instructions.

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


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

Figure 51. FCNTL Call Instruction Example

For equivalent PL/I and assembler language declarations, see "Converting Parameter Descriptions" on page 331.

Parameter Values Set by the Application

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

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

COMMAND
A fullword binary number with the following values.

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

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

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

Parameter Values Returned to the Application

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See "Appendix B. Return Codes" on page 547, 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 358 for a discussion of the use of GIVESOCKET and TAKESOCKET calls.

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 52 shows an example of GETCLIENTID call instructions.

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

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

Figure 52. GETCLIENTID Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

Parameter Values Set by the Application

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

Parameter Values Returned to the Application

CLIENT
A client-ID structure that describes the application that issued the call.

DOMAIN
A fullword binary number specifying the caller's domain. For TCP/IP the value is set to 2 for AF_INET.

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

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

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

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See "Appendix B. Return Codes" on page 547, for information about ERRNO return codes.

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

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

GETHOSTBYADDR

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 53 on page 344 shows an example of GETHOSTBYADDR call instructions.

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

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

Figure 53. GETHOSTBYADDR Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

Parameter Values Set by the Application

SOC-FUNCTION

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

HOSTADDR

A fullword binary field set to the internet address (specified in network byte order) of the host whose name is being sought. See “Appendix B. Return Codes” on page 547, for information about ERRNO return codes.

Parameter Values Returned to the Application

HOSTENT

A fullword containing the address of the HOSTENT structure.

RETCODE

A fullword binary field that returns one of the following:

Value	Description
0	Successful call
–1	An error occurred

GETHOSTBYADDR returns the HOSTENT structure shown in Figure 54 on page 345.

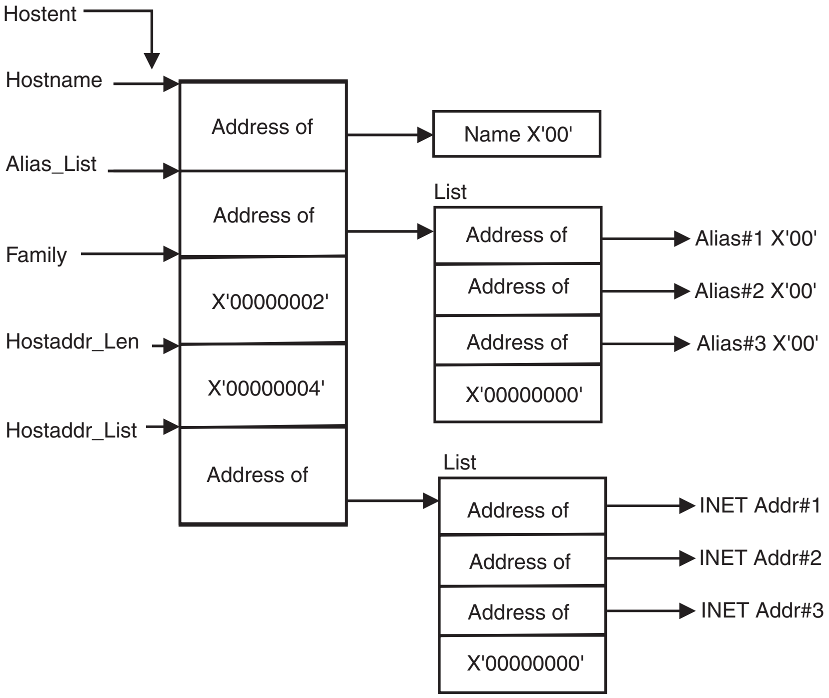


Figure 54. HOSTENT Structure Returned by the GETHOSTBYADDR Call

This structure contains:

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

The HOSTENT structure uses indirect addressing to return a variable number of alias names and internet addresses. If you are coding in PL/1 or assembler language, this structure can be processed in a relatively straight-forward manner. If you are coding in COBOL, this structure may be difficult to interpret. You can use the subroutine EZACIC08 to simplify interpretation of the information returned by the GETHOSTBYADDR and GETHOSTBYNAME calls. For more information about EZACIC08, see “EZACIC08” on page 410.

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 IBM Communications Server: IP Configuration Reference*.
2. The C runtime libraries are required when GETHOSTBYNAME is issued by your program. For CICS, the C runtime library must be included in the link list.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 55 shows an example of GETHOSTBYNAME call instructions.

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

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

Figure 55. GETHOSTBYNAME Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

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

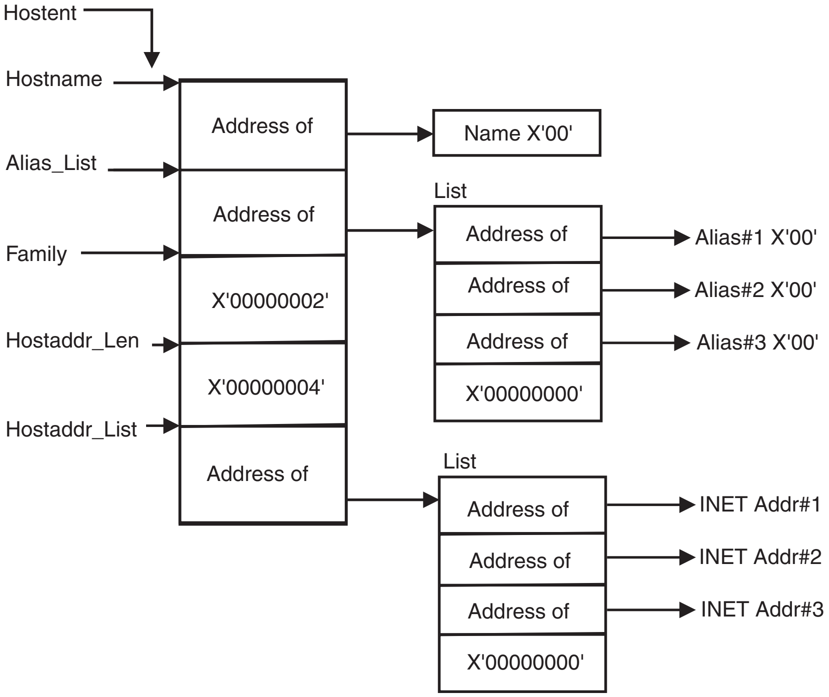


Figure 56. HOSTENT Structure Returned by the GETHOSTYBYNAME Call

GETHOSTBYNAME returns the HOSTENT structure shown in Figure 56. This structure contains:

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

The HOSTENT structure uses indirect addressing to return a variable number of alias names and internet addresses. If you are coding in PL/1 or assembler language, this structure can be processed in a relatively straight-forward manner. If you are coding in COBOL, this structure may be difficult to interpret. You can use the subroutine EZACIC08 to simplify interpretation of the information returned by the GETHOSTBYADDR and GETHOSTBYNAME calls. For more information about EZACIC08, see “EZACIC08” on page 410.

GETHOSTID

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 57 shows an example of GETHOSTID call instructions.

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

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

Figure 57. GETHOSTID Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

Parameter Values Set by the Application

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

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

GETHOSTNAME

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 58 shows an example of GETHOSTNAME call instructions.

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

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

Figure 58. GETHOSTNAME Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

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 “Appendix B. Return Codes” on page 547, 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 *TCP/IP for MVS: Planning and Migration Guide*.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 59 on page 351 shows an example of GETIBMOPT call instructions.

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

PROCEDURE

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

Figure 59. GETIBMOPT Call Instruction Example

Parameter Values Set by the Application

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

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

Parameter Values Returned to the Application

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

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

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

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

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

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

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

The version field is X'0302' for TCP/IP V3R2 for MVS. The version field is X'0304' for TCP/IP CS for OS/390 V2R5. The version field is X'0308' for TCP/IP CS for OS/390 V2R8.

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

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

Figure 60. Example of Name Field

ERRNO

A fullword binary field. If RETCODE is negative, this contains an error number. See “Appendix B. Return Codes” on page 547, for information about ERRNO return codes.

RETCODE

A fullword binary field with the following values:

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

GETPEERNAME

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode

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

Figure 61 shows an example of GETPEERNAME call instructions.

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

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

Figure 61. GETPEERNAME Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

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 eight-byte reserved field. This field is required, but not used.

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix B. Return Codes” on page 547, for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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

GETSOCKNAME

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

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 62 shows an example of GETSOCKNAME call instructions.

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

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

Figure 62. GETSOCKNAME Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

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, zero is returned.

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

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

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix B. Return Codes” on page 547, for information about ERRNO return codes.

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

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

GETSOCKOPT

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

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts

Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 63 shows an example of GETSOCKOPT call instructions.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'GETSOCKOPT'.
    01  S               PIC 9(4) BINARY.
    01  OPTNAME         PIC 9(8) BINARY.
        88  SO-REUSEADDR  VALUE  4.
        88  SO-KEEPALIVE  VALUE  8.
        88  SO-BROADCAST  VALUE  32.
        88  SO-LINGER     VALUE  128.
        88  SO-OOBINLINE  VALUE  256.
        88  SO-SNDBUF     VALUE  4097.
        88  SO-RCVBUF     VALUE  4098.
        88  SO-ERROR      VALUE  4103.
        88  SO-TYPE       VALUE  4104.
    01  OPTVAL          PIC X(16) BINARY.
    01  OPTLEN          PIC 9(8) BINARY.
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

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

Figure 63. GETSOCKOPT Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

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

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

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

- the TCPSENDBufrsize keyword on the TCPCONFIG statement in the PROFILE.TCPIP data set for a TCP socket

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

SO-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 393.

OPTLEN

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

- For all values of OPTNAME except SO-LINGER, OPTLEN will be set to 4 (one fullword).
- For OPTNAME of SO-LINGER, OPTVAL contains two fullwords, so OPTLEN will be set to 8 (two fullwords).

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix B. Return Codes” on page 547, 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.

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 64 on page 360 shows an example of GIVESOCKET call instructions.

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

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

Figure 64. GIVESOCKET Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

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 this field should be set to blanks. When this is done, any MVS address space that requests the socket can have it.

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

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

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

Parameter Values Returned to the Application

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix B. Return Codes” on page 547, for information about ERRNO return codes.

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

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

INITAPI

The INITAPI call connects an application to the TCP/IP interface. Almost all sockets programs that are written in COBOL, PL/I, or assembler language must issue the INITAPI macro before they issue other sockets macros.

The exceptions to this rule are the following calls, which, when issued first, will generate a default INITAPI call.
- GETCLIENTID
- GETHOSTID
- GETHOSTNAME
- GETIBMOPT
- SELECT
- SELECTEX
- SOCKET
- TAKESOCKET

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 65 on page 362 shows an example of INITAPI call instructions.

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

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

Figure 65. INITAPI Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

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 eight-byte character field which should be set to the MVS jobname of the TCP/IP address space with which you are connecting.

ADSNAME

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

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 zero. If you have 50 sockets, they are numbered from zero to 49. If MAXSNO is not specified, the value for MAXSNO is 49.

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix B. Return Codes” on page 547, 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 17 on page 366, for information about REQARG and RETARG.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 66 on page 364 shows an example of IOCTL call instructions.

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

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

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

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

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


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

Figure 66. IOCTL Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

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 67 for the address format.

SIOCGIFBRDADDR

Requests the network interface broadcast address for a given interface name. See the NAME field in Figure 67 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 67.

- 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 67. 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 67 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 17.

Table 17. 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 67 on page 365 for format.)
SIOCGIFBRDADDR X'C020A712'	32	First 16 bytes—interface name. Last 16 bytes—not used	32	Network interface address (See Figure 67 on page 365 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 67 on page 365 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
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 17 for information about REQARG and RETARG.

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix B. Return Codes” on page 547, 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 68 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. Figure 68 shows a COBOL II example for SIOCGIFCONF.

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

Figure 68. COBOL II Example for SIOCGIFCONF

LISTEN

The LISTEN call:

- Completes the bind, if BIND has not already been called for the socket.
- Creates a connection-request queue of a specified length for incoming connection requests.

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

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task

Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 69 shows an example of LISTEN call instructions.

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

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

Figure 69. LISTEN Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

Parameter Values Set by the Application

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

S A halfword binary number set to the socket descriptor.

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

Parameter Values Returned to the Application

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix B. Return Codes” on page 547, 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 407 for a subroutine that will translate ASCII input data to EBCDIC.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 70 shows an example of READ call instructions.

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

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

Figure 70. READ Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

Parameter Values Set by the Application

SOC-FUNCTION

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

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

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

Parameter Values Returned to the Application

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

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See "Appendix B. Return Codes" on page 547, 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.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 71 on page 371 shows an example of READV call instructions.

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

Figure 71. READV Call Instruction Example

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 one.

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

Parameter Values Returned to the Application

ERRNO
A fullword binary field. If RETCODE is negative, this contains an error number. See “Appendix B. Return Codes” on page 547, 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.

RECV

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

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

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

If data is not available for the socket, and the socket is in blocking mode, RECV blocks the caller until data arrives. If data is not available and the socket is in nonblocking mode, RECV returns a –1 and sets ERRNO to 35 (EWOULDBLOCK). See “FCNTL” on page 340 or “IOCTL” on page 363 for a description of how to set nonblocking mode.

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

Note: See “EZACIC05” on page 407 for a subroutine that will translate ASCII input data to EBCDIC.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 72 on page 373 shows an example of RECV call instructions.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'RECV'.
    01  S               PIC 9(4) BINARY.
    01  FLAGS           PIC 9(8) BINARY.
         88  NO-FLAG               VALUE IS 0.
         88  OOB                   VALUE IS 1.
         88  PEEK                  VALUE IS 2.
    01  NBYTE           PIC 9(8) BINARY.
    01  BUF             PIC X(length of buffer).
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

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

Figure 72. RECV Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

Parameter Values Set by the Application

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

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

FLAGS
A fullword binary field with values as follows:

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

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

Parameter Values Returned to the Application

BUF The input buffer to receive the data.

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix B. Return Codes” on page 547, for information about ERRNO return codes.

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

Value Description

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

RECVFROM

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

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

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

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

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

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

If data is not available for the socket, and the socket is in blocking mode, RECVFROM blocks the caller until data arrives. If data is not available and the socket is in nonblocking mode, RECVFROM returns a –1 and sets ERRNO to 35 (EWOULDBLOCK). See "FCNTL" on page 340 or "IOCTL" on page 363 for a description of how to set nonblocking mode.

Note: See "EZACIC05" on page 407 for a subroutine that will translate ASCII input data to EBCDIC.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 73 shows an example of RECVFROM call instructions.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'RECVFROM'.
    01  S               PIC 9(4) BINARY.
    01  FLAGS           PIC 9(8) BINARY.
        88  NO-FLAG                VALUE IS 0.
        88  OOB                    VALUE IS 1.
        88  PEEK                   VALUE IS 2.
    01  NBYTE           PIC 9(8) BINARY.
    01  BUF             PIC X(length of buffer).
    01  NAME.
        03  FAMILY      PIC 9(4) BINARY.
        03  PORT        PIC 9(4) BINARY.
        03  IP-ADDRESS  PIC 9(8) BINARY.
        03  RESERVED    PIC X(8).
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

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

Figure 73. RECVFROM Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

Parameter Values Set by the Application

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

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

FLAGS
A fullword binary field containing flag values as follows:

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

NBYTE
A fullword binary number specifying the length of the input buffer.

Parameter Values Returned to the Application

BUF Defines an input buffer to receive the input data.

NAME A structure containing the address of the socket that sent the data. The structure is:

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

PORT A halfword binary number specifying the port number of the sending socket.

IP-ADDRESS
A fullword binary number specifying the 32-bit internet address of the sending socket.

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

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See "Appendix B. Return Codes" on page 547, for information about ERRNO return codes.

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

Value	**Description**
0	The socket is closed.
>0	A positive return code indicates the number of bytes of data transferred by the read call.
–1	Check ERRNO for an error code.

RECVMSG

The RECVMSG call receives messages on a socket with descriptor S and stores them in an array of message headers. If a datagram packet is too long to fit in the supplied buffers, datagram sockets discard extra bytes.

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 74 on page 377 shows an example of RECVMSG call instructions.

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

     01  FLAGS           PIC 9(8)   BINARY.
         88  NO-FLAG                  VALUE IS 0.
         88  OOB                      VALUE IS 1.
         88  PEEK                     VALUE IS 2.
     01  ERRNO           PIC 9(8)   BINARY.
     01  RETCODE         PIC S9(8)  BINARY.

LINKAGE SECTION.

     01 RECVMSG-IOVECTOR.
        03 IOV1A               USAGE IS POINTER.
           05 IOV1AL              PIC 9(8) COMP.
           05 IOV1L               PIC 9(8) COMP.
        03 IOV2A               USAGE IS POINTER.
           05 IOV2AL              PIC 9(8) COMP.
           05 IOV2L               PIC 9(8) COMP.
        03 IOV3A               USAGE IS POINTER.
           05 IOV3AL              PIC 9(8) COMP.
           05 IOV3L               PIC 9(8) COMP.

     01 RECVMSG-BUFFER1     PIC X(16).
     01 RECVMSG-BUFFER2     PIC X(16).
     01 RECVMSG-BUFFER3     PIC X(16).
     01 RECVMSG-BUFNO       PIC 9(8) COMP.

PROCEDURE

           SET MSG-NAME TO NULLS.
           SET MSG-NAME-LEN TO NULLS.
           SET IOV TO ADDRESS OF RECVMSG-IOVECTOR.
           MOVE 3 TO RECVMSG-BUFNO.
           SET MSG-IOVCNT TO ADDRESS OF RECVMSG-BUFNO.
           SET IOV1A TO ADDRESS OF RECVMSG-BUFFER1.
           MOVE 0 TO MSG-IOV1AL.
           MOVE LENGTH OF RECVMSG-BUFFER1 TO MSG-IOV1L.
           SET IOV2A TO ADDRESS OF RECVMSG-BUFFER2.
           MOVE 0 TO IOV2AL.
           MOVE LENGTH OF RECVMSG-BUFFER2 TO IOV2L.
           SET IOV3A TO ADDRESS OF RECVMSG-BUFFER3.
           MOVE 0 TO IOV3AL.
           MOVE LENGTH OF RECVMSG-BUFFER3 TO IOV3L.
           SET MSG-ACCRIGHTS TO NULLS.
           SET MSG-ACCRIGHTS-LEN TO NULLS.
           MOVE X'00000000' TO FLAGS.
           MOVE SPACES TO RECVMSG-BUFFER1.
           MOVE SPACES TO RECVMSG-BUFFER2.
           MOVE SPACES TO RECVMSG-BUFFER3.

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

Figure 74. RECVMSG Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

Parameter Values Set by the Application

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

MSG On input, a pointer to a message header into which the message is received upon completion of the call.

Field **Description**

NAME On input, a pointer to a buffer where the sender address is stored upon completion of the call.

NAME-LEN
On input, a pointer to the size of the address buffer that is filled in on completion of the call.

IOV On input, a pointer to an array of tripleword structures with the number of structures equal to the value in IOVCNT and the format of the structures as follows:

Fullword 1
A pointer to the address of a data buffer

Fullword 2
Reserved

Fullword 3
A pointer to the length of the data buffer referenced in fullword 1.

In COBOL, the IOV structure must be defined separately in the Linkage section, as shown in the example.

IOVCNT
On input, a pointer to a fullword binary field specifying the number of data buffers provided for this call.

ACCRIGHTS
On input, a pointer to the access rights received. This field is ignored.

ACCRLEN
On input, a pointer to the length of the access rights received. This field is ignored.

FLAGS
A fullword binary field with values as follows:

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

Parameter Values Returned to the Application

ERRNO
A fullword binary field. If RETCODE is negative, this contains an error number. See “Appendix B. Return Codes” on page 547, for information about ERRNO return codes.

RETCODE
A fullword binary field with the following values:

Value	Description
<0	Call returned error. See ERRNO field.
0	Connection partner has closed connection.
>0	Number of bytes read.

SELECT

In a process where multiple I/O operations can occur it is necessary for the program to be able to wait on one or several of the operations to complete.

For example, consider a program that issues a READ to multiple sockets whose blocking mode is set. Because the socket would block on a READ call, only one socket could be read at a time. Setting the sockets nonblocking would solve this problem, but would require polling each socket repeatedly until data became available. The SELECT call allows you to test several sockets and to execute a subsequent I/O call only when one of the tested sockets is ready; thereby ensuring that the I/O call will not block.

To use the SELECT call as a timer in your program, do one of the following:
- Set the read, write, and except arrays to zeros.
- Specify MAXSOC <= 0.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Defining Which Sockets to Test

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

- When a socket is ready to read, one fo the following has occurred:
  - A buffer for the specified sockets contains input data. If input data is available for a given socket, a read operation on that socket will not block.

  - A connection has been requested on that socket.
- When a socket is ready to write, TCP/IP can accommodate additional output data. If TCP/IP can accept additional output for a given socket, a write operation on that socket will not block.
- When an exception condition has occurred on a specified socket it is an indication that a TAKESOCKET has occurred for that socket.

Each socket descriptor is represented by a bit in a bit string. The bit strings are contained in 32-bit fullwords, numbered from right to left. The right-most bit represents socket descriptor zero; 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 408.

Read Operations

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

To test whether any of several sockets is ready for reading, set the appropriate bits in RSNDMSK to one before issuing the SELECT call. When the SELECT call returns, the corresponding bits in the RRETMSK indicate sockets 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 one before issuing the SELECT call. When the SELECT call returns, the corresponding bits in the WRETMSK indicate sockets ready for writing.

Exception Operations

For each socket to be tested, the SELECT call can check for an existing exception condition. Two exception conditions are supported:

- The calling program (concurrent server) has issued a GIVESOCKET command and the target child server has successfully issued the TAKESOCKET call. When this condition is selected, the calling program (concurrent server) should issue CLOSE to dissociate itself from the socket.

- A socket has received out-of-band data. On this condition, a READ will return the out-of-band data ahead of program data.

To test whether any of several sockets have an exception condition, set the ESNDMSK bits representing those sockets to one. When the SELECT call returns, the corresponding bits in the ERETMSK indicate sockets with exception conditions.

MAXSOC Parameter

The SELECT call must test each bit in each string before returning results. For efficiency, the MAXSOC parameter can be used to specify the largest socket descriptor number that needs to be tested for any event type. The SELECT call tests only bits in the range zero through the MAXSOC value.

TIMEOUT Parameter

If the time specified in the TIMEOUT parameter elapses before any event is detected, the SELECT call returns, RETCODE is set to 0.

Figure 75 shows an example of SELECT call instructions.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'SELECT'.
    01  MAXSOC          PIC 9(8) BINARY.
    01  TIMEOUT.
        03  TIMEOUT-SECONDS   PIC 9(8) BINARY.
        03  TIMEOUT-MICROSEC  PIC 9(8) BINARY.
    01  RSNDMSK        PIC X(*).
    01  WSNDMSK        PIC X(*).
    01  ESNDMSK        PIC X(*).
    01  RRETMSK        PIC X(*).
    01  WRETMSK        PIC X(*).
    01  ERETMSK        PIC X(*).
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

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

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

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

Figure 75. SELECT Call Instruction Example

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

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

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 two-word TIMEOUT as follows:

- TIMEOUT-SECONDS, word one of the TIMEOUT field, is the seconds component of the time-out value.
- TIMEOUT-MICROSEC, word two of the TIMEOUT field, is the microseconds component of the time-out value (0—999999).

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

RSNDMSK

A bit string sent to request read event status.

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

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

WSNDMSK

A bit string sent to request write event status.

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

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

ESNDMSK

A bit string sent to request exception event status.

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

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

Parameter Values Returned to the Application

RRETMSK

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

WRETMSK

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

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

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix B. Return Codes” on page 547, 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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 76 on page 384 shows an example of SELECTEX call instructions.

```
 WORKING STORAGE
     01  SOC-FUNCTION    PIC X(16)  VALUE IS 'SELECTEX'.
     01  MAXSOC          PIC 9(8)   BINARY.
     01  TIMEOUT.
         03  TIMEOUT-SECONDS    PIC 9(8) BINARY.
         03  TIMEOUT-MINUTES    PIC 9(8) BINARY.
     01  RSNDMSK        PIC X(*).
     01  WSNDMSK        PIC X(*).
     01  ESNDMSK        PIC X(*).
     01  RRETMSK        PIC X(*).
     01  WRETMSK        PIC X(*).
     01  ERETMSK        PIC X(*).
     01  SELECB          PIC X(4).
     01  ERRNO           PIC 9(8)   BINARY.
     01  RETCODE         PIC S9(8)  BINARY.


where * is the size of the select mask

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

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

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

Figure 76. SELECTEX Call Instruction Example

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 two-word TIMEOUT as follows:

- TIMEOUT-SECONDS, word one of the TIMEOUT field, is the seconds component of the time-out value.
- TIMEOUT-MICROSEC, word two of the TIMEOUT field, is the microseconds component of the time-out value (0—999999).

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

RSNDMSK

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

WSNDMSK

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

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

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

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

Note: The maximum number of ECBs that can be specified in a list is 63.

Parameter Values Returned to the Application

ERRNO
A fullword binary field; if RETCODE is negative, this contains an error number. See “Appendix B. Return Codes” on page 547, for information about ERRNO return codes.

RETCODE
A fullword binary field

Value	Meaning
>0	The number of ready sockets.
0	Either the SELECTEX time limit has expired (ECB value will be zero) or one of the caller’s ECBs has been posted (ECB value will be non-zero and the caller’s descriptor sets will be set to 0). The caller must initialize the ECB values to zero before issuing the SELECTEX macro.
-1	Error; check ERRNO.

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

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

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

SEND

The SEND call sends data on a specified connected socket.

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 and writing network software.

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

For stream sockets, data is processed as streams of information with no boundaries separating the data. For example, if a program is required to send 1000 bytes, each call to this function can send any number of bytes, up to the entire 1000 bytes, with the number of bytes sent returned in RETCODE. Therefore, programs using stream sockets should place this call in a loop, reissuing the call until all data has been sent.

Note: See “EZACIC04” on page 406 for a subroutine that will translate EBCDIC input data to ASCII.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 77 shows an example of SEND call instructions.

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

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

Figure 77. SEND Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

Parameter Values Set by the Application

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

S A halfword binary number specifying the socket descriptor of the socket that is sending data.

FLAGS
A fullword binary field with values as follows:

Literal Value	Binary Value	Description
NO-FLAG	0	No flag is set. The command behaves like a WRITE call.
OOB	1	Send out-of-band data. (Stream sockets only.) Even if the OOB flag is not set, out-of-band data can be read if the SO-OOBINLINE option is set for the socket.
DONT-ROUTE	4	Do not route. Routing is provided by the calling program.

NBYTE
A fullword binary number set to the number of bytes of data to be transferred.

BUF The buffer containing the data to be transmitted. BUF should be the size specified in NBYTE.

Parameter Values Returned to the Application

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix B. Return Codes” on page 547, for information about ERRNO return codes.

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

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

SENDMSG

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

The following requirements apply to this call:

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

Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 78 on page 389 shows an example of SENDMSG call instructions.

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

Figure 78. SENDMSG Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

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.

ACCRIGHTS-LEN
On input, a pointer to the length of the access rights received. This field is ignored.

FLAGS
A fullword field containing the following:

Literal Value	Binary Value	Description
NO-FLAG	0	No flag is set. The command behaves like a WRITE call.
OOB	1	Send out-of-band data. (Stream sockets only.) Even if the OOB flag is not set, out-of-band data can be read if the SO-OOBINLINE option is set for the socket.
DONTROUTE	4	Do not route. Routing is provided by the calling program.

Parameter Values Returned to the Application

ERRNO

A fullword binary field. If RETCODE is negative, this contains an error number. See “Appendix B. Return Codes” on page 547, for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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

SENDTO

SENDTO is similar to SEND, except that it includes the destination address parameter. The destination address allows you to use the SENDTO call to send datagrams on a UDP socket, regardless of whether the socket is connected.

The FLAGS parameter allows you to:

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

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

For stream sockets, data is processed as streams of information with no boundaries separating the data. For example, if a program is required to send 1000 bytes, each call to this function can send any number of bytes, up to the entire 1000 bytes, with the number of bytes sent returned in RETCODE. Therefore, programs using stream sockets should place SENDTO in a loop that repeats the call until all data has been sent.

Note: See “EZACIC04” on page 406 for a subroutine that will translate EBCDIC input data to ASCII.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 79 shows an example of SENDTO call instructions.

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

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

Figure 79. SENDTO Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

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 eight-byte reserved field. This field is required, but not used.

Parameter Values Returned to the Application

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See "Appendix B. Return Codes" on page 547, for information about ERRNO return codes.

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

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

SETSOCKOPT

The SETSOCKOPT call sets the options associated with a socket. SETSOCKOPT can be called only for sockets in the AF_INET domain.

The OPTVAL and OPTLEN parameters are used to pass data used by the particular set command. The OPTVAL parameter points to a buffer containing the data needed by the set command. The OPTVAL parameter is optional and can be set to 0, if data is not needed by the command. The OPTLEN parameter must be set to the size of the data pointed to by OPTVAL.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See "Addressability mode (Amode) considerations" under "Environmental Restrictions and Programming Requirements" on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 80 on page 394 shows an example of SETSOCKOPT call instructions.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'SETSOCKOPT'.
    01  S               PIC 9(4) BINARY.
    01  OPTNAME         PIC 9(8) BINARY.
        88  SO-REUSEADDR  VALUE  4.
        88  SO-KEEPALIVE  VALUE  8.
        88  SO-BROADCAST  VALUE  32.
        88  SO-LINGER     VALUE  128.
        88  SO-OOBINLINE  VALUE  256.
        88  SO-SNDBUF     VALUE  4097.
        88  SO-RCVBUF     VALUE  4098.
    01  OPTVAL          PIC 9(16) BINARY.
    01  OPTLEN          PIC 9(8) BINARY.
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

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

Figure 80. SETSOCKOPT Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

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 reuse. The default is disabled. This alters the normal algorithm used in the bind() call.

The normal bind() call algorithm allows each internet address and port combination to be bound only once. If the address and port have been bound already, a subsequent bind() will fail and result error EADDRINUSE.

After the 'SO_REUSEADDR' option is active, the following situations are supported:

- A server can bind() the same port multiple times as long as every invocation uses a different local IP address, and the wildcard address INADDR_ANY is used only one time per port.
- A server with active client connections can be restarted and can bind to its port, without having to close all of the client connections.
- For datagram sockets, multicasting is supported so multiple bind() calls can be made to the same class D address and port number.

SO-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.

SO_RCVBUF

Sets the size of the data portion of the TCP/IP receive buffer in OPTVAL. The size of the data portion of the receive buffer is protocol-specific.

SO_SNDBUF

Sets the size of the data portion of the TCP/IP send buffer in OPTVAL. The size of the data portion of the send buffer is protocol-specific.

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 “Appendix B. Return Codes” on page 547, 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.

If you issue SHUTDOWN for a socket that currently has outstanding socket calls pending, see Table 3 on page 35 to determine the effects of this operation on the outstanding socket calls.

The following requirements apply to this call:

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

Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 81 shows an example of SHUTDOWN call instructions.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'SHUTDOWN'.
    01  S               PIC 9(4) BINARY.
    01  HOW             PIC 9(8) BINARY.
        88  END-FROM      VALUE  0.
        88  END-TO        VALUE  1.
        88  END-BOTH      VALUE  2.
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

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

Figure 81. SHUTDOWN Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

Parameter Values Set by the Application

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

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

HOW A fullword binary field. Set to specify whether all or part of a connection is to be shut down. The following values can be set:

Value	**Description**
0 (END-FROM)	Ends further receive operations.
1 (END-TO)	Ends further send operations.
2 (END-BOTH)	Ends further send and receive operations.

Parameter Values Returned to the Application

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix B. Return Codes” on page 547, for information about ERRNO return codes.

RETCODE

A fullword binary field that returns one of the following:

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

SOCKET

The SOCKET call creates an endpoint for communication and returns a socket descriptor representing the endpoint.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 82 shows an example of SOCKET call instructions.

```
WORKING STORAGE
    01  SOC-FUNCTION    PIC X(16)  VALUE IS 'SOCKET'.
    01  AF              PIC 9(8) COMP VALUE 2.
    01  SOCTYPE         PIC 9(8) BINARY.
        88  STREAM        VALUE  1.
        88  DATAGRAM      VALUE  2.
        88  RAW           VALUE  3.
    01  PROTO           PIC 9(8) BINARY.
    01  ERRNO           PIC 9(8) BINARY.
    01  RETCODE         PIC S9(8) BINARY.

PROCEDURE
     CALL 'EZASOKET' USING SOC-FUNCTION AF SOCTYPE
                    PROTO ERRNO RETCODE.
```

Figure 82. SOCKET Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

Parameter Values Set by the Application

SOC-FUNCTION

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

AF

A fullword binary field set to the addressing family. For TCP/IP the value is set to 2 for AF_INET.

SOCTYPE
A fullword binary field set to the type of socket required. The types are:

Value **Description**

1 Stream sockets provide sequenced, two-way byte streams that are reliable and connection-oriented. They support a mechanism for out-of-band data.

2 Datagram sockets provide datagrams, which are connectionless messages of a fixed maximum length whose reliability is not guaranteed. Datagrams can be corrupted, received out of order, lost, or delivered multiple times.

3 Raw sockets provide the interface to internal protocols (such as IP and ICMP).

PROTO
A fullword binary field set to the protocol to be used for the socket. If this field is set to 0, the default protocol is used. For streams, the default is TCP; for datagrams, the default is UDP.

PROTO numbers are found in the *hlq*.etc.proto data set.

Parameter Values Returned to the Application

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix B. Return Codes” on page 547, 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 358 for a discussion of the use of GETSOCKET and TAKESOCKET calls.

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.

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

Figure 83 shows an example of TAKESOCKET call instructions.

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

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

Figure 83. TAKESOCKET Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

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 eight-byte character field set to the task identifier of the task that gave the socket.

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

Parameter Values Returned to the Application

ERRNO

A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix B. Return Codes” on page 547 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.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 84 shows an example of TERMAPI call instructions.

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

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

Figure 84. TERMAPI Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

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

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 85 shows an example of WRITE call instructions.

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

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

Figure 85. WRITE Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

Parameter Values Set by the Application

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

S A halfword binary field set to the socket descriptor.

NBYTE
A fullword binary field set to the number of bytes of data to be transmitted.

BUF Specifies the buffer containing the data to be transmitted.

Parameter Values Returned to the Application

ERRNO
A fullword binary field. If RETCODE is negative, the field contains an error number. See “Appendix B. Return Codes” on page 547, for information about ERRNO return codes.

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

Value	**Description**
≥0	A successful call. A return code greater than zero indicates the number of bytes of data written.
–1	Check ERRNO for an error code.

WRITEV

The WRITEV function writes data on a socket from a set of buffers.

The following requirements apply to this call:

Authorization:	Supervisor state or problem state, any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
Amode:	31-bit or 24-bit **Note:** See “Addressability mode (Amode) considerations” under “Environmental Restrictions and Programming Requirements” on page 329.
ASC mode:	Primary address space control (ASC) mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space

Figure 86 on page 404 shows an example of WRITEV call instructions.

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

Figure 86. WRITEV Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

Parameter Values Set by the Application

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

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

Fullword 1
The address of a data buffer.

Fullword 2
Reserved.

Fullword 3
The length of the data buffer referenced in Fullword 1.

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

Parameters Returned by the Application

ERRNO

A fullword binary field. If RETCODE is negative, this contains an error number. See "Appendix B. Return Codes" on page 547, for information about ERRNO return codes.

RETCODE

A fullword binary field.

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

Using Data Translation Programs for Socket Call Interface

In addition to the socket calls, you can use the following utility programs to translate data:

Data Translation

TCP/IP hosts and networks use ASCII data notation; MVS TCP/IP and its subsystems use EBCDIC data notation. In situations where data must be translated from one notation to the other, you can use the following utility programs:

- EZACIC04—Translates EBCDIC data to ASCII data
- EZACIC05—Translates ASCII data to EBCDIC data

Bit String Processing

In C-language, bit strings are often used to convey flags, switch settings, and so on; TCP/IP makes frequent uses of bit strings. However, since bit strings are difficult to decode in COBOL, TCP/IP includes:

- EZACIC06—Translates bit-masks into character arrays and character arrays into bit-masks.
- EZACIC08—Interprets the variable length address list in the HOSTENT structure returned by GETHOSTBYNAME or GETHOSTBYADDR.

EZACIC04

The EZACIC04 program is used to translate EBCDIC data to ASCII data.

Figure 87 shows an example of EZACIC04 call instructions.

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

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

Figure 87. EZACIC04 Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

OUT-BUFFER
A buffer that contains the following:

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

LENGTH
Specifies the length of the data to be translated.

EZACIC05

The EZACIC05 program is used to translate ASCII data to EBCDIC data. EBCDIC data is required by COBOL, PL/1, and assembler language programs.

Figure 88 shows an example of EZACIC05 call instructions.

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

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

Figure 88. EZACIC05 Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

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

LENGTH
Specifies the length of the data to be translated.

EZACIC06

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

Figure 89 shows an example of EZACIC06 call instructions.

```
WORKING STORAGE
    01  CHAR-MASK.
        05 CHAR-STRING              PIC X(nn).

    01  CHAR-ARRAY  REDEFINES CHAR-MASK.
           05  CHAR-ENTRY-TABLE  OCCURS nn TIMES.
               10  CHAR-ENTRY       PIC X(1).
    01  BIT-MASK.
        05 BIT-ARRAY-FWDS           PIC 9(16) COMP.

    01  BIT-FUNCTION-CODES.
        05  CTOB                    PIC X(4) VALUE 'CTOB'.
        05  BTOC                    PIC X(4) VALUE 'BTOC'.

    01  BIT-MASK-LENGTH             PIC 9(8) COMP VALUE 50 .



PROCEDURE CALL (to convert from character to binary)
     CALL 'EZACIC06' USING CTOB
                           BIT-MASK
                           CHAR-MASK
                           BIT-MASK-LENGTH
                           RETCODE.


PROCEDURE CALL (to convert from binary to character)
     CALL 'EZACIC06' USING BTOC
                           BIT-MASK
                           CHAR-MASK
                           BIT-MASK-LENGTH
                           RETCODE.
```

Figure 89. EZACIC06 Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

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 one making socket zero index number one in the character array. You should keep this in mind when turning character positions on and off.

COMMAND

BTOC—Specifies bit string to character array translation.

CTOB—Specifies character array to bit string translation.

BIT-MASK-LENGTH

Specifies the length of the bit-mask.

RETCODE

A binary field that returns one of the following:

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

Example: If you want to use the SELECT call to test sockets zero, five, and nine, and you are using a character array to represent the sockets, you must set the appropriate characters in the character array to one. In this example, index positions one, six and ten in the character array are set to 1. Then you can call EZACIC06 with the COMMAND parameter set to CTOB. When EZACIC06 returns, BIT-MASK contains a fullword with bits zero, five, and nine (numbered from the right) turned on as required by the SELECT call. These instructions process the bit string shown in the following example.

```
MOVE ZEROS TO CHAR-STRING.
MOVE '1'TO CHAR-ENTRY(1), CHAR-ENTRY(6), CHAR-ENTRY(10).
CALL 'EZACIC06' USING TOKEN CTOB BIT-MASK CH-MASK
      BIT-LENGTH RETCODE.
MOVE BIT-MASK TO ....
```

When the select call returns and you want to check the bit-mask string for socket activity, enter the following instructions.

```
MOVE ..... TO BIT-MASK.
CALL 'EZACIC06' USING TOKEN BTOC BIT-MASK CH-MASK
         BIT-LENGTH RETCODE.
PERFORM TEST-SOCKET THRU TEST-SOCKET-EXIT  VARYING IDX
     FROM 1 BY 1 UNTIL IDX EQUAL 10.

TEST-SOCKET.
      IF CHAR-ENTRY(IDX) EQUAL '1'
           THEN PERFORM SOCKET-RESPONSE THRU SOCKET-RESPONSE-EXIT
           ELSE NEXT SENTENCE.
TEST-SOCKET-EXIT.
      EXIT.
```

EZACIC08

The GETHOSTBYNAME and GETHOSTBYADDR calls were derived from C socket calls that return a structure known as HOSTENT. A given TCP/IP host can have multiple alias names and host internet addresses.

TCP/IP uses indirect addressing to connect the variable number of alias names and internet addresses in the HOSTENT structure that is returned by the GETHOSTBYADDR AND GETHOSTBYNAME calls.

If you are coding in PL/1 or assembler language, the HOSTENT structure can be processed in a relatively straight-forward manner. However, if you are coding in COBOL, HOSTENT can be more difficult to process and you should use the EZACIC08 subroutine to process it for you.

It works as follows:

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

Figure 90 on page 411 shows an example of EZACIC08 call instructions.

```
WORKING STORAGE

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

PROCEDURE

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

   Where xxxx is ADDR or NAME.

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

Figure 90. EZAZIC08 Call Instruction Example

For equivalent PL/I and assembler language declarations, see “Converting Parameter Descriptions” on page 331.

Parameter Values set by the Application

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

HOSTALIAS-SEQ
This halfword field is used by EZACIC08 to index the list of alias names. When EZACIC08 is called, it adds one to the current value of HOSTALIAS-SEQ and uses the resulting value to index into the table of alias names. Therefore, for a given instance of GETHOSTBYxxxx, this field should be set to 0 for the initial call to EZACIC08. For all subsequent calls to EZACIC08, this field should contain the HOSTALIAS-SEQ number returned by the previous invocation.

HOSTADDR-SEQ
This halfword field is used by EZACIC08 to index the list of IP addresses. When EZACIC08 is called, it adds one to the current value of HOSTADDR-SEQ and uses the resulting value to index into the table of IP addresses. Therefore, for a given instance of GETHOSTBYxxxx, this field should be set to 0 for the initial call to EZACIC08. For all subsequent calls to EZACIC08, this field should contain the HOSTADDR-SEQ number returned by the previous call.

Parameter Values Returned to the Application

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

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

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

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

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

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

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

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

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

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

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

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

Value	Description
0	Successful completion
-1	Invalid HOSTENT address

Call Interface PL/1 Sample Programs

This section provides sample programs for the call interface that you can use for a PL/1 application program.

The following are the sample programs available in the *hlq*.SEZAINST data set:

Program	**Description**
EZASOKPS	PL/1 call interface sample server program
EZASOKPC	PL/1 call interface sample client program
CBLOCK	PL/1 common variables

Sample Code for Server Program

The EZASOKPS PL/I sample program is a server program that shows you how to use the following calls:

- INITAPI
- SOCKET
- BIND
- GETSOCKNAME
- LISTEN
- ACCEPT
- READ
- WRITE
- CLOSE
- TERMAPI

```
EZASOKPS: PROC OPTIONS(MAIN);
/* INCLUDE CBLOCK - common variables                                  */
% include CBLOCK;
open file(driver);
/**********************************************************************/
/*                                                                    */
/* Execute INITAPI                                                    */
/*                                                                    */
/**********************************************************************/
call ezasoket(INITAPI, MAXSOC, ID, SUBTASK,
                      MAXSNO, ERRNO, RETCODE);
if retcode < 0 then do;
   msg = 'FAIL: initapi' || errno;
   write file(driver) from (msg);
   goto getout;
end;
/**********************************************************************/
/*                                                                    */
/* Execute SOCKET                                                     */
/*                                                                    */
/**********************************************************************/
call ezasoket(SOCKET, AF_INET, TYPE_STREAM, PROTO,
                     ERRNO, RETCODE);
if retcode < 0 then do;
   msg = blank;                        /* clear field                 */
   msg = 'FAIL: socket, stream, internet' || errno;
   write file(driver) from (msg);
   goto getout;
end;
else sock_stream = retcode;
/**********************************************************************/
/*                                                                    */
/* Execute BIND                                                       */
/*                                                                    */
/**********************************************************************/
```

```
name_id.port = 8888;
name_id.address = '01234567'BX;        /* internet address           */
call ezasoket(BIND, SOCK_STREAM, NAME_ID,
                    ERRNO, RETCODE);
if retcode < 0 then do;
   msg = blank;                        /* clear field                */
   msg = 'FAIL: bind' || errno;
   write file(driver) from (msg);
   goto getout;
end;
/**********************************************************************/
/*                                                                    */
/* Execute GETSOCKNAME                                                */
/*                                                                    */
/**********************************************************************/
name_id.port = 8888;
name_id.address = '01234567'BX;        /* internet address           */
call ezasoket(GETSOCKNAME, SOCK_STREAM,
                     NAME_ID, ERRNO, RETCODE);
msg = blank;                           /* clear field                */
if retcode < 0 then do;
   msg = 'FAIL: getsockname, stream, internet' || errno;
   write file(driver) from (msg);
end;
else do;
   msg = 'getsockname = ' || name_id.address;
   write file(driver) from (msg);
end;
/**********************************************************************/
/*                                                                    */
/* Execute LISTEN                                                     */
/*                                                                    */
/**********************************************************************/
backlog = 5;
call ezasoket(LISTEN, SOCK_STREAM, BACKLOG,
                     ERRNO, RETCODE);
if retcode < 0 then do;
   msg = blank;                        /* clear field                */
   msg = 'FAIL: listen w/ backlog = 5' || errno;
   write file(driver) from (msg);
   goto getout;
end;
/**********************************************************************/
/*                                                                    */
/* Execute ACCEPT                                                     */
/*                                                                    */
/**********************************************************************/
name_id.port = 8888;
name_id.address = '01234567'BX;        /* internet address           */
call ezasoket(ACCEPT, SOCK_STREAM,
                     NAME_ID, ERRNO, RETCODE);
msg = blank;                           /* clear field                */
if retcode < 0 then do;
   msg = 'FAIL: accept' || errno;
   write file(driver) from (msg);
end;
else do;
   accpsock = retcode;
   msg = 'accept socket = ' || accpsock;
   write file(driver) from (msg);
end;
/**********************************************************************/
/*                                                                    */
/* Execute READ                                                       */
/*                                                                    */
/**********************************************************************/
nbyte = length(bufin);
```

```
call ezasoket(READ, ACCPSOCK,
                     NBYTE, BUFIN, ERRNO, RETCODE);
msg = blank;                           /* clear field                 */
if retcode < 0 then do;
   msg = 'FAIL: read' || errno;
   write file(driver) from (msg);
end;
else do;
   msg = 'read = ' || bufin;
   write file(driver) from (msg);
   bufout = bufin;
   nbyte = length(bufout);
end;
/**********************************************************************/
/*                                                                    */
/* Execute WRITE                                                      */
/*                                                                    */
/**********************************************************************/
call ezasoket(WRITE, ACCPSOCK, NBYTE, BUFOUT,
                     ERRNO, RETCODE);
msg = blank;                           /* clear field                 */
if retcode < 0 then do;
   msg = 'FAIL: write' || errno;
   write file(driver) from (msg);
end;
else do;
   msg = 'write = ' || bufout;
   write file(driver) from (msg);
end;
/**********************************************************************/
/*                                                                    */
/* Execute CLOSE accept socket                                        */
/*                                                                    */
/**********************************************************************/
call ezasoket(CLOSE, ACCPSOCK,
                     ERRNO, RETCODE);
if retcode < 0 then do;
   msg = blank;                        /* clear field                 */
   msg = 'FAIL: close, accept sock' || errno;
   write file(driver) from (msg);
end;
/**********************************************************************/
/*                                                                    */
/* Execute TERMAPI                                                    */
/*                                                                    */
/**********************************************************************/
getout:
call ezasoket(TERMAPI);
close file(driver);
end ezasokps;
```

Sample Program for Client Program

The EZASOKPC PL/I sample program is a client program that shows you how to use the following calls provided by the call socket interface.

- INITAPI
- SOCKET
- CONNECT
- GETPEERMANE
- WRITE
- READ
- SHUTDOWN
- TERMAPI

```
EZASOKPC: PROC OPTIONS(MAIN);
/* INCLUDE CBLOCK - common variables                                 */
% include CBLOCK;
open file(driver);
/*********************************************************************/
/*                                                                   */
/* Execute INITAPI                                                   */
/*                                                                   */
/*********************************************************************/
call ezasoket(INITAPI, MAXSOC, ID, SUBTASK,
                      MAXSNO, ERRNO, RETCODE);
if retcode < 0 then do;
   msg = 'FAIL: initapi' || errno;
   write file(driver) from (msg);
   goto getout;
end;
/*********************************************************************/
/*                                                                   */
/* Execute SOCKET                                                    */
/*                                                                   */
/*********************************************************************/
call ezasoket(SOCKET, AF_INET, TYPE_STREAM, PROTO,
                     ERRNO, RETCODE);
if retcode < 0 then do;
   msg = blank;                       /* clear field                 */
   msg = 'FAIL: socket, stream, internet' || errno;
   write file(driver) from (msg);
   goto getout;
end;
sock_stream = retcode;                /* save socket descriptor      */
/*********************************************************************/
/* Execute CONNECT                                                   */
/*                                                                   */
/*********************************************************************/
name_id.port = 8888;
name_id.address = '01234567'BX;       /* internet address            */
call ezasoket(CONNECT, SOCK_STREAM, NAME_ID,
                     ERRNO, RETCODE);
if retcode < 0 then do;
   msg = blank;                       /* clear field                 */
   msg = 'FAIL: connect, stream, internet' || errno;
   write file(driver) from (msg);
   goto getout;
end;
/*********************************************************************/
/*                                                                   */
/*   Execute GETPEERNAME                                             */
/*                                                                   */
/*********************************************************************/
call ezasoket(GETPEERNAME, SOCK_STREAM,
                     NAME_ID, ERRNO, RETCODE);
msg = blank;                          /* clear field                 */
if retcode < 0 then do;
   msg = 'FAIL: getpeername' || errno;
   write file(driver) from (msg);
end;
else do;
   msg = 'getpeername =' || name_id.address;
   write file(driver) from (msg);
end;
/*********************************************************************/
/*                                                                   */
/* Execute WRITE                                                     */
/*                                                                   */
/*********************************************************************/
bufout = message;
nbyte = length(message);
```

```
call ezasoket(WRITE, SOCK_STREAM, NBYTE, BUFOUT,
                     ERRNO, RETCODE);
msg = blank;                          /* clear field                  */
if retcode < 0 then do;
   msg = 'FAIL: write' || errno;
   write file(driver) from (msg);
end;
else do;
   msg = 'write = ' || bufout;
   write file(driver) from (msg);
end;
/*********************************************************************/
/*                                                                   */
/* Execute READ                                                      */
/*                                                                   */
/*********************************************************************/
nbyte = length(bufin);
call ezasoket(READ, SOCK_STREAM,
                    NBYTE, BUFIN, ERRNO, RETCODE);
msg = blank;                          /* clear field                  */
if retcode < 0 then do;
   msg = 'FAIL: read' || errno;
   write file(driver) from (msg);
end;
else do;
   msg = 'read = ' || bufin;
   write file(driver) from (msg);
end;
/*********************************************************************/
/*                                                                   */
/* Execute SHUTDOWN from/to                                          */
/*                                                                   */
/*********************************************************************/
getout:
how = 2;
call ezasoket(SHUTDOWN, SOCK_STREAM, HOW,
                     ERRNO, RETCODE);
if retcode < 0 then do;
   msg = blank;                       /* clear field                  */
   msg = 'FAIL: shutdown' || errno;
   write file(driver) from (msg);
end;
/*********************************************************************/
/*                                                                   */
/* Execute TERMAPI                                                   */
/*                                                                   */
/*********************************************************************/
call ezasoket(TERMAPI);
close file(driver);
end ezasokpc;
```

Common Variables Used in PL/1 Sample Programs

The CBLOCK common storage area contains the variables that are used in the PL/1 programs in this section.

```
/*********************************************************************/
/*                                                                   */
/* SOKET COMMON VARIABLES                                            */
/*                                                                   */
/*********************************************************************/
DCL ABS     BUILTIN;
DCL ADDR    BUILTIN;
DCL ACCEPT  CHAR(16) INIT('ACCEPT');
DCL ACCPSOCK FIXED BIN(15);           /* temporary ACCEPT socket     */
DCL AF_INET FIXED BIN(31) INIT(2);    /* internet domain             */
DCL AF_IUCV FIXED BIN(31) INIT(17);   /* iucv domain                 */
DCL ALIAS   CHAR(255);                /* alternate NAME             */
DCL APITYPE FIXED BIN(15) INIT(2);    /* default API type           */
DCL BACKLOG FIXED BIN(31);            /* max length of pending queue*/
DCL BADNAME CHAR(20);                 /* temporary name             */
DCL BIND    CHAR(16) INIT('BIND');
DCL BIT     BUILTIN;
DCL BITZERO BIT(1);                   /* bit zero value             */
DCL BLANK   CHAR(255) INIT(' ');      /*                            */
DCL BUF     CHAR(80) INIT(' ');       /* macro READ/WRITE buffer    */
DCL BUFF  CHAR(15)         INIT(' '); /* short buffer               */
DCL BUFFER   CHAR(32767) INIT(' ');   /* BUFFER                     */
DCL BUFIN    CHAR(32767) INIT(' ');   /* Read buffer                */
DCL BUFOUT   CHAR(32767) INIT(' ');   /* WRITE buffer               */
DCL 1 CLIENT,                     /* socket addr of connection peer */
      2 DOMAIN FIXED BIN(31) INIT(2), /* domain of client (AF_INET) */
      2 NAME   CHAR(8) INIT(' '),     /* addr identifier for client */
      2 TASK   CHAR(8) INIT(' '),     /* task identifier for client */
      2 RESERVED CHAR(20) INIT(' ');  /* reserved                   */
DCL CLOSE   CHAR(16) INIT('CLOSE');
DCL COMMAND FIXED BIN(31) INIT(3);    /* Query FNDELAY flag         */
DCL CONNECT CHAR(16) INIT('CONNECT');
DCL COUNT FIXED BIN(31) INIT(100);    /* elements in GRP_IOCTL_TABLE*/
DCL DATA_SOCK FIXED BIN(15);          /* temporary datagram socket  */
DCL DEF     FIXED BIN(31) INIT(0);    /* default protocol           */
DCL DRIVER  FILE OUTPUT UNBUF ENV(FB RECSIZE(100)) RECORD;
DCL ERETMSK CHAR(4);                 /* indicate exception events  */
DCL ERR     FIXED BIN(31);            /* error number variable      */
DCL ERRNO   FIXED BIN(31) INIT(0);    /*  error number              */
DCL ESNDMSK CHAR(4);                  /* check for pending          */
                                      /*   exception events         */
DCL EXIT    LABEL;                    /* common exit point          */
DCL EZACIC05 ENTRY OPTIONS(RETCODE,ASM,INTER) EXT;
DCL EZASOKET ENTRY OPTIONS(RETCODE,ASM,INTER) EXT; /* socket call   */
DCL FCNTL  CHAR(16) INIT('FCNTL');
DCL FIONBIO FIXED BIN(31) INIT(-2147178626);/* flag: nonblocking    */
DCL FIONREAD FIXED BIN(31) INIT(+1074046847);/* flag:#readable bytes*/
DCL FLAGS   FIXED BIN(31) INIT(0);    /* default: no flags          */
                                      /*   1 = OOB, SEND OUT-OF-BAND*/
                                      /*   4 = DON'T ROUTE          */
DCL GETCLIENTID  CHAR(16) INIT('GETCLIENTID');
DCL GETHOSTBYADDR  CHAR(16) INIT('GETHOSTBYADDR');
DCL GETHOSTBYNAME CHAR(16) INIT('GETHOSTBYNAME');
DCL GETHOSTNAME CHAR(16) INIT('GETHOSTNAME');
DCL GETHOSTID CHAR(16) INIT('GETHOSTID');
DCL GETIBMOPT CHAR(16) INIT('GETIBMOPT');
DCL GETPEERNAME CHAR(16) INIT('GETPEERNAME');
DCL GETSOCKNAME CHAR(16) INIT('GETSOCKNAME');
DCL GETSOCKOPT CHAR(16) INIT('GETSOCKOPT');
DCL GIVESOCKET  CHAR(16) INIT('GIVESOCKET');
DCL GLOBAL CHAR(16) INIT('GLOBAL');
DCL HOSTADDR FIXED BIN(31);           /* host internet address      */
DCL HOW     FIXED BIN(31) INIT(2);    /* how shutdown is to be done */
DCL I       FIXED BIN(15);            /* loop index                 */
DCL ICMP    FIXED BIN(31) INIT(2);    /* prototype icmp  ???        */
DCL 1 ID,                             /*                            */
      2 TCPNAME CHAR(8) INIT('TCPIP'), /* remote address space      */
      2 ADSNAME CHAR(8) INIT('USER19J'); /* local address space     */
DCL IDENT   POINTER;                  /* TCP/IP Addr Space          */
DCL IFCONF  CHAR(255);                /* configuration structure    */
DCL IFREQ   CHAR(255);                /* interface structure        */
DCL INDEX   BUILTIN;
DCL IOCTL  CHAR(16) INIT('IOCTL');
DCL IOCTL_CMD FIXED BIN(31);          /* ioctl command              */
DCL IOCTL_REQARG  POINTER ;           /* send   pointer to data area*/
DCL IOCTL_RETARG  POINTER ;           /* return pointer to data area*/
DCL IOCTL_REQ00   FIXED BIN(31);      /* command request argument   */
DCL IOCTL_REQ04   FIXED BIN(31);      /* command request argument   */
DCL IOCTL_REQ08   FIXED BIN(31);      /* command request argument   */
DCL IOCTL_REQ32   CHAR(32) INIT(' '); /* command request argument   */
DCL IOCTL_RET00   FIXED BIN(31);      /* command return  argument   */
DCL IOCTL_RET04   FIXED BIN(31);      /* command return  argument   */
DCL INITAPI CHAR(16) INIT('INITAPI'); /*                            */
DCL 1 INTERNET,                       /* internet address           */
      2 NETID1 FIXED BIN(31) INIT(9), /* network id, part 1         */
      2 NETID2 FIXED BIN(31) INIT(67), /* network id, part 2        */
      2 SUBNETID FIXED BIN(31) INIT(30), /* subnet id               */
      2 HOSTID FIXED BIN(31) INIT(137); /* host id                  */
DCL IP      FIXED BIN(31) INIT(1);    /* prototype ip   ???         */
DCL J       FIXED BIN(15);            /* loop index                 */
DCL K       FIXED BIN(15);            /* loop index                 */
DCL LENGTH  BUILTIN;
DCL LABL    CHAR(9);
DCL LISTEN  CHAR(16) INIT('LISTEN');
DCL MAXSNO  FIXED BIN(31) INIT(0);    /* max descriptor assigned    */
DCL MAXSOC  FIXED BIN(15) INIT(255);  /* largest socket # checked   */
DCL MESSAGE CHAR(50) INIT('I love my 1 @ Rottweiler!'); /* message  */
DCL MSG     CHAR(100) INIT(' ');      /*   message text             */
DCL 1 NAME_ID,                    /* socket addr of connection peer */
      2 FAMILY FIXED BIN(15) INIT(2), /*addr'g family; TCP/IP def   */
      2 PORT   FIXED BIN(15),         /* system assigned port #     */
      2 ADDRESS FIXED BIN(31),        /* 32-bit internet            */
      2 RESERVED CHAR(8);             /* reserved                   */
DCL NAMEL   CHAR(255)    VARYING;     /* name field, long           */
DCL NAMES   CHAR(24);                 /* name field, short          */
DCL NAMELEN FIXED BIN(31);            /* length of name/alias field */
DCL NBYTE   FIXED BIN(31);            /* Number of bytes in buffer  */
DCL NOTE(3) CHAR(25) INIT('Now is the time for 198 g',
                          'ood people to come to the',
                          ' aid of their parties!');
DCL NS      FIXED BIN(15);            /* socket descriptor, new     */
DCL NULL    BUILTIN;
DCL OPTL    FIXED BIN(31);            /* length of OPTVAL string    */
DCL OPTLEN  FIXED BIN(31);            /* length of OPTVAL string    */
DCL OPTN    CHAR(15);                 /* OPTNAME value (macro)      */
DCL OPTNAME FIXED BIN(31);            /* OPTNAME value (call)       */
DCL OPTVAL  CHAR(255);                /* GETSOCKOPT option data     */
DCL OPTVALD FIXED BIN(31);            /* SETSOCKOPT option data     */
DCL 1 OPT_STRUC,                      /* structure for option       */
      2 ON_OFF FIXED BIN(31) INIT(1), /* enable option              */
      2 TIME   FIXED BIN(31) INIT(5); /* time-out in seconds        */
DCL 1 OPT_STRUCT,                     /* structure for option       */
      2 ON     FIXED BIN(31),         /* used for getsockopt        */
      2 TIMEOUT FIXED BIN(31);        /* time-out in seconds        */
DCL PLITEST BUILTIN;                  /* debug tool                 */
DCL PROTO   FIXED BIN(31) INIT(0);    /* prototype default          */
DCL READ    CHAR(16) INIT('READ');
DCL READV   CHAR(16) INIT('READV');
DCL RECV    CHAR(16) INIT('RECV');
DCL RECVFROM CHAR(16) INIT('RECVFROM');
DCL RECVMSG CHAR(16) INIT('RECVMSG');
DCL REUSE   FIXED BIN(31) INIT('4');  /* toggle, reuse local addr   */
DCL REQARG  FIXED BIN(31);            /* command request argument   */
DCL RETC    FIXED BIN(31);            /* return code variable       */
DCL RETARG  CHAR(255);                /* return argument data area  */
DCL RETCODE FIXED BIN(31) INIT(0);    /* return code                */
DCL RETLEN  FIXED BIN(31);            /* return area data length    */
DCL RRETMSK CHAR(4);                 /* indicate READ EVENTS       */
DCL RSNDMSK CHAR(4):               /* check for pending read events */
DCL RTENTRY CHAR(50) INIT('dummy table'); /* router entry           */
DCL SAVEFAM  FIXED BIN(15);           /* temporary family name      */
DCL SELECB CHAR(4) INIT('1');
DCL SELECT CHAR(16) INIT('SELECT');
DCL SELECTEX CHAR(16) INIT('SELECTEX');
DCL SEND CHAR(16) INIT('SEND');
DCL SENDMSG CHAR(16) INIT('SENDMSG');
DCL SENDTO CHAR(16) INIT('SENDTO');
DCL SETSOCKOPT CHAR(16) INIT('SETSOCKOPT');
DCL SHUTDOWN CHAR(16) INIT('SHUTDOWN');
DCL SIOCADDRT FIXED BIN(31) INIT(-2144295158);
                                          /* flag: add routing entry*/
DCL SIOCATMARK FIXED BIN(31) INIT(+1074046727);
                                           /* flag: out-of-band data*/
DCL SIOCDELRT FIXED BIN(31) INIT(-2144295157);
                                          /* flag: delete routing   */
DCL SIOCGIFADDR FIXED BIN(31) INIT(-1071601907);
                                            /*flag: network int addr*/
DCL SIOCGIFBRDADDR FIXED BIN(31) INIT(-1071601902);
                                               /*flag  net broadcast*/
DCL SIOCGIFCONF FIXED BIN(31) INIT(-1073174764);
                                            /* flag: netw int config*/
DCL SIOCGIFDSTADDR FIXED BIN(31) INIT(-1071601905);
                                               /* flag: net des addr*/
DCL SIOCGIFFLAGS FIXED BIN(31) INIT(-1071601903);
                                             /* flag: net intf flags*/
DCL SIOCGIFMETRIC FIXED BIN(31) INIT(-1071601897);
                                              /* flag: get rout metr*/
DCL SIOCGIFNETMASK FIXED BIN(31) INIT(-1071601899);
                                               /* flag: network mask*/
DCL SIOCGIFNONSENSE FIXED BIN(31) INIT(-1234567890);
                                                /* flag: nonsense   */
DCL SIOCSIFMETRIC FIXED BIN(31) INIT(-2145343720);
                                              /* flag: set rout metr*/
DCL SOCK     FIXED BIN(15);           /* socket descriptor          */
DCL SOCKET  CHAR(16) INIT('SOCKET');
DCL SOCK_DATAGRAM FIXED BIN(15);      /* socket descriptor datagram */
DCL SOCK_RAW FIXED BIN(15);           /* socket descriptor raw      */
DCL SOCK_STREAM FIXED BIN(15);        /* stream socket descriptor   */
DCL SOCK_STREAM_1 FIXED BIN(15);      /* stream socket descriptor   */
DCL SO_BROADCAST FIXED BIN(31) INIT(32); /* toggle, broadcast msg   */
DCL SO_ERROR FIXED BIN(31) INIT(4103); /* check/clear async error   */
DCL SO_KEEPALIVE FIXED BIN(31) INIT(8); /* request status of stream*/
DCL SO_LINGER  FIXED BIN(31) INIT(128); /* toggle, linger on close  */
DCL SO_OOBINLINE FIXED BIN(31) INIT(256);/*toggle, out-of-bound data*/
DCL SO_REUSEADDR FIXED
               BIN(31) INIT(4);       /* toggle, local address reuse*/
DCL SO_SNDBUF  FIXED BIN(31) INIT(4097);
DCL SO_TYPE FIXED BIN(31) INIT(4104); /* return type of socket      */
DCL STRING  BUILTIN;
DCL SUBSTR  BUILTIN;
DCL SUBTASK CHAR(8) INIT('ANYNAME');  /* task/path identifier       */
DCL SYNC    CHAR(16) INIT('SYNC');
DCL TAKESOCKET CHAR(16) INIT('TAKESOCKET');
DCL TASK    CHAR(16) INIT('TASK');
DCL TERMAPI CHAR(16) INIT('TERMAPI'); /*                            */
DCL TIME    BUILTIN;
DCL 1 TIMEOUT,
      2  TIME_SEC  FIXED BIN(31),     /* value in secs              */
      2  TIME_MSEC FIXED BIN(31);     /* value in millisecs         */
DCL TYPE_DATAGRAM FIXED BIN(31) INIT(2);/*fixed lengthconnectionless*/
DCL TYPE_RAW  FIXED BIN(31) INIT(3); /* internal protocol interface */
DCL TYPE_STREAM FIXED BIN(31) INIT(1); /* two-way byte stream       */
DCL WRETMSK CHAR(4);                 /* indicate WRITE EVENTS      */
DCL WRITE   CHAR(16) INIT('WRITE');
DCL WRITEV   CHAR(16) INIT('WRITEV');
DCL WSNDMSK CHAR(4);               /*check for pending write events */
```

Chapter 12. REXX Socket Application Programming Interface (API)

This chapter describes the application program interface (API) for socket call instructions written in REXX, for TCP/IP CS for OS/390 environment.

The REXX socket program uses the REXX built-in function RXSOCKET to access the TCP/IP socket interface. The program maps the socket calls from the C programming language to the REXX programming language. This allows you to use REXX to implement and test TCP/IP applications. Examples of the corresponding C socket call are included where they apply.

This chapter describes the following:

- REXX socket initialization
- REXX socket programming hints and tips
- Coding socket built-in function
- Error messages and return codes
- Coding calls to process socket sets
- Coding calls to initialize, change, and close sockets
- Coding calls to exchange data for sockets
- Coding calls to resolve names for REXX sockets
- Coding calls to manage configuration, options, and modes
- REXX socket sample programs

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

REXX Socket Initialization

The member RXSOCKET in the TCP/IP load library is called when the REXX built-in function socket is called. RXSOCKET must be part of the step library when your REXX program calls the socket built-in function.

REXX Socket Programming Hints and Tips

This section contains information that you might find useful if you plan to use the REXX socket interface to TCP/IP.

- To use the socket calls contained in the socket function, one socket set must be active. The Initialize call creates a socket set, and can support multiple calls. The *subtaskid* for a socket set identifies the socket set and normally corresponds to the application name.
- The *name* parameter contains *domain*, *portid*, and *ipaddress*. If specified as an input parameter for socket calls, you can specify *ipaddress* as a name that is resolved by the name server. For example, you can enter 'IBMVM1' or 'IBMVM1.IBM.COM'. When returned as a result, *ipaddress* is always specified in the dotted decimal format. For example, '128.228.1.2' can be returned.
- A socket can be in blocking or nonblocking mode. In blocking mode, calls such as send and recv, block the caller until the operation completes successfully or an error occurs. In nonblocking mode, the caller is not blocked, but the operation ends immediately with the return code 35 (EWOULDBLOCK) or 36 (EINPROGRESS). You can use the fcntl or Ioctl calls to switch between blocking and nonblocking mode.

- When a socket is in nonblocking mode, you can use the select call to wait for socket events. Data arriving at a socket for a read or recv call is a possible event. If the socket is not ready to send data because buffer space for the transmitted message is not available at the receiving socket, your REXX program can wait until the socket is ready for sending data.
- If your application uses the givesock and takesock calls to transfer a socket from a master program to a slave program, both the master and slave programs need to agree on a mechanism for exchanging client IDs and the socket ID to be transferred. The slave program must also signal the master program when the takesocket call is successfully completed. The master program can then close the socket.
- The socket options SO_ASCII and SO_EBCDIC identify the socket's data type for use by the REXX/RXSOCKET program. Setting SO_EBCDIC on has no effect, and setting SO_ASCII on causes all incoming data on the socket to be translated from ASCII to EBCDIC and all outgoing data on the socket to be translated from EBCDIC to ASCII. REXX/RXSOCKET uses the following hierarchy of translation tables:

  ```
  user_prefix.subtaskid.TCPXLBIN
  user_prefix.userid.TCPXLBIN
  user_prefix.STANDARD.TCPXLBIN
  user_prefix.RXSOCKET.TCPXLBIN
  Internal tables
  ```

  The first four tables are data sets and they are searched in the order in which they are listed. If no files are found, the internal tables corresponding to the ISO standard are used. You can change the `user_prefix` parameter to match your site convention.

Coding the Socket Built-in Function

The socket built-in function describes an interface that allows you to use the socket calls associated with the C language in a REXX application. Calls are included to:

- Process socket sets
- Initialize, change, and close sockets
- Exchange data for sockets
- Resolve names for sockets
- Manage configurations, options, and modes for sockets

The first parameter in the Socket built-in function identifies the function of the call. The corresponding C socket call is included where applicable.

Two REXX sample programs for a client/server application are described in “REXX Socket Sample Programs” on page 475.

The socket calls are ordered by function. For example, the initialize and terminate calls are described before the calls that process sockets.

Socket

This call provides access to the TCP/IP socket interface. The subfunction specifies the socket call. Each call contains the name socket, *subfunction*, and optional parameters (*arg*). For a description of each call, see:

- “Coding Calls to Process Socket Sets” on page 424
- “Coding Calls to Initialize, Change, and Close Sockets” on page 430
- “Coding Calls to Exchange Data for Sockets” on page 440
- “Coding Calls to Resolve Names for REXX Sockets” on page 449
- “Coding Calls to Manage Configuration, Options, and Modes” on page 463

```
►►──Socket──(──subfunction,──┬──────────────┬──)──────────────────────────►◄
                             │  ┌─,─────┐   │
                             │  ▼       │   │
                             └───arg────┴───┘
```

Parameters

subfunction
The name of the socket call.

arg
The parameters for the socket call.

Return Values

A character string containing a return code is returned.

The returned string contains several values separated by a blank to allow you to parse it using REXX. The first value is a return code. If the return code is zero, the values following the return code are returned by the subfunction called. If the return code is not zero, the second value indicates an error code and the rest of the string is the corresponding error message.

```
Socket('GetHostId')       ==   '0 9.4.3.2'
Socket('Recv',socket)     ==   '35 EWOULDBLOCK Operation would block'
```

Error Messages and Return Codes

For information about error messages, refer to *TCP/IP for MVS: Messages and Codes*.

The REXX built-in function socket returns a return code in the first token slot in the result string. For more information about return codes, see “Appendix B. Return Codes” on page 547.

Coding Calls to Process Socket Sets

A socket set is a number of preallocated sockets available to a single application. You can define multiple socket sets for one session, but only one socket set can be active at a time.

Initialize

This call preallocates the number of sockets specified and returns the *subtaskid* for the socket set that is active when this Initialize call is issued. It also returns the maximum number of preallocated sockets and the name of the TCP/IP service provider. If this call is successful, the socket set identified by *subtaskid* automatically becomes the active socket set.

```
►►──Socket──(──'Initialize'──,──subtaskid──,──┬─────────┬──,──┬─────────┬──)──────►◄
                                              └─maxdesc─┘     └─service─┘
```

Parameters

subtaskid
A name for a socket set. The name can as many as eight printable characters and cannot contain blanks.

maxdesc
The number of preallocated sockets in a socket set. The number can be in the range 1—2000. The default is 40.

service
The name of the TCP/IP service.

Return Values

A string containing a return code, *subtaskid*, *maxdesc*, and *service* is returned.

Example

```
Socket('Initialize','myId')   ==   '0 myId 40 TCPIP'
```

Socketsetlist

This call returns a list of the subtask IDs for all available socket sets in the current order of the stack.

```
►►──Socket──(──'Socketsetlist'──)──────────────────────────────────────────────►◄
```

Return Values

A string containing a return code and the active *subtaskid* is returned. If this call changes the active *subtaskid*, the previous *subtaskid* is also returned.

Example

```
Socket('SocketSetList')       ==   '0 myId firstId'
```

Socketset

This call returns the *subtaskid* of the active socket set, and optionally makes the specified socket set the active socket set.

```
►►──Socket──(──'Socketset'──,──┬───────────┬──)─────────────────────────────►◄
                               └─subtaskid─┘
```

Parameters

subtaskid
The name for a socket set. The name can be can be as many as eight printable nonblank characters.

Return Values

A string containing a return code and a subtask ID is returned.

Example

```
Socket('SocketSet','firstId')  ==  '0 myId'
```

Socketsetstatus

This call returns the status of the socket set. If the socket set is connected, the call returns the number of free and the number of allocated sockets in the socket set. If the set is severed, the reason for the TCP/IP sever is also returned. If the *subtaskid* parameter is not specified, the active socket set is used. Initialized socket sets should be in connected status, and uninitialized socket sets should be in free status.

A socket set that is initialized and is not in connected status must be terminated before the *subtaskid* can be reused.

```
►►──Socket──(──'Socketsetstatus'──,──┬─────────────┬──)──────────────────────────►◄
                                     └─subtaskid───┘
```

Parameters

subtaskid
The name for a socket set. The name can be can be as many as eight printable nonblank characters.

Return Values

The string containing a return code, *subtaskid* and status is returned. The call optionally returns connect and sever information.

Example

```
Socket('SocketSetStatus')  ==  '0 myId Connected Free 17 Used 23'
```

Terminate

This call closes all sockets in the socket set, releases the socket set, and returns the *subtaskid* for the socket set. If the *subtaskid* is not specified, the active socket set is ended. If the active socket set is ended, the next socket set in the stack becomes the active socket set.

```
►►──Socket──(──'Terminate'──,──┬─────────────┬──)──────────────────────────────►◄
                               └─subtaskid───┘
```

Parameters

subtaskid
A name for a socket set. The name can be as many as eight printable characters and cannot contain blanks.

Return Values

A string containing a return code and a *subtaskid* is returned.

Example

```
Socket('Terminate','myId')  ==  '0 myId'
```

Coding Calls to Initialize, Change, and Close Sockets

A socket is an endpoint for communication that can be named and addressed in a network. A socket is represented by a socket identifier (*socketid*). A socket ID used in a Socket call must be in the active socket set.

Accept

This call is used by a server to accept a connection request from a client. It accepts the first connection on the queue of pending connections. It creates a new *socketid* with the same properties as the given *socketid*. If the queue has no pending connection requests, accept blocks the caller unless the socket is in nonblocking mode. If no connection request is pending and the socket is in nonblocking mode, accept ends with the return code 35 (EWOULDBLOCK). The original socket remains available to accept more connection requests.

```
►►──Socket──(──'Accept'──,──socketid──)────────────────────────────────────►◄
```

Parameters

socketid
The socket descriptor.

Return Values

A string containing a return code, new socket ID, and name is returned.

Example

```
Socket('Accept',5)  ==   '0 6 AF_INET 5678 9.4.3.2'

C socket call: accept(s, name, namelen)
```

Bind

This call binds a unique local name to the socket with the given *socketid*. After calling socket, a socket does not have a name associated with it, but it does belong to an addressing family. The form of the name depends on the addressing family. The bind call also allows servers to specify the network interfaces from which they wish to receive UDP packets and TCP connection requests.

```
►►──Socket──(──'Bind'──,──socketid──,──name──)─────────────────────────────────►◄
```

Parameters

socketid
The socket descriptor that is to be bound.

name
The network address consists of:

domain
Must be set to 2 for AF_INET.

portid Set to the port number to which the socket must bind.

ipaddress
Set to the IP address to which the socket must bind.

Return Values

A string containing a return code is returned.

Example

```
Socket('Bind',5,'AF_INET 1234 128.228.1.2')   ==   '0'

C socket call: bind(s, name, namelen)
```

Close

This call shuts down the socket associated with the *socketid*, and frees resources allocated to the socket. If the *socketid* refers to an open TCP connection, the connection is closed. If a stream socket is closed when there is input data queued, the TCP connection is reset rather than being cleanly closed.

```
►►──Socket──(──'Close'──,──socketid──)──────────────────────────────────►◄
```

Parameters

socketid
The socket descriptor.

Return Values

A string containing a return code is returned.

Example

```
Socket('Close',6)   ==   '0'

C socket call: close(s)
```

Connect

For stream sockets, the connect call completes the bind for a socket, if bind has not been called, and tries to establish a connection to another socket. For datagram sockets, this call specifies the peer for a socket. If the socket is in blocking mode, this function blocks the caller until the connection is established, or until an error is received. If the socket is in nonblocking mode, this function ends with the return code 36 (EINPROGRESS), or another return code indicating an error.

```
►►—Socket—(—'Connect'—,—socketid—,—name—)———————————————————————————►◄
```

Parameters

socketid
The socket descriptor.

name
The network address consists of:

domain
Must be set to 2 for AF_INET.

portid Set to the port number to which the socket must bind.

ipaddress
Set to the IP address to which the socket must bind.

Return Values

A string containing a return code is returned.

Example

```
Socket('Connect',5,'AF_INET 1234 128.228.1.2')       ==   '0'
Socket('Connect',5,'AF_INET 1234 CUNYVM')            ==   '0'
Socket('Connect',5,'AF_INET 1234 CUNYVM.CUNY.EDU')   ==   '0'

C socket call: connect(s, name, namelen)
```

Givesocket

This call transfers the socket with the given socket descriptor to another application. Givesocket makes the specified socket available to a takesocket call issued by another application running on the same host. Any connected stream socket can be given. Normally, givesocket is used by a master program that obtains sockets using the accept call and gives them to slave programs that handle one socket at a time.

```
►►──Socket──(──GIVESOCKET──,──socketid──,──clientid──)─────────────────────────►◄
```

Parameters

socketid
The socket descriptor.

clientid
The identifier for another application. The format is *domain*, *userid*, and *subtaskid*. The domain field must be set to 2 for AF_INET.

Return Values

A string containing a return code is returned.

Example

```
Socket('GiveSocket',6,'AF_INET USERID2 hisId')   ==   '0'

C socket call: givesocket(s, clientid)
```

Listen

This call applies only to stream sockets and performs two tasks. If Bind has not been called, it completes the bind for a socket. It also creates a connection request queue, whose length is specified as *backlog*, for incoming connection requests. If the queue is full, the connection request is ignored.

```
►►──Socket──(──'Listen'──,──socketid──,──┬─────────────┬──)──────────────────────►◄
                                         └─backlog─────┘
```

Parameters

socketid
The socket descriptor.

backlog
The number of pending connect requests. The number is an integer between 0 and the maximum number that can be specified with the SOMAX CONN parameter in TCPIP.PROFILE. The default is 10. The minimum size to the connection request queue (number of pending connect requests) is effectively two. Therefore, if BACKLOG is specified as 0, 1, or 2, then the number of pending connect requests is 2.

Return Values

A string containing a return code is returned.

Example

```
Socket('Listen',5,10)   ==   '0'

C socket call: listen(s, backlog)
```

Shutdown

This call shuts down all or part of a duplex connection. The *how* parameter sets the condition for shutting down the connection to the socket.

```
►►──Socket──(──'Shutdown'──,──socketid──,──┬───────┬──)──────────────────────────►◄
                                           └─how───┘
```

If you issue SHUTDOWN for a socket that currently has outstanding socket calls pending, see Table 3 on page 35 to determine the effects of this operation on the outstanding socket calls.

Parameters

socketid
The socket descriptor.

how
The choices are as follows:

- ’SEND’, ’SENDING’, ’TO’, ’WRITE’, ’WRITING’—Ends further send operations.
- ’FROM’, ’READ’, ’READING’, ’RECEIVE’, ’RECEIVING’—Ends further receive operations.
- ’BOTH’ (default)—Ends further send and receive operations.

Return Values

A string containing a return code is returned.

Example

```
Socket('ShutDown',6,'BOTH')   ==   '0'

C socket call: shutdown(s, how)
```

Socket

This call creates a socket in the active socket set that is an endpoint for communication, and returns a socket identification.

```
►►──Socket──(──'Socket'──,──┬────────┬──,──┬──────┬──,──┬──────────┬──)──────────►◄
                            └─domain─┘     └─type─┘     └─protocol─┘
```

Parameters

domain
The addressing family must be set to 2 for AF_INET.

type
One of the following socket types. SOCK_STREAM or STREAM is the default.
- 'SOCK_STREAM'
- 'STREAM'
- 'SOCK_DGRAM'
- 'DATAGRAM'
- 'SOCK_RAW'
- 'RAW'

Note: If you select the type SOCK_RAW or RAW, the application must be an APF authorized application.

protocol
One of the following protocol names. The *protocol* field **should be set to 0** to allow TCP/IP to assign the default protocol for the domain and socket type selected. The protocol defaults are TCP for stream sockets and UDP for datagram sockets. There is no default for raw sockets.
- '0'
- 'IPPROTO_UDP'
- 'UDP'
- 'IPPROTO_TCP'
- 'TCP'

When the socket type is raw, the following values are valid for the *protocol* field.
- 'IPPROTO_ICMP'
- 'IPPROTO_RAW'
- 'RAW'

Return Values

A string containing a return code and a new socket ID is returned.

Example

```
Socket('Socket')  ==   '0 5'

C socket call: socket(domain, type, protocol)
```

Takesocket

This call acquires a socket from another program that obtains the other program's *clientid* and *socketid* by a method not defined by TCP/IP. After a successful call to takesocket, the other application must close the socket.

```
►►──Socket──(──'Takesocket'──,──clientid──,──socketid──)─────────────────────────►◄
```

Parameters

clientid
The identifier for another application. The identifier contains a *domain*, *userid*, and *subtaskid*. The domain field must be set to 2 for AF_INET.

socketid
The socket descriptor belonging to another *clientid*.

Return Values

A string containing a return code and a new socket ID is returned.

Example

```
Socket('TakeSocket','AF_INET USERID1 myId',6)   ==   '0 7'

C socket call: takesocket(clientid, hisdesc)
```

Coding Calls to Exchange Data for Sockets

On a connected stream socket and on datagram sockets, you can send and receive data. You can use the calls in this section to send, receive, read, and write data.

Read

This call reads up to *maxlength* bytes of data. This is the conventional TCP/IP read data operation. If less than the number of bytes requested is available, the call returns the number currently available. If data is not available at the socket, the call waits for data to arrive and blocks the caller, unless the socket is in nonblocking mode. If the length is zero, the other side of the call closed the stream socket.

For datagram sockets, Read returns the entire datagram that was sent, providing that the datagram fits into the specified buffer.

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

```
►►──Socket──(──'Read'──,──socketid──,──┬───────────────┬──)──────────────────────►◄
                                       └──maxlength──┘
```

Parameters

socketid
The socket descriptor.

maxlength
The maximum data length. The length is a number in the range 1—100 000. The default is 10 000.

Return Values

A string containing a return code, the data length, and the data read is returned.

Example

```
Socket('Read',6)             ==   '0 21 This is the data line'

C socket call: read(s, buf, len)
```

Recv

This call receives up to *maxlength* bytes of data from the incoming message. RECV applies only to connected sockets. For additional control of the incoming data, you can use RECV to:

- Peek at the incoming data without having it removed from the buffer.
- Read out-of-band data.

If more than the number of bytes requested is available on a datagram socket, the call discards excess bytes. If less than the number of bytes requested is available, the call returns the number of bytes currently available. If data is not available at the socket, the call waits for data to arrive and blocks the caller, unless the socket is in nonblocking mode. If the length is zero, the other side closed the stream socket.

```
►►──Socket──(──'Recv'──,──socketid──,──┬─────────────┬──,──┬─────────────┬──)────────►◄
                                       └─maxlength───┘     └─recvflags───┘
```

Parameters

socketid
The socket descriptor.

maxlength
The maximum data length. The length is a number in the range 1—100 000. The default is 10 000.

recvflags
Specifies the following receive flags:

'MSG_OOB', 'OOB', 'OUT_OF_BAND'
Read out-of-band data on the socket. Only stream sockets created in the AF_INET domain support out-of-band data.

'MSG_PEEK', 'PEEK'
Look at the data on the socket but do not change or destroy it. The next Recv call can read the same data.

'' (default)
Receive the data. No flag is set.

Return Values

A string containing a return code, the data length, and the data received is returned.

Example

```
Socket('Recv',6)              ==   '0 21 This is the data line'
Socket('Recv',6,,'PEEK OOB')  ==   '0 24 This is out-of-band data'

C socket call: recv(s, buf, len, flags)
```

Recvfrom

This call receives the incoming message, up to *maxlength* bytes of data. If more than the number of bytes requested is available on a datagram socket, the call discards excess bytes. If less than the number of bytes requested is available, the call returns the number of bytes available. If data is not available at the socket, Recvfrom waits for data to arrive and blocks the caller, unless the socket is in nonblocking mode.

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

For UDP sockets, RECVFROM returns the entire datagram that was sent if it will fit in the buffer. If the datagram packet is too large to fit in the buffer, the excess bytes are discarded.

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

If data is not available for the socket, and the socket is in blocking mode, RECVFROM blocks the caller until data arrives. If data is not available and the socket is in nonblocking mode, RECVFROM returns a −1 and sets ERRNO to 35 (EWOULDBLOCK).

```
►►──Socket──(──'Recvfrom'──,──socketid──,──┬───────────┬──,──┬───────────┬──────►
                                           └─maxlength─┘     └─recvflags─┘

►──)──────────────────────────────────────────────────────────────────────────►◄
```

Parameters

socketid
The socket descriptor.

maxlength
The maximum data length. The length is a number between 1 and 100 000. The default is 10 000.

recvflags
Specifies the following receive flags:

'MSG_OOB', 'OOB', 'OUT_OF_BAND'
Read out-of-band data on the socket. Only stream sockets created in the AF_INET domain support out-of-band data.

'MSG_PEEK', 'PEEK'
Look at the data on the socket but do not change or destroy it.

'' (default)
Receive the data. No flag is set.

Return Values

A string containing a return code, a name, a length, and the data is returned.

Example

```
Socket('RecvFrom',6)          ==   '0 AF_INET 5678 9.4.3.2 9 Data line'

C socket call: recvfrom(s, buf, len, flags, name, namelen)
```

Send

This call sends the outgoing data message to a connected socket. If Send cannot send the number of bytes of data that is requested, it waits until sending is possible. This blocks the caller, unless the socket is in nonblocking mode. For datagram sockets, the socket should not be in blocking mode.

```
►►──Socket──(──'Send'──,──socketid──,──data──,──┬───────────┬──)──────────────►◄
                                                └─sendflags─┘
```

Parameters

socketid
The socket descriptor.

data
The message string for data exchange.

sendflags
Specifies the following send flags:

'MSG_OOB', 'OOB', 'OUT_OF_BAND'
Sends out-of-band data on sockets that support it. Only stream sockets created in the AF_INET domain support out-of-band data.

'MSG_DONTROUTE', 'DONTROUTE'
Do not route the data. Routing is handled by the calling program.

'' (default)
Send the data. No flag is set.

Return Values

A string containing a return code and the length of the data sent is returned.

Example

```
Socket('Send',6,'Some text')                ==   '0 9'
Socket('Send',6,'Out-of-band data','OOB')   ==   '0 16'

C socket call: send(s, buf, len, flags)
```

Sendto

Sendto is similar to send, except that it includes the destination address parameter. You can use the destination address if you want to use the Sendto call to send datagrams on a UDP socket, regardless of whether or not the socket is connected. For datagram sockets, the socket should not be in blocking mode.

Use the FLAGS parameter to:

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

For stream sockets, data is processed as streams of information with no boundaries separating the data. For example, if a program is required to send 1000 bytes, each call to this function can send any number of bytes, up to the entire 1000 bytes, with the number of bytes sent returned in the return values. Therefore, programs using stream sockets should place Sendto in a loop that repeats the call until all data has been sent.

```
►►──Socket──(──'Sendto'──,──socketid──,──data──,──┬───────────┬──,──┬──────┬──────►
                                                  └─sendflags─┘     └─name─┘

►──)──────────────────────────────────────────────────────────────────────────────►◄
```

Parameters

socketid
The socket descriptor.

data
The message string for data exchange.

sendflags
Specifies the following send flags:

'MSG_DONTROUTE', 'DONTROUTE'
Do not route the data. Routing is handled by the calling program.

'' (default)
Send the data. No flag is set.

'MSG_OOB', 'OOB', 'OUT_OF_BAND'
Sends out-of-band data on sockets that support it. Only stream sockets created in the AF_INET domain support out-of-band data.

name
The network address in the format:

domain
Must be set to 2 for AF_INET.

portid Set to the port number to which the socket must bind

ipaddress
Set to the IP address to which the socket must bind

Return Values

A string containing a return code and a length is returned.

Example

```
Socket('SendTo',6,'Some text',,'AF_INET 5678 9.4.3.2'              == '0 9'
Socket('SendTo',6,'Some text',,'AF_INET 5678 ZURLVM1'              == '0 9'
Socket('SendTo',6,'Some text',,'AF_INET 5678 ZURLVM1.ZURICH.IBM.COM' == '0 9'

C socket call: sendto(s, buf, len, flags, name, namelen)
```

Write

This call writes the given number of bytes of data for a connected socket. This call is similar to SEND, except that it lacks the control flags available with SEND. If it is not possible to write the number of bytes requested,

Write waits until conditions are suitable for writing data. This blocks the caller, unless the socket is in nonblocking mode. For datagram sockets, the socket should not be in blocking mode.

```
►►—Socket—(—'Write'—,—socketid—,—data—)——————————————————————————►◄
```

Parameters

socketid
The socket descriptor.

data
The message string for data exchange.

Return Values

A string containing a return code and the length of the data written is returned.

Example

```
Socket('Write',6,'Some text')   ==   '0 9'

C socket call: write(s, buf, len)
```

Coding Calls to Resolve Names for REXX Sockets

You can use the name resolution calls to get information such as name, address, client identification, and host name. You can also use the name resolution calls to resolve an internet protocol address (*ipaddress*) to a symbolic name or a symbolic name to an *ipaddress*.

Getclientid

This call returns the calling program's TCP/IP virtual machine identifier.

```
►►──Socket──(──'Getclientid'──,──┬────────┬──)─────────────────────────────────►◄
                                 └─domain─┘
```

Parameters

domain
The socket domain. The name must be AF_INET.

Return Values

A string containing a return code and a client identification is returned.

Example

```
Socket('GetClientId')         ==   '0 AF_INET USERID1 myId'

C socket call: getclientid(domain, clientid)
```

Getdomainname

This call returns the domain name for the processor running the program.

```
►►──Socket──(──'Getdomainname'──)──────────────────────────────────────────►◄
```

Return Values

A string containing a return code and a domain name is returned.

Example

```
Socket('GetDomainName')       ==   '0 ZURICH.IBM.COM'

C socket call: getdomainname(name, namelen)
```

Gethostbyaddr

This call resolves the host name through a name server, if one is present.

```
►►──Socket──(──'Gethostbyaddr'──,──ipaddress──)──────────────────────────────────►◄
```

Parameters

ipaddress

The *ipaddress* in dotted-decimal notation. You can also use the following keywords:

'INADDR_ANY', 'ANY'

If you do not specify an address the system uses any available address.

'INADDR_BROADCAST', 'BROADCAST'

Sets the address to the broadcast address.

'' (default)

Send the data. No flag is set.

Note: If an *ipaddress* is not present, the Gethostbyaddr call gets the address from *hlq*.HOSTS.ADDRINFO. For more information, see *OS/390 IBM Communications Server: IP Configuration Reference*.

Return Values

A string containing a return code and a full host name is returned.

Example

```
Socket('GetHostByAddr',128.228.1.2')   ==   '0 CUNYVM.CUNY.EDU'

C socket call: gethostbyaddr(addr, addrlen, domain)
```

Gethostbyname

This call tries to resolve the host name through a name server, if one is present. Gethostbyname returns all *ipaddresses* for multihome hosts. The addresses are separated by blanks.

```
►►──Socket──(──'Gethostbyname'──,──┬─hostname─────┬──)─────────────────────────►◄
                                   └─fullhostname─┘
```

Parameters

hostname
The host processor name as a character string.

fullhostname
A fully qualified host name in the form hostname.domainname.

Return Values

A string containing a return code and an *ipaddress* list is returned.

Example

```
Socket('GetHostByName','CUNYVM')           ==   '0 128.228.1.2'
Socket('GetHostByName','CUNYVM.CUNY.EDU')   ==   '0 128.228.1.2'

C socket call: gethostbyname(name)
```

Gethostid

This call returns the *ipaddress* for the current host. This address is the default home *ipaddress*.

```
►►──Socket──(──'Gethostid'──)──────────────────────────────────────────────────►◄
```

Return Values

A string containing a return code and the *ipaddress* is returned.

Example

```
Socket('GetHostId')          ==   '0 9.4.3.2'

C socket call: gethostid()
```

Gethostname

This call returns the name of the host processor for the program.

```
►►──Socket──(──'Gethostname'──)──────────────────────────────────────────►◄
```

Return Values

A string containing a return code and the host name is returned.

Example

```
Socket('GetHostName')          ==   '0 ZURLVM1'

C socket call: gethostname(name, namelen)
```

Getpeername

This call returns the name of the peer that is connected to the given socket.

```
►►──Socket──(──'Getpeername'──,──socketid──)─────────────────────────────────────►◄
```

Parameters

socketid
The socket descriptor.

Return Values

A string containing a return code and a peer name is returned.

Example

```
Socket('GetPeerName',6)       ==   '0 AF_INET 1234 128.228.1.2'

C socket call: getpeername(s, name, namelen)
```

Getprotobyname

This call returns the number of a network protocol when you specify a protocol name.

```
►►──Socket──(──'Getprotobyname'──,──protocolname──)──────────────────────────────►◄
```

Parameters

protocolname
The name of a network protocol. The names TCP, UDP, and ICMP are valid.

Return Values

A string containing a return code and a protocol number is returned. If the protocol name specified on the call is incorrect, a zero return code and a zero for the protocol number is returned.

Example

```
Socket('GetProtoByName','TCP')             ==   '0 6'

C socket call: getprotobyname(name)
```

Getprotobynumber

This call returns the name of a network protocol when you specify a protocol number.

```
►►──Socket──(──'Getprotobynumber'──,──protocolnumber──)──────────────────────────►◄
```

Parameters

protocolnumber
A network protocol number. The number is a positive integer.

Return Values

A string containing a return code and a protocol name is returned. If the protocol number specified on the call is incorrect, a zero return code and a null protocol number string is returned.

Example

```
Socket('GetProtoByNumber',6)                 ==   '0 TCP'

C socket call: getprotobynumber(name)
```

Getservbyname

This call returns the service name, the port, and the network protocol name.

```
►►──Socket──(──'Getservbyname'──,──servicename──,──┬────────────────┬──)──────────►◄
                                                   └─protocolname───┘
```

Parameters

servicename
A service name such as FTP.

protocolname
A network protocol name, such as TCP, UDP, or ICMP.

Return Values

A string containing a return code, service name, port ID, and protocol name is returned. If the service name specified on the call is incorrect, a zero return code and a null service name string is returned.

Example

```
Socket('GetServByName','ftp','tcp')       ==   '0 FTP 21 TCP'

C socket call: getservbyname(name, proto)
```

Getservbyport

This call returns the name of a service, port, and network protocol. If the port ID specified on the call is incorrect, a zero return code and a null port ID string is returned. If the port ID and the protocol name are correct, but the protocol name does not exist for the port ID specified, the call returns the proper protocol service name and port ID.

If both port ID and protocolname are correct, but the protocolname does not exist for the specified port ID, the call returns the proper protocolname, port ID, and service name.

```
►►──Socket──(──'Getservbyport'──,──portid──,──┬──────────────┬──)──────────────►◄
                                              └─protocolname─┘
```

Parameters

portid
A port number. The number must be an integer between zero and 65 535. You can also specify ANY or INPORT_ANY to get a port ID from TCP/IP.

'INPORT_ANY', 'ANY'
If you do not specify a port number, TCP/IP returns any available port ID.

protocolname
A network protocol name, such as TCP, UDP, or ICMP.

Return Values

A string containing a return code, service name, port ID, and protocol name is returned.

Example

```
Socket('GetServByPort',21,'tcp')          ==   '0 FTP 21 TCP'

C socket call: getservbyport(name, proto)
```

Getsockname

This call returns the name to which the given socket was bound. Stream sockets are not assigned a name, until after a successful call to bind, connect, or accept.

```
►►──Socket──(──'Getsockname'──,──socketid──)──────────────────────────────────────►◄
```

Parameters

socketid
The socket descriptor.

Return Values

A string containing a return code and a socket name is returned.

Example

```
Socket('GetSockName',7)       ==   '0 AF_INET 5678 9.4.3.2'

C socket call: getsockname(s, name, namelen)
```

Resolve

This call resolves the host name through a name server, if a name server is present.

```
►►──Socket──(──'Resolve'──,──┬─ipaddress────┬──)──────────────────────────────────►◄
                             ├─hostname─────┤
                             └─fullhostname─┘
```

Parameters

ipaddress
The *ipaddress* in dotted decimal notation.

Note: If an *ipaddress* is not present, the resolve call gets the address from *hlq*.HOSTS.ADDRINFO. For more information, see *OS/390 IBM Communications Server: IP Configuration Reference*.

hostname
The name of a host processor.

fullhostname
A fully qualified host name in the form hostname.domainname.

Return Values

A string containing a return code, an *ipaddress*, and a full host name is returned.

Example

```
Socket('Resolve','128.228.1.2')       ==   '0 128.228.1.2 CUNYVM.CUNY.EDU'
Socket('Resolve','CUNYVM')            ==   '0 128.228.1.2 CUNYVM.CUNY.EDU'
Socket('Resolve','CUNYVM.CUNY.EDU')   ==   '0 128.228.1.2 CUNYVM.CUNY.EDU'
```

Coding Calls to Manage Configuration, Options, and Modes

Use this group of calls to obtain the version number of the REXX/SOCKETS function package, get socket options, set socket options, or socket mode of operation. There are also socket calls to determine the network configuration.

Fcntl

This call allows you to control the operating characteristics of a socket. You can use Fcntl to set the blocking or nonblocking mode for a socket.

```
►►──Socket──(──'Fcntl'──,──socketid──,──fcmd──,──┬──────────┬──)──────────────────►◄
                                                 └──fvalue──┘
```

Parameters

socketid
The socket descriptor for this socket.

fcmd
The command. The options are F_SETFL or F_GETFL.

'F_SETFL'
Sets the status flags for the socket. One flag, FNDELAY, can be set.

'F_GETFL'
Gets the status for the socket. One flag, FNDELAY, can be retrieved.

The FNDELAY flag marks the socket as being in nonblocking mode. If data is not present on calls that can block, such as read, readv, and recv, fcntl returns error code 35 (EWOULDBLOCK).

fvalue
The following operating characteristic values:

- 'BLOCKING '
- '0'
- 'NON-BLOCKING'
- 'FNDELAY'

Return Values

A string containing a return code and *fvalue* for F_GETFL is returned.

Example

```
Socket('Fcntl',5,'F_SETFL','NON-BLOCKING')   ==   '0'
Socket('Fcntl',5,'F_GETFL')                  ==   '0 NON-BLOCKING'

C socket call: fcntl(s, cmd, data)
```

Getsockopt

The Getsockopt call gets the active options for a socket that were set with the Setsockopt call.

Multiple options can be associated with each socket. These options are described below. You must specify the option that you want when you issue the Getsockopt call.

```
►►──Socket──(──'Getsockopt'──,──socketid──,──level──,──optname──)──────────────►◄
```

Parameters

socketid
The socket descriptor for the socket requiring options.

level
The protocol level for which the socket is being set. The levels are SOL_SOCKET and IPPROTO_TCP. All commands beginning with SO_ are for protocol level SOL_SOCKET, and are interpreted by the general socket code. All commands beginning with TCP_ are for protocol level IPPROTO_TCP, and are interpreted by the TCP/IP internal code.

Note: Only SOL_SOCKET is supported for MVS sockets.

optname
Specifies one or more of the following commands:

'SO_ASCII'
Returns the status of the translation to ASCII data option. When SO_ASCII is set, data is translated to ASCII. When SO_ASCII is not set, data is not translated to or from ASCII. This option is ignored by ASCII hosts. The *optvalue* parameter, ON or OFF, is returned and is optionally followed by the name of the translation table used, if translation is applied to the data.

'SO_BROADCAST'
Requests the status of the broadcast option, which is the ability to send broadcast messages over the socket. The default is *disabled*. This option does not apply to stream sockets. The *optvalue* parameter can be ON or OFF.

'SO_DEBUG'
Returns the status of the debug option. The default is *disabled*. The debug option controls the recording of debug information. The *optvalue* parameter can be ON or OFF.

'SO_EBCDIC'
Returns the status of the translation to EBCDIC data option. When SO_EBCDIC is set, data is translated to EBCDIC. When SO_EBCDIC is not set, data is not translated to or from EBCDIC. The *optvalue* parameter, ON or OFF, is returned and is optionally followed by the name of the translation table used, if translation is applied to the data.

'SO_ERROR'
Requests pending errors on the socket and clears the error status. Use SO_ERROR to check for asynchronous errors on connected datagram sockets or for other asynchronous errors that are not explicitly returned by one of the socket calls. The *optvalue* parameter can be ON or OFF.

'SO_KEEPALIVE'

SO_KEEPALIVE returns the status of the keepalive option. The default is *disabled*. Keepalive periodically sends a datagram on an idle connection. If the remote TCP/IP does not respond to this datagram or to retransmissions of this datagram the connection ends with the error 60 (ETIMEDOUT). The *optvalue* parameter can be ON or OFF.

'SO_LINGER'

Requests the status of SO_LINGER.

- When the SO_LINGER option is enabled and data transmission has not been completed, The default is *disabled*. a Close call blocks the calling program until the data is transmitted or until the connection has timed out.
- If SO_LINGER is not enabled, a CLOSE call returns without blocking the caller and TCP/IP still tries to send the data. Normally the data transfer is successful, but it cannot be guaranteed, because TCP/IP repeats the Send request for only a specified period of time.

The *optvalue* parameter can be ON, OFF, or a number. If ON is selected, the default number is 120.

'SO_OOBINLINE'

Requests the status of how out-of-band data is to be received. This option applies only to stream sockets in the AF_INET domain.

- When SO_OOBINLINE is enabled, out-of-band data is placed in the normal data input queue as it is received. The default is *disabled*. Recv and Recvfrom can then receive the data without enabling the MSG_OOB flag.
- When SO_OOBINLINE is disabled out-of-band data is placed in the priority data input queue as it is received. Recv and Recvfrom must now enable the MSG_OOB flag to receive the data.

The *optvalue* parameter can be ON or OFF.

'SO_SNDBUF'

Returns the size of the TCP/IP send buffer in OPTVAL. The *optvalue* parameter can be ON or OFF.

'SO_REUSEADDR'

Toggles local address reuse. The default is disabled. This alters the normal algorithm used in the bind() call.

The normal bind() call algorithm allows each internet address and port combination to be bound only once. If the address and port have been bound already, a subsequent bind() will fail and result error EADDRINUSE.

After the 'SO_REUSEADDR' option is active, the following situations are supported:

- A server can bind() the same port multiple times as long as every invocation uses a different local IP address, and the wildcard address INADDR_ANY is used only one time per port.
- A server with active client connections can be restarted and can bind to its port, without having to close all of the client connections.
- For datagram sockets, multicasting is supported so multiple bind() calls can be made to the same class D address and port number.

'SO_TYPE'

Returns the socket type, SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW in the *optvalue* field.

Return Values

A string containing a return code and an option value is returned.

Example

```
Socket('GetSockOpt',5,'Sol_Socket','So_ASCII')       ==   '0 On STANDARD'
Socket('GetSockOpt',5,'Sol_Socket','So_Broadcast')   ==   '0 On'
Socket('GetSockOpt',5,'Sol_Socket','So_Error')       ==   '0 0'
Socket('GetSockOpt',5,'Sol_Socket','So_Linger')      ==   '0 On 60'
Socket('GetSockOpt',5,'Sol_Socket','So_Sndbuf')      ==   '0 8192'
Socket('GetSockOpt',5,'Sol_Socket','So_Type')        ==   '0 SOCK_STREAM'

C socket call: getsockopt(s, level, optname, optval, optlen)
```

Ioctl

This call allows you to control the operating characteristics of a socket. You can use this call to set blocking or nonblocking mode and to control the operations characteristics determined by the *icmd* command.

```
►►—Socket—(—'Ioctl'—,—socketid—,—icmd—,—┬────────┬—)——————————————►◄
                                        └─ivalue─┘
```

Parameters

socketid
The socket descriptor for this socket.

icmd
The following operating characteristics commands:

'FIONBIO'
Use FIONBIO to set or clear nonblocking for socket I/O. Set or clear is determined by the value in *ivalue*. The *ivalue* parameter can be ON or OFF.

'FIONREAD'
Use FIONREAD to get the number of immediately readable bytes of data for the socket and return it in *ivalue*.

'SIOCATMARK'
Use SIOCATMARK to determine if the current location in the input data is pointing to out-of-band data. The command returns YES or NO in the *ivalue* field.

'SIOCGIFADDR'
Use SIOCGIFADDR to get the network interface address. The *ivalue* parameter returns the address in the format; interface, domain, port, and IP address.

'SIOCGIFBRDADDR'
Use SIOCGIFBRDADDR to get the network interface broadcast address. The *ivalue* parameter returns the address in the format; interface, domain, port, and internet address.

'SIOCGIFCONF'
Use SIOCGIFCONF to get the network interface configuration. The *ivalue* parameter contains the maximum number of interfaces that is returned. The call returns a list of interfaces in the format; interface, domain, port, and internet address.

'SIOCGIFDSTADDR'
Use SIOCGIFDSTADDR to get the network interface destination address. The *ivalue* parameter returns the address in the format; interface, domain, port, and internet address.

'SIOCGIFFLAGS'
Use SIOCGIFFLAGS to get the network interface flags. The *ivalue* parameter returns the flags in four hexadecimal digits. The symbolic names of the enabled flags are also returned.

'SIOCGIFMETRIC'
Use SIOCGIFMETRIC to get the network interface routing metric. The *ivalue* parameter returns the interface together with the metric integer.

'SIOCGIFNETMASK'

Use SIOCGIFNETMASK to get the network interface network mask. The *ivalue* parameter returns the mask in the format; interface, domain, port, and internet address. The call also sets or clears nonblocking I/O for a socket depending on the value specified for the *ivalue* input parameter. The *ivalue* parameter can be ON or OFF.

ivalue

The operating characteristics value. The operating characteristics value depends on the value specified for *icmd*. The *ivalue* parameter can be used as input or output or both on the same call.

Return Values

A string containing a return code and *ivalue* information is returned.

Example

```
Socket('Ioctl',5,'FionBio','On')         ==  '0'
Socket('Ioctl',5,'FionRead')             ==  '0 8192'
Socket('Ioctl',5,'SiocAtMark')           ==  '0 No'
Socket('Ioctl',5,'SiocGifConf',2)        ==  '0 TR1 AF_INET 0 9.4.3.2 TR2 AF_INET 0 9.4.3.3'
Socket('Ioctl',5,'SiocGifAddr','TR1')    ==  '0 TR1 AF_INET 0 9.4.3.2'
Socket('Ioctl',5,'SiocGifFlags','TR1')   ==
    '0 TR1 0063 IFF_UP IFF_BROADCAST IFF_NOTRAILERS IFF_RUNNING'
Socket('Ioctl',5,'SiocGifMetric','TR1')  ==  '0 TR1 0'
Socket('Ioctl',5,'SiocGifNetMask','TR1') ==  '0 TR1 AF_INET 0 255.255.255.0'

C socket call: ioctl(s, cmd, data)
```

Select

Use this call to wait for socket-related events. Select returns all active socket IDs that have completed events when it is called. It does not check for order of completion.

A close on the other side of a socket connection is not reported as an exception, but as a read event that returns zero bytes of data. When connect is called with a socket in nonblocking mode, the connect call ends and returns the code, 36 (EINPROGRESS). The connection setup completion is then reported as a write event on the socket. When accept is called with a socket in nonblocking mode, the accept call ends and returns the code, 35 (EWOULDBLOCK). The availability of the connection request is reported as a Read event on the original socket, and accept should be called only after the read has been reported.

```
►►──Socket──(──'Select'──,'Read'──socketidlist──'Write'──socketidlist──────────────►

►─'Exception'──socketidlist──,──────────────────)──────────────────────────────────►◄
                                └─timeout.──┘
```

Parameters

socketidlist
A list of socket descriptors.

timeout
A positive integer indicating the maximum wait time in seconds. The default is FOREVER.

Return Values

A string containing a return code, count, read socket ID list, write socket ID list, and exception socket ID list is returned.

Example

```
Socket('Select','Read 5 Write Exception',10)   ==   '0 1 READ 5 WRITE EXCEPTION'

C socket call: select(nfds, readfds, writefds, exceptfds, timeout)
```

Setsockopt

Setsockopt sets the options associated with a socket. Setsockopt can be called only for sockets in the AF_INET domain. This call is not supported in the AF_IUCV domain.

The *optvalue* parameter is used to pass data used by the particular set command. The *optvalue* parameter points to a buffer containing the data needed by the set command. The *optvalue* parameter is optional and can be set to 0, if data is not needed by the command.

```
►►──Socket──(──'Setsockopt'──,──socketid──,──level──,──optname──,──optvalue──)────►◄
```

Parameters

socketid
The socket descriptor for the socket setting options.

level
The protocol level for which the socket is being set. The levels are SOL_SOCKET and IPPROTO_TCP. All commands beginning with SO_ are for protocol level SOL_SOCKET, and are interpreted by the general socket code. All commands beginning with TCP_ are for protocol level IPPROTO_TCP, and are interpreted by the TCP/IP internal code.

Note: Only SOL_SOCKET is supported for MVS sockets.

optname
Specifies one or more of the following commands:

'SO_ASCII'
Sets the translation to ASCII data option. When SO_ASCII is set, data is translated to ASCII. When SO_ASCII is not set, data is not translated to or from ASCII. The *optvalue* parameter can be ON or OFF and is optionally followed by the name of the translation table that is used, if translation is applied to the data.

'SO_BROADCAST'
Sets the broadcast option, which is the ability to send broadcast messages over the socket. The default is *disabled*. This option does not apply to stream sockets. The *optvalue* parameter can be ON or OFF.

'SO_DEBUG'
Use SO_DEBUG to set the debug option. The default is *disabled*. The debug option controls the recording of debug information. The *optvalue* parameter can be ON or OFF.

'SO_EBCDIC'
Sets the translation to EBCDIC data option. When SO_EBCDIC is set, data is translated to EBCDIC. When SO_EBCDIC is not set, data is not translated to or from EBCDIC. This option is ignored by EBCDIC hosts. The *optvalue* parameter ON or OFF is returned and is optionally followed by the name of the translation table used, if translation is applied to the data.

'SO_KEEPALIVE'
SO_KEEPALIVE sets the status of the keepalive option. The default is *disabled*. Keepalive periodically sends a datagram on an idle connection. If the remote TCP/IP does not respond to this datagram or

to retransmissions of this datagram the connection is ends with the error 60 (ETIMEDOUT). The *optvalue* parameter can be ON or OFF.

'SO_LINGER'

Sets the SO_LINGER option to delay closing a socket.

- When the SO_LINGER option is 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. The default is *disabled*.
- If SO_LINGER is not enabled, a CLOSE call returns without blocking the caller and TCP/IP still tries to send the data. Normally this transfer is successful, but it cannot be guaranteed, because TCP/IP repeats the Send request for only a specified period of time.

This option applies only to stream sockets.

The *optvalue* parameter can be ON, OFF, or a number. If ON is selected, the default number is 120.

'SO_OOBINLINE'

Sets the how to receive out-of-band data option. This option applies only to stream sockets in the AF_INET domain.

- When SO_OOBINLINE is enabled, out-of-band data is placed in the normal data input queue as it is received. The default is *disabled*. Recv and recvfrom can then receive the data without enabling the MSG_OOB flag.
- When SO_OOBINLINE is disabled out-of-band data is placed in the priority data input queue as it is received. Recv and recvfrom must now enable the MSG_OOB flag to receive the data.

The *optvalue* parameter can be ON or OFF.

'SO_REUSEADDR'

Toggles local address reuse. The default is disabled. This alters the normal algorithm used in the bind() call.

The normal bind() call algorithm allows each internet address and port combination to be bound only once. If the address and port have been bound already, a subsequent bind() will fail and result error EADDRINUSE.

After the 'SO_REUSEADDR' option is active, the following situations are supported:

- A server can bind() the same port multiple times as long as every invocation uses a different local IP address, and the wildcard address INADDR_ANY is used only one time per port.
- A server with active client connections can be restarted and can bind to its port, without having to close all of the client connections.
- For datagram sockets, multicasting is supported so multiple bind() calls can be made to the same class D address and port number.

'SO_SNDBUF'

Gets the size of the send buffer in bytes. The *optvalue* parameter is not allowed.

optvalue

Returns the option value for the requested option.

- For all values of *optname* other than SO_LINGER, *optvalue* is a 32-bit field, containing the status of the requested option.
  - If the requested option is enabled, this field contains a positive value. If the requested option is disabled, this field contains a zero value.
  - If *optname* is set to SO_ERROR, *optvalue* contains the most recent error for the socket. This error variable is then cleared.
  - If *optname* is set to SO_TYPE, *optvalue* returns X'1' for SOCK_STREAM, X'2' for SOCK_DGRAM, or X'3' for SOCK_RAW.
- If SO_LINGER is requested in *optname*, the LINGER 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 the option is disabled.
  - The SO_LINGER value indicates the amount of time in seconds that TCP/IP continues trying to send the data after the CLOSE call is issued.

Return Values

A string containing a return code is returned.

Example

```
Socket('SetSockOpt',5,'Sol_Socket','So_ASCII','On')       ==   '0'
Socket('SetSockOpt',5,'Sol_Socket','So_Broadcast','On')   ==   '0'
Socket('SetSockOpt',5,'Sol_Socket','So_Linger',60)        ==   '0'

C socket call: setsockopt(s, level, optname, optval, optlen)
```

Version

This call returns the name REXX/SOCKETS, version number, and version date for the REXX Socket service.

```
►►──Socket──(──'Version'──)────────────────────────────────────────────────►◄
```

Return Values

A string containing a return code and REXX/SOCKETS version data is returned.

Example

```
Socket('Version')            ==   '0 REXX/SOCKETS CS V2R6 APR 17 1998'
```

REXX Socket Sample Programs

This section provides sample REXX socket programs.

The following are the sample REXX socket programs available in the *hlq*.SEZAINST data set:

Program	Description
REXX-EXEC RSCLIENT	Client sample program
REXX-EXEC RSSERVER	Server sample program

Before you start the client program, you must start the server program in another address space. The two programs can run on different hosts, but the internet address of the host running the server program must be entered with the command starting the client program, and the hosts must be connected on the same network using TCP/IP.

The REXX-EXEC RSCLIENT Sample Program

The client sample program is a REXX socket program that shows you how to use the calls provided by REXX/SOCKETS. The program connects to the server sample program and receives data, which is displayed on the screen. It uses sockets in blocking mode.

After parsing and testing the input parameters, the RSCLIENT EXEC obtains a socket set using the initialize call and a socket using the socket call. The program then connects to the server and writes the user ID, the node ID, and the number of lines requested on the connection to the server. It reads data in a loop and displays it on the screen until the data length is zero, indicating that the server has closed the connection. If an error occurs, the client program displays the return code, determines the status of the socket set, and ends the socket set.

The server adds the EBCDIC new line character to the end of each record, and the client uses this character to determine the start of a new record. If the connection is abnormally closed, the client does not display partially received records.

```
trace o
signal on halt
signal on syntax

/* Set error code values                                            */
initialized = 0

parse arg argstring
argstring = strip(argstring)
if substr(argstring,1,1) = '?' then do
  say 'RSSERVER and RSCLIENT  are a pair of programs which provide an'
  say 'example of how to use REXX/SOCKETS to implement a service. The'
  say 'server must be started before the clients get started.        '
  say '                                                              '
  say 'The RSSERVER program runs on a dedicated TSO USERID.          '
  say 'It returns a number of data lines as requested to the client. '
  say 'It is started with the command:   RSSERVER                    '
  say 'and terminated with the ATTN KEY and the immediate command HI.'
  say 'Alternate methods of running the RSSERVER are TSO/E Background'
  say '(IKJEFT01) or MVS Batch (IRXJCL).                             '
  say '                                                              '
  say 'The RSCLIENT program is used  to request a number of arbitrary'
  say 'data lines  from the server.  One or more clients can access  '
  say 'the server until it is terminated.                            '
  say 'It is started with the command:  RSCLIENT number      '
  say 'where "number" is the number of data lines to be requested and'
  say '"server" is the ipaddress of the service virtual machine. (The'
  say 'default ipaddress is the one of the host  on which RSCLIENT is'
  say 'running, assuming that RSSERVER runs on the same host.)       '
  say '                                                              '
  exit 100
end

/* Split arguments into parameters and options                        */
parse upper var argstring parameters '(' options ')' rest

/* Parse the parameters                                               */
parse var parameters lines server rest
if datatype(lines,'W') then call error 'E', 24, 'Invalid number'
lines = lines + 0
if rest='' then call error 'E', 24, 'Invalid parameters'

/* Parse the options                                                  */
do forever
  parse var options token options
  select
    when token='' then leave
    otherwise call error 'E', 20, 'Invalid option "'token'"'
  end
end

/* Initialize control information                                     */
port = '1952'               /* The port used by the server            */

/* Initialize                                                         */
call Socket 'Initialize', 'RSCLIENT'
if src=0 then initialized = 1
else call error 'E', 200, 'Unable to initialize SOCKET'
if server='' then do
  server = Socket('GetHostId')
  if src^=0 then call error 'E', 200, 'Cannot get the local ipaddress'
end
ipaddress = server

/* Initialize for receiving lines sent by the server                  */
s = Socket('Socket')
if src^=0 then call error 'E', 32, 'SOCKET(SOCKET) rc='src
hostname = translate(Socket('GetHostName'))
if src^=0 then call error 'E', 32, 'SOCKET(GETHOSTNAME) rc='src
call Socket 'Connect', s, 'AF_INET' port ipaddress
if src^=0 then call error 'E', 32, 'SOCKET(CONNECT) rc='src
call Socket 'Write', s, hostname userid() lines
if src^=0 then call error 'E', 32, 'SOCKET(WRITE) rc='src

/* Wait for lines sent by the server                                  */
dataline = ''
num = 0
do forever

  /* Receive a line and display it                                    */
  parse value Socket('Read', s) with len newline
  if src^=0 ] len<=0'' then leave
  dataline = dataline ]] newline
  do forever
    if pos('15'x,dataline)=0 then leave
    parse var dataline nextline '15'x dataline
    num = num + 1
    say right(num,5)':' nextline
  end
end

/* Terminate and exit                                                 */
```

```
call Socket 'Terminate'
exit 0

/* Calling the real SOCKET function                                    */
socket: procedure expose src
  a0 = arg(1)
  a1 = arg(2)
  a2 = arg(3)
  a3 = arg(4)
  a4 = arg(5)
  a5 = arg(6)
  a6 = arg(7)
  a7 = arg(8)
  a8 = arg(9)
  a9 = arg(10)
  parse value 'SOCKET'(a0,a1,a2,a3,a4,a5,a6,a7,a8,a9) with src res
return res

/* Syntax error routine                                                */
syntax:
  call error 'E', rc, '==> REXX Error No.' 20000+rc
return


/* Halt processing routine                                             */
halt:
  call error 'E', 4, '==> REXX Interrupted'
return

/* Error message and exit routine                                      */
error:
  type = arg(1)
  retc = arg(2)
  text = arg(3)
  ecretc = right(retc,3,'0')
  ectype = translate(type)
  ecfull = 'RXSCLI' ]] ecretc ]] ectype
  say '===> Error:' ecfull text
  if typeˆ='E' then return
  if initialized
     then do
       parse value Socket('SocketSetStatus') with . status severreason
       if statusˆ='Connected'
          then say 'The status of the socket set is' status severreason
     end
  call Socket 'Terminate'
exit retc
```

The REXX-EXEC RSSERVER Sample Program

The server sample program shows an example of how to use sockets in nonblocking mode. The program waits for connect requests from client programs, accepts the requests, and then sends data. The sample can handle multiple client requests in parallel processing.

The server program sets up a socket to accept connection requests from clients and waits in a loop for events reported by the select call. If a socket event occurs, it is processed. A read event can occur on the original socket for accepting connection requests and on sockets for accepted socket requests. A write event can only occur on sockets for accepted socket requests.

A read event on the original socket for connection requests means that a connection request from a client occurred. Read events on other sockets indicate that there is either data to receive or that the client has closed the socket. Write events indicate that the server can send more data. The server program sends only one line of data in response to a write event.

The server program keeps a list of sockets to which it wants to write. It keeps this list to avoid unwanted socket events. The TCP/IP protocol is not designed for one single-threaded program communicating on many different sockets, but for multithread applications where one thread processes only events from a single socket.

```
trace o
signal on syntax
signal on halt

/* Set error code values                                              */
initialized = 0

parse arg argstring
argstring = strip(argstring)
if substr(argstring,1,1) = '?' then do
  say 'RSSERVER and RSCLIENT  are a pair of programs which provide an'
  say 'example of how to use REXX/SOCKETS to implement a service. The'
  say 'server must be started before the clients get started.       '
  say '                                                              '
  say 'The RSSERVER program runs on a dedicated TSO USERID.          '
  say 'It returns a number of data lines as requested to the client. '
  say 'It is started with the command:  RSSERVER                     '
  say 'and terminated with the ATTN key and the immediate command HI.'
  say 'Alternate methods of running the RSSERVER are TSO/E Background'
  say '(IKJEFT01) or MVS Batch (IRXJCL).                             '
  say '                                                              '
  say 'The RSCLIENT program is used  to request a number of arbitrary'
  say 'data lines  from the server.  One or more clients can access  '
  say 'the server until it is terminated.                            '
  say 'It is started with the command:  RSCLIENT number       '
  say 'where "number" is the number of data lines to be requested and'
  say '"server" is the ipaddress of the service virtual machine. (The'
  say 'default ipaddress is the one of the host  on which RSCLIENT is'
  say 'running, assuming that RSSERVER runs on the same host.)       '
  say '                                                              '
  exit 100
end

/* Split arguments into parameters and options                        */
parse upper var argstring parameters '(' options ')' rest

/* Parse the parameters                                               */
parse var parameters rest
if restˆ='' then call error 'E', 24, 'Invalid parameters specified'

/* Parse the options                                                  */
do forever
  parse var options token options
  select
    when token='' then leave
    otherwise call error 'E', 20, 'Invalid option "'token'"'
  end
end

/* Initialize control information                                     */
port = '1952'                /* The port used for the service         */

/* Initialize                                                         */
say 'RSSERVER: Initializing'
call Socket 'Initialize', 'RSSERVER'
if src=0 then initialized = 1
else call error 'E', 200, 'Unable to initialize SOCKET'
```

```
ipaddress = Socket('GetHostId')
if srcˆ=0 then call error 'E', 200, 'Unable to get the local ipaddress'
say 'RSSERVER: Initialized: ipaddress='ipaddress 'port='port

/* Initialize for accepting connection requests                       */
s = Socket('Socket')
if srcˆ=0 then call error 'E', 32, 'SOCKET(SOCKET) rc='src
call Socket 'Bind', s, 'AF_INET' port ipaddress
if srcˆ=0 then call error 'E', 32, 'SOCKET(BIND) rc='src
call Socket 'Listen', s, 10
if srcˆ=0 then call error 'E', 32, 'SOCKET(LISTEN) rc='src
call Socket 'Ioctl', s, 'FIONBIO', 'ON'
if srcˆ=0 then call error 'E', 36, 'Cannot set mode of socket' s

/* Wait for new connections and send lines                            */
timeout    = 60
linecount. =  0
wlist = ''
do forever

  /* Wait for an event                                                */
  if wlistˆ='' then sockevtlist = 'Write'wlist 'Read * Exception'
  else sockevtlist = 'Write Read * Exception'
  sellist = Socket('Select',sockevtlist,timeout)
  if srcˆ=0 then call error 'E', 36, 'SOCKET(SELECT) rc='src
  parse upper var sellist . 'READ' orlist 'WRITE' owlist 'EXCEPTION' .
  if orlistˆ='' ] owlistˆ='' then do
     event = 'SOCKET'
     if orlistˆ='' then do
        parse var orlist orsocket .
        rest = 'READ' orsocket
     end
     else do
       parse var owlist owsocket .
       rest = 'WRITE' owsocket
     end
   end
   else event = 'TIME'

   select

    /* Accept connections from clients, receive and send messages     */
    when event='SOCKET' then do
      parse var rest keyword ts .

      /* Accept new connections from clients                          */
      if keyword='READ' & ts=s then do
        nsn = Socket('Accept',s)
        if src=0 then do
          parse var nsn ns . np nia .
          say 'RSSERVER: Connected by' nia 'on port' np 'and socket' ns
        end
      end

      /* Get nodeid, userid and number of lines to be sent            */
      if keyword='READ' & tsˆ=s then do
        parse value Socket('Recv',ts) with len nid uid count .
        if src=0 & len>0 & datatype(count,'W') then do
          if count<0 then count = 0
          if count>5000 then count = 5000
          ra = 'by' uid 'at' nid
          say 'RSSERVER: Request for' count 'lines on socket' ts ra
          linecount.ts = linecount.ts + count
          call addsock(ts)
        end
        else do
          call Socket 'Close',ts
          linecount.ts = 0
          call delsock(ts)
          say 'RSSERVER: Disconnected socket' ts
        end
      end

      /* Get nodeid, userid and number of lines to be sent            */
      if keyword='WRITE' then do
        if linecount.ts>0 then do
          num = random(1,sourceline())      /* Return random-selected */
          msg = sourceline(num) ]] '15'x    /*   line of this program */
          call Socket 'Send',ts,msg
          if src=0 then linecount.ts = linecount.ts - 1
          else linecount.ts = 0
        end
        else do
          call Socket 'Close',ts
          linecount.ts = 0
          call delsock(ts)
          say 'RSSERVER: Disconnected socket' ts
        end
      end

    end

    /* Unknown event (should not occur)                               */
    otherwise nop
  end
end

/* Terminate and exit                                                 */
call Socket 'Terminate'
say 'RSSERVER: Terminated'
exit 0

/* Procedure to add a socket to the write socket list                 */
addsock: procedure expose wlist
  s = arg(1)
  p = wordpos(s,wlist)
  if p=0 then wlist = wlist s
return

/* Procedure to del a socket from the write socket list               */
delsock: procedure expose wlist
  s = arg(1)
  p = wordpos(s,wlist)
  if p>0 then do
    templist = ''
    do i=1 to words(wlist)
      if i^=p then templist = templist word(wlist,i)
    end
    wlist = templist
  end
return

/* Calling the real SOCKET function                                   */
socket: procedure expose initialized src
  a0 = arg(1)
  a1 = arg(2)
  a2 = arg(3)
  a3 = arg(4)
  a4 = arg(5)
  a5 = arg(6)
  a6 = arg(7)
  a7 = arg(8)
  a8 = arg(9)
  a9 = arg(10)
  parse value 'SOCKET'(a0,a1,a2,a3,a4,a5,a6,a7,a8,a9) with src res
return res


/* Syntax error routine                                               */
syntax:
  call error 'E', rc, '==> REXX Error No.' 20000+rc
return

/* Halt exit routine                                                  */
halt:
  call error 'E', 4, '==> REXX Interrupted'
return

/* Error message and exit routine                                     */
error:
  type = arg(1)
  retc = arg(2)
  text = arg(3)
  ecretc = right(retc,3,'0')
  ectype = translate(type)
  ecfull = 'RXSSRV' ]] ecretc ]] ectype
  say '===> Error:' ecfull text
  if typeˆ='E' then return
  if initialized
     then do
       parse value Socket('SocketSetStatus') with . status severreason
       if statusˆ='Connected'
          then say 'The status of the socket set is' status severreason
     end
  call Socket 'Terminate'
exit retc
```

Chapter 13. Pascal Application Programming Interface (API)

This chapter describes the Pascal language application program interface (API) provided with TCP/IP. This interface allows programmers to write application programs that use the TCP, UDP, and IP layers of the TCP/IP protocol suite. Topics include:

- Software requirements
- Data structures
- Using procedure calls
- Pascal return codes
- Procedure calls
- Sample Pascal program

To use the Pascal language API, you should have experience in Pascal language programming and be familiar with the principles of internetwork communication.

For more information about sockets, refer to *UNIX Programmer's Reference Manual*.

Your program uses procedure calls to initiate communication with the TCP/IP address space. Most of these procedure calls return with a code that indicates success or the type of failure incurred by the call. The TCP/IP address space starts asynchronous communication by sending you notification.

The general sequence of operations is:

1. Start TCP/UDP/IP service (BeginTcpIp).
2. Specify the set of notifications that TCP/UDP/IP can send you (Handle).
3. Establish a connection (TcpOpen, UdpOpen, RawIpOpen, and TcpWaitOpen).

   Note: If using TcpOpen, communication must wait for the appropriate notification of connection.
4. Transfer a data buffer to or from the TCP/IP address space (TcpSend, TcpFSend, TcpWaitSend, TcpReceive, TcpFReceive, TcpWaitReceive, UdpSend, UdpNReceive, RawIpSend, UdpReceive, and RawIpReceive).

   Note: TcpWaitReceive and TcpWaitSend are synchronous calls.

   TcpFSend and TcpSend are the asynchronous ways of sending data on a TCP connection. Both procedures return to your program immediately. TcpSend does not wait under any circumstance.

   TcpSend and TcpFSend differ in how they handle the situation when TCPIP address space has insufficient buffer space to accept the data being sent.

   In the case of insufficient buffer space, TCPIP responds to TcpSend with the return code NObufferSPACE. This return code is sent back to the application. It is the application's responsibility to wait for BUFFERspaceAVAILABLE notification and resend the data.

   In the case of TcpFSend with insufficient buffer space,the PASCAL API will block until buffer space becomes available or an error is detected. This is the only condition under which TcpFSend will block.

5. Check the status returned from TCP/IP in the form of notifications (GetNextNote).
6. Repeat the data transfer operations (Steps 4 and 5) until the data is exhausted.
7. Terminate the connection (TcpClose, UdpClose, and RawIpClose).

   Note: If using TcpClose, you must wait for the connection to terminate.
8. Terminate the communication service (EndTcpIp).

Control is returned to you, in most instances, after the initiation of your request. When appropriate, some procedures have alternative wait versions that return only after request completion.

A sample program is supplied with TCP/IP. See "Sample Pascal Program" on page 526, for a listing of the sample program.

Software Requirements

To develop programs in Pascal that interface directly to the TCP, UDP, and IP protocol boundaries, you must have the IBM VS Pascal Compiler and Library (5668-767).

Pascal API Header Files

The following is a list of the headers used by Pascal applications:

- cmclien
- cmcomm
- cminter
- cmresglb

Data Structures

Programs containing Pascal language API calls must include the appropriate data structures. The data structures are declared in CMCOMM and CMCLIEN. To include these data sets in your program source, enter:
Additional include statements are required in programs that use certain calls. The

```
%include  CMCOMM
%include  CMCLIEN
```

following list shows the members that need to be included for the various calls:

- CMRESGLB for GetHostResol
- CMINTER for GetHostNumber, GetHostString, IsLocalAddress, and IsLocalHost

The load modules are in the *hlq*.SEZACMTX data set. Include this data set in your SYSLIB concatenation when you are creating a load module to link an application program. You must specify *hlq*.SEZACMTX before the Pascal libraries when linking TCP/IP programs.

Connection State

ConnectionState is the current state of the connection. See Figure 91 on page 485 for the Pascal declaration of the ConnectionStateType data type. ConnectionStateType is used in StatusInfoType and NotificationInfoType. It defines the client program view of the state of a TCP connection, in a form more readily usable than the formal TCP connection state defined by RFC 793. See Table 18 on page 485 for the mapping between TCP states and ConnectionStateType.

```
ConnectionStateType  =
(
        CONNECTIONclosing,
        LISTENING,
        NONEXISTENT,
        OPEN,
        RECEIVINGonly,
        SENDINGonly,
        TRYINGtoOPEN
);
```

Figure 91. Pascal Declaration of Connection State Type

CONNECTIONclosing
Indicates that no more data can be transmitted on this connection, because it is going through the TCP connection closing sequence.

LISTENING
Indicates that you are waiting for a foreign site to open a connection.

NONEXISTENT
Indicates that a connection no longer exists.

OPEN
Indicates that data can go either way on the connection.

RECEIVINGonly
Indicates that data can be received, but cannot be sent on this connection, because the client has done a TcpClose.

SENDINGonly
Indicates that data can be sent out, but cannot be received on this connection, because the foreign application has done a TcpClose or equivalent.

TRYINGtoOPEN
Indicates that you are trying to contact a foreign site to establish a connection.

Table 18 lists the TCP connection states.

Table 18. TCP Connection States

TCP State	ConnectionStateType
CLOSED	NONEXISTENT
LAST-ACK, CLOSING, TIME-WAIT	If there is incoming data that the client program has not received, then RECEIVINGonly, else CONNECTIONclosing.
CLOSE-WAIT	If there is incoming data that the client program has not received, then OPEN, else SENDINGonly.
ESTABLISHED	OPEN
FIN-WAIT-1, FIN-WAIT-2	RECEIVINGonly
LISTEN	LISTENING
SYN-SENT, SYN-RECEIVED	TRYINGtoOPEN

Connection Information Record

The connection information record is used as a parameter in several of the procedure calls. It enables you and the TCP/IP program to exchange information about the connection. The Pascal declaration is shown in Figure 92.

```
StatusInfoType =
      record
      Connection:  ConnectionType;
      OpenAttemptTimeout:  integer;
      Security:  SecurityType;
      Compartment:  CompartmentType;
      Precedence:  PrecedenceType;
      BytesToRead:  integer;
      UnackedBytes:  integer;
      ConnectionState:  ConnectionStateType;
      LocalSocket:  SocketType;
      ForeignSocket:  SocketType;
      end;
```

Figure 92. Pascal Declaration of Connection Information Record

Connection
Is a number identifying the connection that is described. This connection number is different from the connection number displayed by the NETSTAT command.

OpenAttemptTimeout
Is the number of seconds that TCP continues to attempt to open a connection. You specify this number. If the limit is exceeded, TCP stops trying to open the connection and shuts down any partially open connection.

Security, Compartment, Precedence
Are used only when working within a multilevel secure environment.

BytesToRead
Is the number of data bytes received from the foreign host by TCP, but not yet delivered to the client. TCP maintains this value.

UnackedBytes
Is the number of bytes sent by your program, but not yet sent to the foreign TCP, or the number of bytes sent to the foreign TCP, but not yet acknowledged.

ConnectionState

ConnectionState is the current state of the connection. ConnectionStateType defines the client program view of the state of a TCP connection, in a form more readily usable than the formal TCP connection state defined by RFC 793.

LocalSocket
Is the local internet address and local port. Together, these form one end of a connection. The foreign socket forms the other end. See Figure 93 on page 487 for the Pascal declaration of the SocketType record.

ForeignSocket
Is the foreign, or remote, internet address and its associated port. These

form one end of a connection. The local socket forms the other end. Figure 93 shows the Pascal declaration of a socket type.

```
InternetAddressType = UnsignedIntegerType;
PortType = UnsignedHalfWordType;
SocketType =
      record
      Address: InternetAddressType;
      Port: PortType;
      end;
```

Figure 93. Pascal Declaration of Socket Type

Address
Is the internet address.

Port Is the port.

Notification Record

The notification record is used to provide event information. You receive this information by using the GetNextNote call. If it is a variant record, the number of fields depends on the type of notification. See Figure 94 for the Pascal declaration of this record.

```
NotificationInfoType =
      record
      Connection: ConnectionType;
      Protocol: ProtocolType;
      case NotificationTag: NotificationEnumType of
            BUFFERspaceAVAILABLE:
                 (
                 AmountOfSpaceInBytes: integer
                 );
            CONNECTIONstateCHANGED:
                 (
                 NewState: ConnectionStateType;
                 Reason: CallReturnCodeType
                 );
            DATAdelivered:
                 (
                 BytesDelivered: integer;
                 LastUrgentByte: integer;
                 PushFlag: Boolean
                 );
```

Figure 94. Notification Record (Part 1 of 2)

```
FSENDresponse:
     (
     SendTurnCode:  CallReturnCodeType;
     SendRequestErr:  Boolean;
     );
PINGresponse:
     (
     PingTurnCode:  CallReturnCodeType;
     ElapsedTime:  TimeStampType
     );
RAWIPpacketsDELIVERED:
     (
     RawIpDataLength:  integer;
     RawIpFullLength:  integer;
     );
RAWIPspaceAVAILABLE:
     (
     RawIpSpaceInBytes:  integer;
     );
RESOURCESavailable:  ();
SMSGreceived:  ();
TIMERexpired:
     (
     Datum:  integer;
     AssociatedTimer:  TimerPointerType
     );
UDPdatagramDELIVERED:
     (
     DataLength:  integer;
     ForeignSocket:  SocketType;
     FullLength:  integer
     );
UDPdatagramSPACEavailable:  ();
UDPresourcesAVAILABLE:  ();
URGENTpending:
     (
     BytesToRead:  integer;
     UrgentSpan:  integer
     );
USERdefinedNOTIFICATION:
     (
     UserData:  UserNotificationDataType
     );
end;
```

Figure 94. Notification Record (Part 2 of 2)

Connection
Is the client's connection number to which the notification applies. In the case of USERdefinedNOTIFICATION, this field is as supplied by the user in the AddUserNote call.

Protocol
In the case of USERdefinedNOTIFICATION, this field is as supplied by the user in the AddUserNote call. For all other notifications, this field is reserved.

NotificationTag

Is the type of notification being sent, and a set of fields depends on the value of the tag. Possible tag values relevant to the TCP/UDP/IP interface and the corresponding fields are:

BUFFERspaceAVAILABLE

Notification given when space becomes available on a connection for which TcpSend previously returned NObufferSPACE.

AmountOfSpaceInBytes

The minimum number of bytes that the TCP/IP service has available for buffer space for this connection. The actual amount of buffer space might be more than this number.

CONNECTIONstateCHANGED

Indicates that a TCP connection has changed state.

NewState

The new state for this connection.

Reason

The reason for the state change. This field is meaningful only if the NewState field has a value of NONEXISTENT.

Notes:

1. The following is the sequence of state notifications for a connection.

   For active open:
   - OPEN
   - RECEIVINGonly or SENDINGonly
   - CONNECTIONclosing
   - NONEXISTENT

   For passive open:
   - OPEN
   - RECEIVINGonly or SENDINGonly
   - CONNECTIONclosing
   - NONEXISTENT

   Your program should be prepared for any intermediate step or steps to be skipped.
2. The normal TCP connection closing sequence can lead to a connection staying in CONNECTIONclosing state for up to two minutes, corresponding to the TCP state TIME-WAIT.
3. Reason codes giving the reason for a connection changing to NONEXISTENT are:
   - OK
   - UNREACHABLEnetwork
   - TIMEOUTopen
   - OPENrejected
   - REMOTEreset
   - WRONGsecORprc
   - FATALerror
   - TCPipSHUTDOWN

DATAdelivered

Notification given when your buffer (named in an earlier TcpReceive or TcpFReceive request) contains data.

Note: The data delivered should be treated as part of a byte stream, not as a message. There is no guarantee that the data sent in one TcpSend (or equivalent) call on the foreign host is delivered in a single DATAdelivered notification, even if the PushFlag is set.

BytesDelivered
Number of bytes of data delivered to you.

LastUrgentByte
Number of bytes of urgent data remaining, including data just delivered.

PushFlag
`TRUE` if the last byte of data was received with the push bit set.

FSENDresponse
Notification given when a TcpFSend request is completed, successfully or unsuccessfully.

SendTurnCode
The status of the send operation.

PINGresponse
Notification given when a PINGresponse is received.

PingTurnCode
The status of the PING operation.

ElapsedTime
The time elapsed between the sending of a request and the reception of a response. This field is valid only if PingTurnCode has a value of OK.

RAWIPpacketsDELIVERED
Notification given when your buffer (indicated in an earlier RawIpReceive request) contains a datagram. Only one datagram is delivered on each notification. Your buffer contains the entire IP header, plus as much of the datagram as fits in your buffer.

RawIpDataLength
The actual data length delivered to your buffer. If this is less than RawIpFullLength, the datagram was truncated.

RawIpFullLength
Length of the packet, from the TotalLength field of the IP header.

RAWIPspaceAVAILABLE
When space becomes available after a client does a RawIpSend and receives a NObufferSPACE return code, the client receives this notification to indicate that space is now available.

RawIpSpaceInBytes
The amount of space available always equals the maximum size IP datagram.

RESOURCESavailable
Notice given when resources needed for a TcpOpen or TcpWaitOpen are available. This notification is sent only if a previous TcpOpen or TcpWaitOpen returned ZEROresources.

SMSGreceived
Notification given when one or more special messages (Smsgs) arrive. The GetSmsg call is used to retrieve queued Smsgs.

TIMERexpired
Notification given when a timer set through SetTimer expires.

Datum
The data specified when SetTimer was called.

AssociatedTimer
The address of the timer that expired.

UDPdatagramDELIVERED
Notification given when your buffer, indicated in an earlier UdpNReceive or UdpReceive request, contains a datagram. Your buffer contains the datagram excluding the UDP header.

Note: If UdpReceive was used, your buffer contains the entire datagram excluding the header, with the length indicated by DataLength. If UdpNReceive was used, and DataLength is less than FullLength, your buffer contains a truncated datagram. The reason is that your buffer was too small to contain the entire datagram.

DataLength
Length of the data delivered to your buffer.

ForeignSocket
The source of the datagram.

FullLength
The length of the entire datagram, excluding the UDP header. This field is set only if UdpNReceive was used.

UDPdatagramSPACEavailable
Notification given when buffer space becomes available for a datagram for which UdpSend previously returned NObufferSPACE because of insufficient resources.

UDPresourcesAVAILABLE
Notice given when resources needed for a UdpOpen are available. This notification is sent only if a previous UdpOpen returned UDPzeroRESOURCES.

URGENTpending
Notification given when there is urgent data pending on a TCP connection.

BytesToRead
The number of incoming bytes not yet delivered to the client.

UrgentSpan
Number of bytes that are not delivered to the last known urgent pointer. No urgent data is pending if this is negative.

USERdefinedNOTIFICATION
Notice generated from data passed to AddUserNote by your program.

UserData
A 40-byte field supplied by your program through

AddUserNote. Connection and protocol fields also are set from the values supplied to AddUserNote.

File Specification Record

The file specification record is used to fully specify a data set. The Pascal declaration is shown in Figure 95.

```
SpecOfFileType  =
            record
            Owner:  DirectoryNameType;
            Case  SpecOfSystemType  of
           VM:
            (
                    VirtualAddress:VirtualAddressType;
                    NewVirtualAddress:VirtualAddressType;
                    DiskPassword:  DirectoryNameType;
                    Filename:  DirectoryNameType;
                    Filetype:  DirectoryNameType;
                    Filemode:  FilemodeType
           );
           MVS:
           (
                    DatasetPassword:  DirectoryNameType;
                    FullDatasetName:  DatasetNameType;
                    MemberName:  MemberNameType;
                    DDName:  DDNameType
           );
           end;
```

Figure 95. Pascal Declaration of File Specification Record

Using Procedure Calls

Your program uses procedure calls to initiate communication with the TCP/IP address space. Most of these procedure calls return with a code, which indicates success or the type of failure incurred by the call. See Table 19 on page 494 for an explanation of the return codes.

Before invoking any of the other interface procedures, use BeginTcpIp to start the TCP/UDP/IP service. Once the TCP/UDP/IP service has begun, use the Handle procedure to specify a set of notifications that the TCP/UDP/IP service can send you. To terminate the TCP/UDP/IP service, use the EndTcpIp procedure.

Notifications

The TCP/IP address space notifies you of asynchronous events. Also, some notifications are generated in your address space by the TCP interface. Notifications can be received only after BeginTcIp.

The notifications are received by the TCP interface and kept in a queue. Use GetNextNote to get the next notification. The notifications are in Pascal variant record form. See Figure 94 on page 487 for more information.

TCP Initialization Procedures

The TCP Initialization procedures affect all present and future connections. Use these procedures to initialize the TCPenvironment for your program.

TCP Termination Procedure

The Pascal API has one termination procedure call. Use the EndTcpIp call when you have finished with the TCP/IP services.

TCP Communication Procedures

The TCP communication procedures apply to a particular client connection. Use these procedures to establish a connection and to communicate. You must call the BeginTcpIp initialization routine before you can begin using TCP communication procedures.

PING Interface

The Ping interface lets a client send an ICMP echo request to a foreign host. You must call the BeginTcpIp initialization routine before you can begin using the PING Interface.

Monitor Procedures

The MonQuery monitor procedure provides a mechanism for querying the TCP/IP address space.

Any program using this monitor procedure must include CMCOMM and CMCLIEN.

UDP Communication Procedures

The UDP communication procedures describe the programming interface for the User Datagram Protocol (UDP) provided in the TCP/IP product.

Raw IP Interface

The Raw IP interface lets a client program send and receive arbitrary IP datagrams on any IP Internet protocol except TCP and UDP. Only one client can use any given protocol at one time. Only clients that are APF-authorized can use the Raw IP interface.

Timer Routines

The timer routines are used with the TCP/UDP/IP interface. You must call the BeginTcpIp initialization routine before you can begin using the timer routines.

Host Lookup Routines

The host lookup routines (with the exception of GetHostResol) are declared in the CMINTER member of the *hlq*.SEZACMAC data set. The host lookup routine GetHostResol is declared in the CMRESGLB member of the *hlq*.SEZACMAC data set. Any program using these procedures must include CMINTER or CMRESGLB after the INCLUDE statements for CMCOMM and CMCLIEN.

Assembler Calls

AddUserNote is provided and can be called directly from an assembler language interrupt handler.

Other Routines

This group includes the following procedures.

- GetSmsg
- ReadXlateTable
- SayCalRe
- SayConSt
- SayIntAd
- SayIntNum
- SayNotEn
- SayPorTy
- SayProTy

Pascal Return Codes

When using Pascal procedure calls, check to determine whether the call has been completed successfully. Use the SayCalRe function (see “SayCalRe” on page 508) to convert the ReturnCode parameter to a printable form.

The SayCalRe function converts a return value into a descriptive message. For example, if SayCalRe is invoked with the return value BADlengthARGUMENT, it returns the message `invalid length specified`. See Table 19 for a description of Pascal return codes and their equivalent message text from SayCalRe.

Most return values are self-explanatory in the context where they occur. The return codes you see as a result of issuing a TCP/UDP/IP request are in the range –128 to 0.

Table 19. Pascal Language Return Codes

Return Value	Return Code	Message Text
OK	0	OK.
ABNORMALcondition	–1	Abnormal condition due to CSA storage shortage.
ALREADYclosing	–2	Connection is already closing.
BADlengthARGUMENT	–3	Invalid length specified.
CANNOTsendDATA	–4	Cannot send data.
CLIENTrestart	–5	Client reinitialized TCP/IP service.
CONNECTIONalreadyEXISTS	–7	Connection already exists.
ERRORinPROFILE	–8	Error in profile data set. Details are in PROFILE.TCPERROR or the //SYSERROR DD file.
FATALerror	–9	Fatal error; not valid user parameter (storage key).
HASnoPASSWORD	–10	No password is in the RACF directory.
INCORRECTpassword	–11	TCPIP is not authorized to access the data set.
INVALIDrequest	–12	Request not valid.
INVALIDuserID	–13	User ID not valid.
INVALIDvirtualADDRESS	–14	Virtual address not valid.
LOCALportNOTavailable	–16	The requested local port is not available.
NObufferSPACE (TcpSend only)	–19	No more space for data currently available. This applies to this connection only; space might still be available for other connections.

Table 19. Pascal Language Return Codes (continued)

Return Value	Return Code	Message Text
NONlocalADDRESS	–21	The internet address is not local to this host.
NOoutstandingNOTIFICATIONS	–22	No outstanding notifications.
NOsuchCONNECTION	–23	No such connection.
NOtcpIPservice	–24	No TCP/IP service is available.
NOTyetBEGUN	–25	TCP/IP service not yet begun.
NOTyetOPEN	–26	The connection is not yet open.
OPENrejected	–27	Foreign host rejected the open attempt.
PARAMlocalADDRESS	–28	TcpOpen error: local address not valid.
PARAMstate	–29	TcpOpen error: initial state not valid.
PARAMtimeout	–30	Time-out parameter not valid.
PARAMunspecADDRESS	–31	TcpOpen error: unspecified foreign address in active open.
PARAMunspecPORT	–32	TcpOpen error: unspecified foreign port in active open.
PROFILEnotFOUND	–33	TCPIP cannot read PROFILE data set.
RECEIVEstillPENDING	–34	Receive is still pending on this connection.
REMOTEclose	–35	Foreign host unexpectedly closed the connection.
REMOTEreset	–36	Foreign host abended the connection.
SOFTWAREerror	–37	Software error in TCP/IP.
TCPipSHUTDOWN	–38	TCP/IP is being shut down.
TIMEOUTopen	–40	Foreign host did not respond within OPEN time-out.
TOOmanyOPENS	–41	Too many open connections exist already .
UNAUTHORIZEDuser	–43	You are not authorized to issue this command.
UNIMPLEMENTEDrequest	–45	TCP/IP request not implemented.
UNREACHABLEnetwork	–47	Destination network cannot be reached.
UNSPECIFIEDconnection	–48	Connection not specified.
VIRTUALmemoryTOOsmall	–49	Client address space has too little storage.
WRONGsecORprc	–50	Foreign host disagreed on security or precedence.
ZEROresources	–56	TCP cannot handle more connections now.
UDPlocalADDRESS	–57	Local address for UDP not correct.
UDPunspecADDRESS	–59	Address was not specified; specification is necessary.
UDPunspecPORT	–60	Port was unspecified; specification is necessary.
UDPzeroRESOURCES	–61	UDP cannot handle more traffic.
FSENDstillPENDING	–62	FSend still pending on this connection.
ERRORopeningORreadingFILE	–80	Error opening or reading data set.
FILEformatINVALID	–81	File format is not valid.
SAYCALRE*	–130	Unknown TCP return code.

* Invalid return codes (out of the range -128 to 0) return Unknown TCP return codes when translated using SAYCALRE.

Procedure Calls

This section provides the syntax, parameters, and other appropriate information for each Pascal procedure call supported by TCP/IP.

AddUserNote

This procedure can be called from assembler language code to add a USERdefinedNOTIFICATION notification to the note queue and cause the initiation of GetNextNote if it is waiting for a notification. Figure 96 shows a sample calling sequence.

```
           LA    R13,PASCSAVE
           LA    R1,PASCPARM
           L     R15,=V(ADDUSERN)
           BALR  R14,R15
                  .
                  .
PASCSAVE   DS    18F     Register save area
ENV        DC    F'0'    Zero initially.  It is filled with
                         an environment address.  Pass it unchanged
                         in subsequent calls to ADDUSERN.
DATA1      DS    H       Data for Connection field of notification.
DATA2      DS    C       Data for Protocol field of notification.
DATA3      DS    XL40    Data for UserData field of notification.
RC         DS    F       AddUserNote stores return code here.

PASCPARM   DC    A(ENV)
           DC    A(DATA1)
           DC    A(DATA2)
           DC    A(DATA3)
           DC    A(RC)
```

Figure 96. Sample Calling Sequence

Parameter **Description**

ReturnCode Indicates the success or failure of the call. Possible return values are:
- OK
- NObufferSPACE

BeginTcpIp

Use BeginTcpIp to inform the TCP/IP address space that you want to start using its services as show in Figure 97.

```
   procedure  BeginTcpIp
              (
        var     ReturnCode:  integer
              );
              external;
```

Figure 97. BeginTcpIp Example

Parameter **Description**

ReturnCode Indicates success or failure of call. Possible return values are:
- OK

- ABNORMALcondition
- FATALerror
- NOtcpIPservice
- TCPipSHUTDOWN
- VIRTUALmemoryTOOsmall

For a description of the Pascal return codes, see Table 19 on page 494.

ClearTimer

This procedure resets the timer to prevent it timing out as shown in Figure 98.

```
procedure  ClearTimer
           (
                T:  TimerPointerType
           );
           external;
```

Figure 98. ClearTimer Example

Parameter	Description
T	A timer pointer, as returned by a previous CreateTimer call.

CreateTimer

This procedure allocates a timer. The timer is not set in any way. See “SetTimer” on page 511 to activate the timer. Figure 99 shows an example.

```
procedure  CreateTimer
           (
      var       T:  TimerPointerType
           );
           external;
```

Figure 99. CreateTimer Example

Parameter	Description
T	Is set to a timer pointer that can be used in subsequent SetTimer, ClearTimer, and DestroyTimer calls.

DestroyTimer

This procedure deallocates (frees) a timer you created. Figure 100 shows an example.

```
procedure  DestroyTimer
           (
      var       T:  TimerPointerType
           );
           external;
```

Figure 100. DestroyTimer Example

Parameter **Description**

T	A timer pointer, as returned by a previous CreateTimer call.

EndTcpIp

Use EndTcpIp when you have finished with the TCP/IP services. The procedure shown in Figure 101 releases ports and protocols in use that are not permanently reserved. It causes TCP to clean up the data structures it has associated with you.

```
procedure EndTcpIp;
          external;
```

Figure 101. EndTcpIp Example

GetHostNumber

The GetHostNumber procedure resolves a host name into an internet address. This is shown in Figure 102.

GetHostNumber uses a table lookup to convert the name of a host to an internet address, and returns this address in the HostNumber field. When the name is a dotted decimal number, GetHostNumber returns the integer represented by that dotted decimal. The dotted decimal representation of a 32-bit number has one decimal integer for each of the four bytes, separated by dots. For example, `14.0.0.7` for X'0E000007'. See *OS/390 IBM Communications Server: IP Configuration Reference* for information about how to create host lookup tables.

The HostNumber field is set to NOhost if the host is not found.

```
procedure GetHostNumber
          (
     const     Name:  string;
     var         HostNumber:  InternetAddressType
          );
          external;
```

Figure 102. GetHostNumber Example

Parameter	Description
Name	The name or dotted decimal number to be converted.
HostNumber	Set to the converted address, or NOhost if conversion fails.

GetHostResol

The GetHostResol procedure converts a host name into an internet address by using a name server. Figure 103 on page 499 shows an example.

GetHostResol passes the query to the remote name server through the resolver. The name server converts the name of a host to an internet address, and returns this address in the HostNumber field. If the name server does not respond or does not find the name, the host name is converted to a host number by table lookup. When the name is a dotted decimal number, the integer represented by that dotted decimal is returned. The dotted decimal representation of a 32-bit number has one decimal integer for each of the four bytes, separated by dots. For example, `14.0.0.7` for X'0E000007'.

The HostNumber field is set to NOhost if the host is not found.

```
   procedure  GetHostResol
              (
        const       Name:  string;
        var           HostNumber:  InternetAddressType
              );
              external;
```

Figure 103. GetHostResol Example

Parameter	Description
Name	The name or dotted decimal number to be converted.
HostNumber	Set to the converted address, or NOhost if conversion fails.

GetHostString

The GetHostString procedure call uses a table lookup to convert an internet address to a host name, and returns this string in the Name field. The first host name found in the lookup is returned. If no host name is found, a gateway or network name is returned. If no gateway or network name is found, a null string is returned. An example is shown in Figure 104.

```
   procedure  GetHostString
              (
                    Address:  InternetAddressType;
        var      Name:  SiteNameType
              );
              external;
```

Figure 104. GetHostString Example

Parameter	Description
Address	The address to be converted. The address must be in dotted decimal format.
Name	Set to the corresponding host, gateway, or network name, or to null string if a match is not found.

GetIdentity

This procedure returns the following information:

- The user ID of the MVS user
- The host machine name
- The network domain name
- The user ID of the TCP/IP address space

The host machine name and domain name are extracted from the HostName and DomainOrigin statements, respectively, in the *user_id*.TCPIP.DATA data set. If the *user_id*.TCPIP.DATA data set does not exist, the *hlq*.TCPIP.DATA data set is used. If a HostName statement is not specified, then the default host machine name is the name specified by the TCP/IP installer during installation. See *OS/390 IBM Communications Server: IP Configuration Reference*. The TCP/IP address space user ID is extracted from the TcpipUserid statement in the *user_id*.TCPIP.DATA data set; if the statement is not specified, the default is TCP/IP.

Figure 105 shows the GetIdentity procedure.

```
procedure  GetIdentity
           (
     var      UserId:   DirectoryNameType;
     var      HostName, DomainName: String;
     var      TcpIpServiceName: DirectoryNameType;
     var      Result: integer
           );
           external;
```

Figure 105. GetIdentity Example

Parameter **Description**

UserId The user ID of the TSO user or the job name of a batch job that has invoked GetIdentity.

HostName The host machine name.

DomainName The network domain name.

TcpIpServiceName
The user ID of the TCP/IP address space.

Result Indicates success or failure of the call.

GetNextNote

Use this procedure to retrieve notifications from the queue. This procedure returns the next notification queued for you. Figure 106 shows an example of the GetNextNote procedure.

```
procedure  GetNextNote
           (
     var      Note: NotificationInfoType;
                 ShouldWait: Boolean;
     var      ReturnCode: integer
           );
           external;
```

Figure 106. GetNextNote Example

Parameter **Description**

Note Next notification is stored here when ReturnCode is OK.

ShouldWait Set ShouldWait to `TRUE` if you want GetNextNote to wait until a notification becomes available. Set ShouldWait to `FALSE` if you want GetNextNote to return immediately. When ShouldWait is set to `FALSE`, ReturnCode is set to NOoutstandingNOTIFICATIONS if notification is not currently queued.

ReturnCode Indicates success or failure of call. Possible return values are:
- OK
- NOoutstandingNOTIFICATIONS
- NOTyetBEGUN

For a description of Pascal return codes, see Table 19 on page 494.

GetSmsg

Your program should call this procedure after receiving an SMSGreceived notification. Each call to GetSmsg retrieves one queued Smsg. Your program should exhaust all queued Smsgs, by calling GetSmsg repeatedly until the Success field returns with a value of FALSE. After a value of FALSE is returned, do not call GetSmsg again until you receive another SMSGreceived notification. Figure 107 shows an example of the GetSmsg procedure.

```
procedure  GetSMsg
           (
     var      Smsg:  SmsgType;
     var      Success:  Boolean;
           );
           external;
```

Figure 107. GetSmsg Example

Parameter	Description
Smsg	Set to the returned Smsg if Success is set to TRUE.
Success	If Smsg returned TRUE; otherwise FALSE.

Handle

Use the Handle procedure to specify that you want to receive notifications in the given set as shown in Figure 108. You must always use it after calling the BeginTcpIp procedure and before accessing the TCP/IP services. This Pascal set of notifications can contain any of the NotificationEnumType values shown in Figure 94 on page 487.

```
procedure  Handle
           (
                 Notifications:  NotificationSetType;
     var      ReturnCode:  integer
           );
           external;
```

Figure 108. Handle Example

Parameter	Description
Notifications	The set of notification types to be handled.
ReturnCode	Indicates success or failure of the call. Possible return values are: • OK • NOTyetBEGUN

For a description of Pascal return codes, see Table 19 on page 494.

IsLocalAddress

This procedure queries the TCP/IP address space to determine whether the HostAddress is one of the addresses recognized for this host. If the address is local, it returns OK. If the address is not local, it returns NONlocalADDRESS. Figure 109 on page 502 shows an example.

```
procedure  IsLocalAddress
           (
                HostAddress:  InternetAddressType;
      var       ReturnCode:  integer
           );
           external;
```

Figure 109. IsLocalAddress Example

Parameter	Description
HostAddress	The host address to be tested.
ReturnCode	Indicates whether the host address is local, or it might indicate an error. Possible return values are: • OK • NONlocalADDRESS • TCPipSHUTDOWN • FATALerror • SOFTWAREerror

For a description of Pascal return codes, see Table 19 on page 494.

IsLocalHost

This procedure returns the correct host class for Name, which can be a host name or a dotted decimal address. Figure 110 shows an example of the IsLocalHost procedure.

The host classes are:

HOSTlocal
Is an internet address for the local host.

HOSTloopback
Is one of the dummy internet addresses used to designate various levels of loopback testing.

HOSTremote
Is a known host name for some remote host.

HOSTunknown
Is an unknown host name (or other error).

```
procedure  IsLocalHost
           (
      const       Name:  string;
      var           Class:  HostClassType
           );
           external;
```

Figure 110. IsLocalHost Example

Parameter	Description
Name	The host name.
Class	The host class.

MonQuery

The MonQuery procedure is used to obtain status information or to request TCP/IP to perform certain actions. This procedure is used by the NETSTAT command as shown in Figure 111. For more information about the NETSTAT command, refer to the *OS/390 IBM Communications Server: IP User's Guide*.

```
procedure  MonQuery
           (
               QueryRecord:  MonQueryRecordType;
               Buffer:  integer;
               BufSize:  integer;
      var    ReturnCode:  integer;
      var    Length:  integer
           );
           external;
```

Figure 111. MonQuery Example

Parameter **Description**

Buffer The address of the buffer to receive data.

BufSize The size of the buffer.

ReturnCode Indicates success or failure of the call. Possible return values are:
- OK
- FATALerror
- NOTyetBEGUN
- TCPipSHUTDOWN
- UNIMPLEMENTEDrequest
- SOFTWAREerror

Length The length of the data returned in the buffer.

QueryRecord Your program sets up a QueryRecord to specify the type of status information to be retrieved. The MonQueryRecordType is shown in Figure 112.

```
MonQueryRecordType  =
          record
          case  QueryType:   MonQueryType  of
                        QUERYhomeONLY:  ();
          end;  {  MonQueryRecordType  }
```

Figure 112. Monitor Query Record

The only QueryType values available for customer use is:

QUERYhomeONLY
Is used to obtain a list of the home internet addresses recognized by your TCP/IP. Your program sets the Buffer to the address of a variable of type HomeOnlyListType, and the BufSize to its length. When MonQuery returns, Length is set to the length of the Buffer that was used, if ReturnCode is OK. Divide the Length by size of (InternetAddressType) to get the number of the home addresses that are returned.

For a description of Pascal return codes, see Table 19 on page 494.

PingRequest

Use this procedure to send an ICMP echo request to a foreign host. When a response is received or the time-out limit is reached, you receive a PingResponse notification.

The PingRequest procedure is used by the PING user command as shown in Figure 113. Refer to the *TCP/IP for MVS: User’s Guide* for more information about the PING command.

```
procedure  PingRequest
                (
                      ForeignAddress:  InternetAddressType;
                      Length:  integer;
                      Timeout:  integer;
           var        ReturnCode:  integer
                );
                external;
```

Figure 113. PingRequest Example

Parameter **Description**

ForeignAddress
The address of the foreign host to receive a PING.

Length Indicates the length of the PING packet, excluding the IP header. The range of values for this field is 8—65507 bytes.

Timeout The amount of time to wait for a response, in seconds.

ReturnCode Indicates success or failure of a call. Possible return values are:
- OK
- ABNORMALcondition
- BADlengthARGUMENT
- CONNECTIONalreadyEXISTS
- VIRTUALmemoryTOOsmall
- NOTyetBEGUN
- TIMEOUTopen
- PARAMtimeout
- SOFTWAREerror
- TCPipSHUTDOWN
- UNAUTHORIZEDuser

For a description of Pascal return codes, see Table 19 on page 494.

Note: CONNECTIONalreadyEXISTS, in this context, means a PING request is outstanding.

RawIpClose

This procedure tells the TCP/IP address space that the client does not handle the protocol any longer. Any queued incoming packets are discarded. Figure 114 on page 505 shows an example of the RawIpClose procedure.

When the client is not handling the protocol, a return code of NOsuchCONNECTION is received.

```
procedure  RawIpClose
           (
                 ProtocolNo:  integer;
      var      ReturnCode:  integer
           );
           external;
```

Figure 114. RawIpClose Example

Parameter **Description**

ProtocolNo The number of the Internet protocol.

ReturnCode Indicates the success or failure of a call. Possible return values are:
- OK
- NOsuchCONNECTION
- NOTyetBEGUN
- SOFTWAREerror
- TCPipSHUTDOWN
- UNAUTHORIZEDuser

For a description of Pascal return codes, see Table 19 on page 494.

RawIpOpen

This procedure tells the TCP/IP address space that the client wants to send and receive packets of the specified protocol. Figure 115 shows an example.

Do not use protocols 6 and 17. They specify the TCP (6) and UDP (17) protocols. When you specify 6, 17, or a protocol that has been opened by another address space, you receive the LOCALportNOTavailable return code.

```
procedure  RawIpOpen
           (
                 ProtocolNo:  integer;
      var      ReturnCode:  integer
           );
           external;
```

Figure 115. RawIpOpen Example

Parameter **Description**

ProtocolNo The number of the Internet protocol.

ReturnCode Indicates success or failure of a call. Possible return values are:
- OK
- LOCALportNOTavailable
- NOTyetBEGUN
- SOFTWAREerror
- TCPipSHUTDOWN
- UNAUTHORIZEDuser

For a description of Pascal return codes, see Table 19 on page 494.

Note: You can open the ICMP protocol, but your program receives only those ICMP packets not interpreted by the TCP/IP address space.

RawIpReceive

Use the procedure shown in Figure 116 to specify a buffer to receive Raw IP datagrams of the specified protocol. You get the notification RAWIPpacketsDELIVERED when a packet is put in the buffer.

```
procedure RawIpReceive
          (
                ProtocolNo:  integer;
                Buffer:  Address31Type;
                BufferLength:  integer;
      var       ReturnCode:  integer
          );
          external;
```

Figure 116. RawIpReceive Example

Parameter **Description**

ProtocolNo The number of the Internet protocol.

Buffer The address of your buffer.

BufferLength The length of your buffer. If you specify a length greater than 65535 bytes, only the first 65535 bytes are used.

ReturnCode Indicates success or failure of a call. Possible return values are:

- OK
- NOsuchCONNECTION
- NOTyetBEGUN
- SOFTWAREerror
- TCPipSHUTDOWN
- UNAUTHORIZEDuser
- INVALIDvirtualADDRESS

For a description of Pascal return codes, see Table 19 on page 494.

RawIpSend

This procedure shown in this example sends IP datagrams of the given protocol number. The entire packet, including the IP header, must be in the buffer. The TCP/IP address space uses the total length field of the IP header to determine where each packet ends. Subsequent packets begin at the next doubleword (eight-byte) boundary within the buffer.

The packets in your buffer are transmitted unchanged with the following exceptions:

- They can be fragmented; the fragment offset and flag fields in the header are filled.
- The version field in the header is filled.
- The `checksum` field in the header is filled.
- The source address field in the header is filled.

You get the return code NOsuchCONNECTION if the client is not handling the protocol, or if a packet in the buffer has another protocol. The return code BADlengthARGUMENT is received when:

- The DataLength is fewer than 40 bytes, or greater than 65535 bytes.
- NumPackets is zero.

- All packets do not fit into DataLength.

A ReturnCode value of NObufferSPACE indicates that the data is rejected, because TCP/IP is out of buffers. When buffer space is available, the notification RAWIPspaceAVAILABLE is sent to the client.

```
procedure  RawIpSend
           (
                ProtocolNo:  integer;
                Buffer:  Address31Type;
                DataLength:  integer;
                NumPackets:  integers;
      var       ReturnCode:  integer
           );
           external;
```

Figure 117. RawIpSend Example

Parameter **Description**

ProtocolNo The number of the Internet protocol.

Buffer The address of your buffer containing packets to send.

DataLength The total length of data in your buffer.

NumPackets The number of packets in your buffer.

ReturnCode Indicates the success or failure of a call. Possible return values are:
- OK
- BADlengthARGUMENT
- NObufferSPACE
- NOsuchCONNECTION
- NOTyetBEGUN
- SOFTWAREerror
- TCPipSHUTDOWN
- UNAUTHORIZEDuser
- INVALIDvirtualADDRESS

Note: If your buffer contains multiple packets waiting to be sent, some of the packets might have been sent even if ReturnCode is not OK.

For a description of Pascal return codes, see Table 19 on page 494.

ReadXlateTable

The procedure shown in Figure 118 on page 508 reads the binary translation table data set specified by TableName, and fills in the AtoETable and EtoATable translation tables.

```
procedure ReadXlateTable
          (
    var      TableName: DirectoryNameType;
    var      AtoETable: AtoEType;
    var      EtoATable: EtoAType;
    var      TranslateTableSpec: SpecOfFileType;
    var      ReturnCode: integer
          );
          external;
```

Figure 118. ReadXlateTable Example

Parameter **Description**

TableName The name of the translate table. ReadXlateTable tries to read *user_id*.TableName.TCPXLBIN. If that data set exists but it has an incorrect format, ReadXlateTable returns with a ReturnCode FILEformatINVALID. If *user_id*.TableName.TCPXLBIN does not exist, ReadXlateTable tries to read *hlq*.TableName.TCPXLBIN. ReturnCode reflects the status of reading that data set.

AtoETable Filled with ASCII-to-EBCDIC table if return code is OK.

EtoATable Filled with EBCDIC-to-ASCII table if return code is OK.

TranslateTableSpec
If ReturnCode is OK, TranslateTableSpec contains the complete specification of the data set that ReadXlateTable used. If ReturnCode is not OK, TranslateTableSpec contains the complete specification of the last data set that ReadXlateTable tried to use.

ReturnCode Indicates success or failure of a call. Possible return values are:
- OK
- ERRORopeningORreadingFILE
- FILEformatINVALID

SayCalRe

This function returns a printable string describing the return code passed in CallReturn. Figure 119 shows an example.

```
function SayCalRe
         )
               CallReturn: integer
         ):
         WordType;
         external;
```

Figure 119. SayCalRe Example

Parameter **Description**
CallReturn The return code to be described.

SayConSt

This function returns a printable string describing the connection state passed in State. For example, if SayConSt is invoked with the type identifier RECEIVINGonly, it returns the message `Receiving only`. Figure 120 shows an example of this procedure.

```
function SayConSt
          (
               State: ConnectionStateType
          ):
          Wordtype;
          external;
```

Figure 120. SayConSt Example

Parameter **Description**

State The connection state to be described.

SayIntAd

This function converts the internet address specified by InternetAddress to a printable string. The address is looked up in *hlq*.HOSTS.ADDRINFO, and the name is returned if found. If it is not found, the dotted decimal format of the address is returned. Figure 121 shows an example of this procedure.

```
function SayIntAd
          (
               InternetAddress: InternetAddressType
          ):
          WordType;
          external;
```

Figure 121. SayIntAd Example

Parameter **Description**

InternetAddress
The internet address to be converted.

SayIntNum

This function converts the internet address specified by InternetAddress to a printable string, in dotted decimal form as shown in Figure 122.

```
function SayIntNum
          (
               InternetAddress: InternetAddressType
          ):
          Wordtype;
          external;
```

Figure 122. SayIntNum Example

Parameter **Description**

InternetAddress
The internet address to be converted.

SayNotEn

This function returns a printable string describing the notification enumeration type passed in Notification. For example, if SayNotEn is invoked with the type identifier FSENDreponse, it returns the message ″Fsend response″.

```
function  SayNotEn
          (
                Notification:  NotificationEnumType
          );
          Wordtype;
          external;
```

Figure 123. SayNotEn Example

Parameter	Description
Notification	The notification enumeration type to be described.

SayPorTy

This function returns a printable string describing the port number passed in Port, if it is a well-known port number such as port number 23, the Telnet port. Otherwise, the EBCDIC representation of the number is returned. Figure 124 shows an example of this function.

```
function  SayPorTy
          (
                Port:  PortType
          ):
          WordType;
          external;
```

Figure 124. SayPorTy Example

Parameter	Description
Port	The port number to be described.

SayProTy

This function converts the protocol type specified by Protocol to a printable string, if it is a well-known protocol number, such as 6 (TCP). Otherwise, the EBCDIC representation of the number is returned. Figure 125 shows an example of this function.

```
function  SayProTy
          (
                Protocol:  ProtocolType
          ):
          WordType;
          external;
```

Figure 125. SayProTy Example

Parameter	Description
Protocol	The number of the protocol to be described.

SetTimer

The procedure shown in Figure 126 sets a timer to expire after a specified time interval. Specify the amount of time in seconds. When it times out, you receive the TIMERexpired notification, which contains the data and the timer pointer.

Note: This procedure resets any previous time interval set on this timer.

```
procedure  SetTimer
           (
                T:  TimerPointerType;
                AmountOfTime:  integer;
                Data:  integer
           );
           external;
```

Figure 126. SetTimer Example

Parameter **Description**

T A timer pointer, as returned by a previous CreateTimer call.

AmountOfTime
The time interval in seconds.

Data An integer value to be returned with the TIMERexpired notification.

TcpAbort

Use the procedure shown in Figure 127 to shut down a specific connection immediately. Data sent by your application on the abended connection might be lost. TCP sends a reset packet to notify the foreign host that you have abended the connection, but there is no guarantee that the reset will be received by the foreign host.

```
procedure  TcpAbort
           (
                Connection:  ConnectionType;
      var       ReturnCode:  integer
           );
           external;
```

Figure 127. TcpAbort Example

Parameter **Description**

Connection The connection number, as returned by TcpOpen or TcpWaitOpen in the Connection field of the StatusInfoType record.

ReturnCode Indicates success or failure of call. Possible return values are:
- OK
- ABNORMALcondition
- FATALerror
- NOsuchCONNECTION
- NOTyetBEGUN
- TCPipSHUTDOWN
- SOFTWAREerror
- REMOTEreset

The connection is fully terminated when you receive the notification CONNECTIONstateCHANGED with the NewState field set to NONEXISTENT.

For a description of Pascal return codes, see Table 19 on page 494.

TcpClose

Use the procedure shown in Figure 128 to begin the TCP one-way closing sequence. During this closing sequence, you, the local client, cannot send any more data. Data might be delivered to you until the foreign application also closes. TcpClose also causes all data sent on that connection by your application, and buffered by TCPIP, to be sent to the foreign application immediately.

```
procedure  TcpClose
           (
                 Connection:  ConnectionType;
      var      ReturnCode:  integer
           );
           external;
```

Figure 128. TcpClose Example

Parameter	Description
Connection	The connection number, as returned by TcpOpen or TcpWaitOpen in the Connection field of the StatusInfoType record.
ReturnCode	Indicates success or failure of call. Possible return values are: • OK • ABNORMALcondition • ALREADYclosing • NOsuchCONNECTION • NOTyetBEGUN • TCPipSHUTDOWN • SOFTWAREError • REMOTEreset

For a description of Pascal return codes, see Table 19 on page 494.

Notes:

1. If you receive the notification CONNECTIONstateCHANGED with a NewState of SENDINGonly, the remote application has done TcpClose (or an equivalent function) and is receiving only. Respond with TcpClose when you finish sending data on the connection.
2. The connection is fully closed when you receive the notification CONNECTIONstateCHANGED, with a NewState field set to NONEXISTENT.

TcpFReceive, TcpReceive, and TcpWaitReceive

The examples in this section illustrate TcpFReceive, TcpReceive, and TcpWaitReceive.

TcpFReceive and TcpReceive are the asynchronous ways of specifying a buffer to receive data for a given connection. Both procedures return to your program immediately. A return code of OK means that the request has been accepted. When received data has been placed in your buffer, your program receives a DATAdelivered notification.

TcpWaitReceive is the synchronous interface for receiving data from a TCP connection. TcpWaitReceive does not return to your program until data has been received into your buffer, or until an error occurs. Therefore, it is not necessary that TcpWaitReceive receive a notification when data is delivered. The BytesRead parameter is set to the number of bytes received by the data delivery, but if the number is less than zero, the parameter indicates an error.

```
procedure  TcpFReceive
           (
                Connection:  ConnectionType;
                Buffer:  Address31Type;
                BytesToRead:  integer;
         var    ReturnCode:  integer
           );
           external;
```

Figure 129. TcpFReceive Example

```
procedure  TcpReceive
           (
                Connection:  ConnectionType;
                Buffer:  Address31Type;
                BytesToRead:  integer;
      var      ReturnCode:  integer
           );
           external;
```

Figure 130. TcpReceive Example

```
procedure  TcpWaitReceive
           (
                Connection:  ConnectionType;
                Buffer:  Address31Type;
                BytesToRead:  integer;
      var      BytesRead:  integer
           );
           external;
```

Figure 131. TcpWaitReceive Example

Parameter **Description**

Connection The connection number, as returned by TcpOpen or TcpWaitOpen in the Connection field of the StatusInfoType record.

Buffer The address of the buffer to contain the received data.

BytesToRead The size of the buffer. TCP/IP usually buffers the incoming data until this many bytes are received. Data is delivered sooner if the sender specified the PushFlag, or if the sender does a TcpClose or equivalent.

Note: The order of TcpFReceive or TcpReceive calls on multiple connections, and the order of DATAdelivered notifications among the connections, are not necessarily related.

BytesRead Is set when TcpWaitReceive returns. If it is greater than zero, it indicates the number of bytes received into your buffer. If it is less than or equal to zero, it indicates an error. Possible BytesRead values are:

- OK^{+}
- ABNORMALcondition
- FATALerror
- TIMEOUTopen^{+}
- UNREACHABLEnetwork^{+}
- BADlengthARGUMENT
- NOsuchCONNECTION
- NOTyetBEGUN
- NOTyetOPEN
- OPENrejected^{+}
- RECEIVEstillPENDING
- REMOTEreset^{+}
- TCPipSHUTDOWN^{+}
- REMOTEclose

ReturnCode Indicates success or failure of call. Possible return values are:

- OK
- ABNORMALcondition
- BADlengthARGUMENT
- FATALerror
- NOsuchCONNECTION
- NOTyetBEGUN
- NOTyetOPEN
- RECEIVEstillPENDING
- REMOTEclose
- TCPipSHUTDOWN
- INVALIDvirtualADDRESS
- SOFTWAREerror

For a description of Pascal return codes, see Table 19 on page 494.

Notes™ (TcpWaitReceive):

1. For BytesRead OK, the function was initiated, but the connection is no longer receiving for an unspecified reason. Your program does not have to issue TcpClose, but the connection is not completely terminated until a NONEXISTENT notification is received for the connection.
2. For BytesRead REMOTEclose, the foreign host has closed the connection. Your program should respond with TcpClose.
3. If you receive any of the codes marked with $^{+}$, the function was initiated but the connection has now been terminated (see 2 on page 489). Your program should not issue TcpClose, but the connection is not completely terminated until NONEXISTENT notification is received for the connection.
4. TcpWaitReceive is intended to be used by programs that manage a single TCP connection. It is not suitable for use by multiple connection servers.
5. A return code of TCPipSHUTDOWN can be returned either because the connection initiation has failed, or because the connection has been terminated, because of shutdown. In either case, your program should not issue any more TCP/IP calls.

TcpFSend, TcpSend, and TcpWaitSend

The examples in this section illustrate TcpFSend, TcpSend, and TcpWaitSend.

TcpFSend and TcpSend are the asynchronous ways of sending data on a TCP connection. Both procedures return to your program immediately (do not wait under any circumstance).

TcpWaitSend is a simple synchronous method of sending data on a TCP connection. It does not return immediately if the TCPIP address space has insufficient space to accept the data being sent.

In the case of insufficient buffer space, when space becomes available, a BUFFERspaceAVAILABLE notification is received.

Your program can issue successive TcpWaitSend calls. Buffer shortage conditions are handled transparently. Errors at this point are most likely unable to recover, or are caused by a terminated connection.

If you receive any of the codes listed for Reason in the CONNECTIONstateCHANGED notification, except for OK, the connection was terminated for the indicated reason. Your program should not issue a TcpClose, but the connection is not completely terminated until your program receives a NONEXISTENT notification for the connection.

```
procedure TcpFSend
          (
               Connection: ConnectionType;
               Buffer: Address31Type;
               BufferLength: integer;
               PushFlag: Boolean;
               UrgentFlag: Boolean;
     var     ReturnCode: integer
          );
          external;
```

Figure 132. TcpFSend Example

```
procedure TcpSend
      (
               Connection: ConnectionType;
               Buffer: Address31Type;
               BufferLength: integer;
               PushFlag: Boolean;
               UrgentFlag: Boolean;
          var  ReturnCode: integer
      );
      external;
```

Figure 133. TcpSend Example

```
procedure  TcpWaitSend
           (
                Connection:  ConnectionType;
                Buffer:  Address31Type;
                BufferLength:  integer;
                PushFlag:  Boolean;
                UrgentFlag:  Boolean;
      var      ReturnCode:  integer
           );
           external;
```

Figure 134. TcpWaitSend Example

Parameter **Description**

Connection The connection number, as returned by TcpOpen or TcpWaitOpen in the Connection field of the StatusInfoType record.

Buffer The address of the buffer containing the data to send.

BufferLength The size of the buffer.

PushFlag Is set to force the data, and previously queued data, to be sent immediately to the foreign application.

UrgentFlag Is set to mark the data as *urgent*. The semantics of urgent data depends on your application.

Note: Use urgent data with caution. If the foreign application follows the Telnet-style use of urgent data, it might flush all urgent data, until a special character sequence is encountered.

ReturnCode Indicates success or failure of call:
- OK
- ABNORMALcondition
- BADlengthARGUMENT
- CANNOTsendDATA
- FATALerror
- NObufferSPACE (TcpSend and TcpFSend)
- NOsuchCONNECTION
- NOTyetBEGUN
- NOTyetOPEN
- TCPipSHUTDOWN
- INVALIDvirtualADDRESS
- SOFTWAREerror
- REMOTEreset

For a description of Pascal return codes, see Table 19 on page 494.

Notes:

1. A successful TcpFSend, TcpSend, and TcpWaitSend means that TCP has received the data to be sent and stored it in its internal buffers. TCP then puts the data in packets and transmits it when the conditions permit.
2. Data sent in a TcpFSend, TcpSend, or TcpWaitSend request can be split into numerous packets by TCP, or the data can wait in TCP’s buffer space and share a packet with other TcpFSend, TcpSend, or TcpWaitSend, requests.
3. The PushFlag is used to expedite when TCP sends the data.

Setting the PushFlag to `FALSE` allows TCP to buffer the data and wait until it has enough data to transmit so as to use the transmission line more efficiently. There can be some delay before the foreign host receives the data.

Setting the PushFlag to `TRUE` instructs TCP to put data into packets and transmit any buffered data from previous Send requests along with the data in the current TcpFSend, TcpSend, or TcpWaitSend request without delay or consideration of transmission line efficiency. A successful send does not imply that the foreign application has actually received the data, only that the data will be sent as soon as possible.

4. TcpWaitSend is intended for programs that manage a single TCP connection. It is not suitable for use by multiple connection servers.

TcpNameChange

Use the procedure shown in Figure 135 if the address space running the TCP/IP program is not named TCP/IP and is not the same as specified in the TcpipUserid statement of the *hlq*.TCPIP.DATA data set. (See the *OS/390 IBM Communications Server: IP Configuration Reference* .)

If required, this procedure must be called before the BeginTcpIp procedure.

```
procedure  TcpNameChange
           (
                NewNameOfTcp:  DirectoryNameType
           );
           external;
```

Figure 135. TcpNameChange Example

Parameter **Description**

NewNameOfTcp
The name of the address space running TCP/IP.

TcpOpen and TcpWaitOpen

The examples in this section illustrate TcpOpen and TcpWaitOpen.

Use TcpOpen or TcpWaitOpen to initiate a TCP connection. TcpOpen returns immediately, and connection establishment proceeds asynchronously with your program’s other operations. The connection is fully established when your program receives a CONNECTIONstateCHANGED notification with NewState set to OPEN. TcpWaitOpen does not return until the connection is established, or until an error occurs.

There are two types of TcpOpen calls: passive open and active open. A passive open call sets the connection state to LISTENING. An active open call sets the connection state to TRYINGtoOPEN.

If a TcpOpen or TcpWaitOpen call returns ZEROresources, and your application handles RESOURCESavailable notifications, you receive a RESOURCESavailable notification when sufficient resources are available to process an open call. The first open your program issues after a RESOURCESavailable notification is guaranteed not to get the ZEROresources return code.

```
procedure TcpOpen
          (
     var     ConnectionInfo:  StatusInfoType;
     var     ReturnCode:  integer
          );
          external;
```

Figure 136. TcpOpen Example

```
procedure TcpWaitOpen
          (
     var     ConnectionInfo:  StatusInfoType;
     var     ReturnCode:  integer
          );
          external;
```

Figure 137. TcpWaitOpen Example

Parameter **Description**

ConnectionInfo
Is a connection information record.

Connection Set this field to UNSPECIFIEDconnection. When the call returns, the field contains the number of the new connection if ReturnCode is OK.

ConnectionState
For active open, set this field to TRYINGtoOPEN. For passive open, set this field to LISTENING.

OpenAttemptTimeout
Set this field to specify how long, in seconds, TCP is to continue attempting to open the connection. If the connection is not fully established during that time, TCP reports the error to you. If you used TcpOpen, you receive a notification. The type of notification that you receive is CONNECTIONstateCHANGED. It has a new state of NONEXISTENT and a reason of TIMEOUTopen. If you used TcpWaitOpen, it returns with ReturnCode set to TIMEOUTopen.

Security This field is reserved. Set it to DEFAULTsecurity.

Compartment This field is reserved. Set it to DEFAULTcompartment.

Precedence This field is reserved. Set it to DEFAULTprecedence.

LocalSocket **Active Open:** You can use an address of UNSPECIFIEDaddress (TCP/IP uses the home address corresponding to the network interface used to route to the foreign address) and a port of UNSPECIFIEDport (TCP/IP assigns a port number, in the range of 1000 to 65534). You can specify the address, the port, or both if particular values are required by your application. The address must be a valid home address for your node, and the port must be available (not reserved, and not in use by another application).

Passive Open: You usually specify a predetermined port number, known by another program, which can do an active open to connect to your program. Alternatively, you can use UNSPECIFIEDport to let TCP/IP assign a port number, obtain the port number through TcpStatus, and transmit it to the other program through an existing TCP connection or manually. You generally specify an address of UNSPECIFIEDaddress, so that the active open to your port succeeds, regardless of the home address to which it was sent.

ForeignSocket

Active Open: The address and port must both be specified, because TCP/IP cannot actively initiate a connection without knowing the destination address and port.

Passive Open: If your program is offering a service to anyone who wants it, specify an address of UNSPECIFIEDaddress and a port of UNSPECIFIEDport. You can specify a particular address and port if you want to accept an active open only from a certain foreign application.

ReturnCode Indicates success or failure of call. Possible return values are:

- OK
- ABNORMALcondition
- FATALerror
- LOCALportNOTavailable
- NOsuchCONNECTION
- NOTyetBEGUN
- OPENrejected (TcpWaitOnly)
- PARAMlocalADDRESS
- PARAMstate
- PARAMtimeout
- PARAMunspecADDRESS
- PARAMunspecPORT
- REMOTEreset (TcpWaitOnly)
- SOFTWAREerror
- TCPipSHUTDOWN
- TIMEOUTopen (TcpWaitOnly)
- TOOmanyOPENS
- UNAUTHORIZEDuser (TcpWaitOnly)
- UNREACHABLEnetwork (TcpWaitOnly)
- WRONGsecORprc
- ZEROresources

For a description of Pascal return codes, see Table 19 on page 494.

TcpOption

Use the procedure shown in Figure 138 to set an option for a TCP connection.

```
procedure  TcpOption
           (
           Connection:  ConnectionType
           OptionName:  integer
           OptionValue:  integer;
     var   ReturnCode:  integer;
           );  external;
```

Figure 138. TcpOption Example

Parameter **Description**

Connection The connection number, as returned by TcpOpen or TcpWaitOpen in the Connection field of the StatusInforType record.

OptionName The code for the option.

Name **Description**

OPTIONtcpKEEPALIVE
If OptionValue is nonzero then the keep-alive mechanism is activated for connection. If OptionValue is zero then the keep-alive mechanism is deactivated for the connection. When activated, the keep-alive mechanism periodically sends a packet on an otherwise idle connection. If the remote TCP does not respond to the packet or to retransmissions of the packet, then the connection state is changed to NONEXISTENT, with reason TIMEOUT connection.

OptionValue The value for the option.

ReturnCode Indicates success or failure of call.

Possible return values are:
- OK
- NOsuchCONNECTION
- NOTyetBEGUN
- TCPipSHUTDOWN
- INVALIDrequest
- SOFTWAREerror

For a description of Pascal return codes, see Table 19 on page 494.

TcpStatus

Use TcpStatus to obtain the current status of a TCP connection. Your program sets the Connection field of the ConnectionInfo record to the number of the connection whose status you want. Figure 139 on page 521 shows an example of TcpStatus.

```
procedure  TcpStatus
           (
      var      ConnectionInfo:  StatusInfoType;
      var      ReturnCode:  integer
           );
           external;
```

Figure 139. TcpStatus Example

Parameter **Description**

ConnectionInfo
If ReturnCode is OK, the following fields are returned.

OpenAttemptTimeout
If the connection is in the process of being opened (including a passive open), this field is set to the number of seconds remaining before the open is terminated if it has not completed. Otherwise, it is set to WAITforever.

BytesToRead
The number of bytes of incoming data queued for your program (waiting for TcpReceive, TcpFReceive, or TcpWaitReceive).

UnackedBytes
The number of bytes sent by your program but not yet sent to the foreign TCP, or the number of bytes sent to the foreign TCP, but not yet acknowledged.

ConnectionState
The current connection state.

LocalSocket
The local socket, consisting of a local address and a local port.

ForeignSocket
The foreign socket, consisting of a foreign address and a foreign port.

ReturnCode
Indicates the success or failure of the call. Possible return values are:
- OK
- NOsuchCONNECTION
- NOTyetBEGUN
- TCPipSHUTDOWN
- REMOTEreset
- SOFTWAREerror

For a description of Pascal return codes, see Table 19 on page 494.

Note: Your program cannot monitor connection state changes exclusively through polling with TcpStatus. It must receive CONNECTIONstateCHANGED notifications through GetNextNote, for the TCP interface to work properly.

UdpClose

The procedure shown in Figure 140 closes the UDP socket specified in the ConnIndex field. All incoming datagrams on this connection are discarded.

```
  procedure  UdpClose
             (
                  ConnIndex:  ConnectionIndexType;
       var        ReturnCode:  CallReturnCodeType
             );
             external;
```

Figure 140. UdpClose Example

Parameter **Description**

ConnIndex The ConnIndex value returned from UdpOpen.

ReturnCode Indicates success or failure of a call. Possible return values are:

- OK
- NOsuchCONNECTION
- NOTyetBEGUN
- TCPipSHUTDOWN
- SOFTWAREerror

For a description of Pascal return codes, see Table 19 on page 494.

UdpNReceive

The procedure shown in Figure 141 notifies the TCP/IP address space that you are willing to receive UDP datagram data. This call returns immediately. The data buffer is not valid until you receive a UDPdatagramDELIVERED notification.

```
  procedure  UdpNReceive
             (
                  ConnIndex:  ConnectionIndexType;
                  BufferAddress:  integer;
                  BufferLength:  integer;
       var        ReturnCode:  CallReturnCodeType
             );
             external;
```

Figure 141. UdpNReceive Example

Parameter **Description**

ConnIndex The ConnIndex value returned from UdpOpen.

BufferAddress
The address of your buffer that is filled with a UDP datagram.

BufferLength The length of your buffer. If you specify a length larger than 65507 bytes, only the first 65507 bytes are used.

ReturnCode Indicates success or failure of a call. Possible return values are:

- OK
- ABNORMALcondition
- FATALerror

- NOsuchCONNECTION
- NOTyetBEGUN
- RECEIVEstillPENDING
- TCPipSHUTDOWN
- SOFTWAREerror
- BADlengthARGUMENT
- INVALIDvirtualADDRESS

For a description of Pascal return codes, see Table 19 on page 494.

UdpOpen

This procedure requests acceptance of UDP datagrams on the specified socket and allows datagrams to be sent from the specified socket. When the socket port is unspecified, UDP selects a port and returns it to the socket port field. When the socket address is unspecified, UDP uses the default local address. If specified, the address must be a valid home address for your node.

Note: When the local address is specified, only the UDP datagrams addressed to it are delivered.

If the ReturnCode indicates the open was successful, use the returned ConnIndex value on any further actions pertaining to this UDP socket. Figure 142 shows an example.

```
procedure  UdpOpen
           (
     var   LocalSocket:  SocketType;
     var   ConnIndex:  ConnectionIndexType;
     var   ReturnCode:  CallReturnCodeType
           );
           external;
```

Figure 142. UdpOpen Example

Parameter **Description**

LocalSocket The local socket (address and port pair).

ConnIndex The ConnIndex value returned from UdpOpen.

ReturnCode Indicates success or failure of a call. Possible return values are:

- OK
- ABNORMALcondition
- FATALerror
- LOCALportNOTavailable
- NOTyetBEGUN
- SOFTWAREerror
- TCPipSHUTDOWN
- UDPlocalADDRESS
- UDPzeroRESOURCES
- TOOmanyOPENS
- UNAUTHORIZEDuser

For a description of Pascal return codes, see Table 19 on page 494.

Note: If a UdpOpen call returns UDPzeroRESOURCES, and your application handles UDPresourcesAVAILABLE notifications, you receive a UDPresourcesAVAILABLE notification when sufficient resources are available to process a UdpOpen call. The first UdpOpen your program issues after a UDPresourcesAVAILABLE notification is guaranteed not to get the UDPzeroRESOURCES return code.

UdpReceive

The procedure shown in Figure 143 notifies the TCP/IP address space that you are willing to receive UDP datagram data.

UdpReceive is for compatibility with old programs only. New programs should use the UdpNReceive procedure, which allows you to specify the size of your buffer.

If you use UdpReceive, TCP/IP can put a datagram as many as 2012 bytes in your buffer. If a larger datagram is sent to your port when UdpReceive is pending, the datagram is discarded without notification.

Note: No data is transferred from the TCP/IP address space in this call. It only tells TCP/IP that you are waiting for a datagram. Data has been transferred when a UDPdatagramDELIVERED notification is received.

```
procedure UdpReceive
          (
                ConnIndex:  ConnectionIndexType;
                DatagramAddress:  integer;
      var    ReturnCode:  CallReturnCodeType
          );
          external;
```

Figure 143. UdpReceive Example

Parameter	Description
ConnIndex	The ConnIndex value returned from UdpOpen.
DatagramAddress	The address of your buffer that is filled with a UDP datagram.
ReturnCode	Indicates success or failure of a call: • OK • ABNORMALcondition • FATALerror • NOsuchCONNECTION • NOTyetBEGUN • SOFTWAREerror • TCPipSHUTDOWN • INVALIDvirtualADDRESS

For a description of Pascal return codes, see Table 19 on page 494.

UdpSend

The procedure shown in Figure 144 on page 525 sends a UDP datagram to the specified foreign socket. The source socket is the local socket selected in the UdpOpen that returned the ConnIndex value that was used. The buffer does not include the UDP header. This header is supplied by TCP/IP.

When there is no buffer space to process the data, an error is returned. In this case, wait for a subsequent UDPdatagramSPACEavailable notification.

```
procedure UdpSend
          (
               ConnIndex:  ConnectionIndexType;
               ForeignSocket:  SocketType;
               BufferAddress:  integer;
               Length:  integer;
      var        ReturnCode:  CallReturnCodeType
          );
          external;
```

Figure 144. UdpSend Example

Parameter **Description**

ConnIndex The ConnIndex value returned from UdpOpen.

ForeignSocket
The foreign socket (address and port) to which the datagram is to be sent.

BufferAddress
The address of your buffer containing the UDP datagram to be sent, excluding UDP header.

Length The length of the datagram to be sent, excluding UDP header. Maximum is 65507 bytes.

ReturnCode Indicates success or failure of a call. Possible return values are:
- OK
- BADlengthARGUMENT
- NObufferSPACE
- NOsuchCONNECTION
- NOTyetBEGUN
- SOFTWAREerror
- TCPipSHUTDOWN
- UDPunspecADDRESS
- UDPunspecPORT
- INVALIDvirtualADDRESS

For a description of Pascal return codes, see Table 19 on page 494.

Unhandle

Use the procedure shown in Figure 145 on page 526 when you no longer want to receive notifications in the given set.

If you request to unhandle the DATAdelivered notification, the Unhandle procedure returns with a code of INVALIDrequest.

```
    procedure  Unhandle
               (
                   Notifications:  NotificationSetType;
          var    ReturnCode:  integer
               );
               external;
```

Figure 145. Unhandle Example

Parameter	Description
Notifications	The set of notifications that you no longer want to receive.
ReturnCode	Indicates success or failure of call. Possible return values are: • OK • NOTyetBEGUN • INVALIDrequest

For a description of Pascal return codes, see Table 19 on page 494.

Sample Pascal Program

This section contains an example of a Pascal application program. The source code can be found in the *hlq*.SEZAINST data set.

Building the Sample Pascal API Module

The following steps describe how to build the Pascal API module:

1. Compile the sample Pascal program.
2. Link-edit the object code module to form an executable module sample.

Running the Sample Module

The following steps describe how to run the sample module:

1. Run the Pascal API sample program with the Receive option (shown in Figure 146).

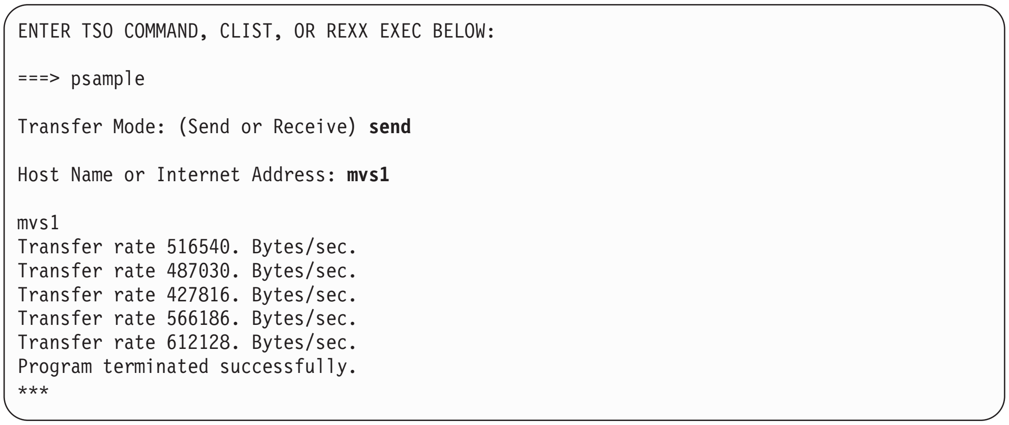
   Run PSAMPLE to start the sample program on the TSO command line. The following example is a typical response:

```
ENTER TSO COMMAND, CLIST, OR REXX EXEC BELOW:

===> psample

Transfer Mode: (Send or Receive) receive

Host Name or Internet Address: mvs1

mvs1
Transfer rate 483884. Bytes/sec.
Transfer rate 442064. Bytes/sec.
Transfer rate 478802. Bytes/sec.
Transfer rate 549568. Bytes/sec.
Transfer rate 635116. Bytes/sec.
Program terminated successfully.
***
```

Figure 146. Sample Pascal API with Receive Option

2. Run the Pascal API sample program with the Send option on a second TSO ID (shown in Figure 147).

   Run PSAMPLE on the TSO command line to start the sample program. The following example is a typical response:



```
ENTER TSO COMMAND, CLIST, OR REXX EXEC BELOW:

===> psample

Transfer Mode: (Send or Receive) send

Host Name or Internet Address: mvs1

mvs1
Transfer rate 516540. Bytes/sec.
Transfer rate 487030. Bytes/sec.
Transfer rate 427816. Bytes/sec.
Transfer rate 566186. Bytes/sec.
Transfer rate 612128. Bytes/sec.
Program terminated successfully.
***
```

Figure 147. Sample Pascal API with Send Option

Sample Pascal Application Program

The following is an example of a Pascal application program.

```
%UHEADER 5647-A01 (C) IBM CORP 1991, 1997.          SYSPARM EZABB01S PSAMPLE
{
TCP/IP for MVS                                    00030000
   SMP/E Distribution Name: EZABB01V (for PSAMPLE source in SEZAINST)   00040000
                            EZABB01S (for PSAMPLE module in SEZAMOD1)   00050000
                                                                        00060000
      Licensed Materials - Property of IBM                              00070000
      This product contains "Restricted Materials of IBM"               00080000
      5647-A01 (C) Copyright IBM Corp. 1991, 1997                       00090000
      All rights reserved.                                              00100000
      US Government Users Restricted Rights -                           00110000
      Use, duplication or disclosure restricted by GSA ADP Schedule     00120000
      Contract with IBM Corp.                                           00130000
      See IBM Copyright Instructions.                                   00140000
                                                                        00215000
}                                                                       00220000
{**********************************************************************}00230000
{*                                                                    *}00240000
{* Memory-to-memory Data Transfer Rate Measurement                    *}00250000
{*                                                                    *}00260000
{* Pseudocode:  Establish access to TCP/IP Services                   *}00270000
{*             Prompt user for operation parameters                   *}00280000
{*             Open a connection (Sender:active, Receiver:passive)    *}00294990
{*             If Sender:                                             *}00300000
{*               Send 5M of data using TcpFSend                       *}00310000
{*               Use GetNextNote to know when Send is complete        *}00320000
{*               Print transfer rate after every 1M of data           *}00330000
{*             else Receiver:                                         *}00340000
{*               Receive 5M of data using TcpFReceive                 *}00350000
{*               Use GetNextNote to know when data is delivered       *}00360000
{*               Print transfer rate after every 1M of data           *}00370000
{*             Close connection                                       *}00380000
{*             Use GetNextNote to wait until connection is closed     *}00390000
{*                                                                    *}00400000
{**********************************************************************}00410000
program PSAMPLE;                                                        00420000
                                                                        00430000
```

```
%include CMALLCL
%include CMINTER
%include CMRESGLB

const
   BUFFERlength = 8192;             { buffer size                    }
   PORTnumber   = 999;              { constant on both sides         }
   CLOCKunitsPERthousandth = '3E8000'x;

static
   Buffer         : packed array (.1..BUFFERlength.) of char;
   BufferAddress  : Address31Type;
   ConnectionInfo : StatusInfoType;
   Count          : integer;
   DataRate       : real;
   Difference     : TimeStampType;
   HostAddress    : InternetAddressType;
   IbmSeconds     : integer;
   Ignored        : integer;
   Line           : string(80);
   Note           : NotificationInfoType;
   PushFlag       : boolean;        { for TcpFSend                   }
   RealRate       : real;
   ReturnCode     : integer;
   SendFlag       : boolean;        { are we sending or receiving    }
   StartingTime   : TimeStampType;
   Thousandths    : integer;
   TotalBytes     : integer;
   UrgentFlag     : boolean;        { for TcpFSend                   }

 var RoundRealRate : integer;

   {******************************************************************}
   {* Print message, release resources and reset environment        *}
   {******************************************************************}
   procedure Restore ( const Message: string;
                       const ReturnCode: integer );
   %UHEADER
   begin
      Write(Message);
      if ReturnCode <> OK then
    {*   Write(SayCalRe(ReturnCode));
      Writeln('');                   *}
         Msg1(Output,1, addr(SayCalRe(ReturnCode)) )
      else Msg0(Output,2);

      EndTcpIp;
      Close (Input);
      Close (Output);
   end;


begin
   TermOut (Output);
   TermIn (Input);

   { Establish access to TCP/IP services }
   BeginTcpIp (ReturnCode);
   if ReturnCode <> OK then begin
   {* Writeln('BeginTcpip: ',SayCalRe(ReturnCode)); *}
      Msg1(Output,4, addr(SayCalRe(ReturnCode)) );
      return;
   end;

   { Inform TCPIP which notifications will be handled by the program }
   Handle ((.DATAdelivered, BUFFERspaceAVAILABLE,
             CONNECTIONstateCHANGED,
             FSendResponse.), ReturnCode);
   if ReturnCode <> OK then begin
      Restore ('Handle: ', ReturnCode);
      return;
   end;

   { Prompt user for operation parameters                            }
{* Writeln('Transfer mode: (Send or Receive)'); *}
   Msg0(Output,5);
   ReadLn (Line);
   if (Substr(Ltrim(Line),1,1) = 's')
   or (Substr(Ltrim(Line),1,1) = 'S') then
      SendFlag := TRUE
   else
      SendFlag := FALSE;

{* Writeln('Host Name or Internet Address :'); *}
   Msg0(Output,6);
   ReadLn (Line);
   GetHostResol (Trim(Ltrim(Line)), HostAddress);
   if HostAddress = NOhost then begin
      Restore ('GetHostResol failed. ', OK);
      return;
   end;

   { Open a TCP connection: active for Send and passive for Receive }
   {   - Connection value will be returned by TcpIp                 }
   {   - initialize IBM reserved fields:  Security, Compartment     }
   {     and Precedence                                             }
   { for Active open  - set Connection State to TRYINGtoOPEN        }
   {                  - must initialize foreign socket              }
   { for Passive open - set ConnectionState to LISTENING            }
   {                  - may leave foreign socket uninitialized to   }
   {                    accept any open attempt                     }
   with ConnectionInfo do begin
      Connection         := UNSPECIFIEDconnection;
      OpenAttemptTimeout := WAITforever;
      Security           := DEFAULTsecurity;
      Compartment        := DEFAULTcompartment;
      Precedence         := DEFAULTprecedence;
      if SendFlag then begin
         ConnectionState       := TRYINGtoOPEN;
         LocalSocket.Address   := UNSPECIFIEDaddress;
         LocalSocket.Port      := UNSPECIFIEDport;
         ForeignSocket.Address := HostAddress;
         ForeignSocket.Port    := PORTnumber;
      end
      else begin
         ConnectionState       := LISTENING;
         LocalSocket.Address   := HostAddress;
         LocalSocket.Port      := PORTnumber;
         ForeignSocket.Address := UNSPECIFIEDaddress;
         ForeignSocket.Port    := UNSPECIFIEDport;
      end;
   end;
   TcpWaitOpen (ConnectionInfo, ReturnCode);
   if ReturnCode <> OK then begin
      Restore ('TcpWaitOpen: ', ReturnCode);
      return;
   end;

   { Initialization }
   BufferAddress := Addr(Buffer(.1.));
   StartingTime  := ClockTime;
   TotalBytes    := 0;
   Count         := 0;
   PushFlag      := false;     { let TcpIp buffer data for efficiency }
```

```
UrgentFlag    := false;               { none of out data is Urgent }

{ Issue first TcpFSend or TcpFReceive }
if SendFlag then
   TcpFSend (ConnectionInfo.Connection, BufferAddress,
                BUFFERlength, PushFlag, UrgentFlag, ReturnCode)
else
   TcpFReceive (ConnectionInfo.Connection, BufferAddress,
                   BUFFERlength, ReturnCode);

if ReturnCode <> OK then begin
{* Writeln('TcpSend/Receive: ',SayCalRe(ReturnCode));  *}
   Msg1(Output,7, addr(SayCalRe(ReturnCode)) );
   return;
end;


{ Repeat until 5M bytes of data have been transferred }
while (Count < 5) do begin
   { Wait until previous transfer operation is completed }
   GetNextNote(Note, True, ReturnCode);
   if ReturnCode <> OK then begin
      restore('GetNextNote :',ReturnCode);
      return;
   end;

   { Proceed with transfer according to the Notification received   }
   { Notifications Expected :                                        }
   {  DATAdelivered - TcpFReceive function call is now complete     }
   {                - receive buffer contains data                   }
   {  FSENDresponse - TcpFSend function call is now complete        }
   {                - send buffer is now available for use           }
   case Note.NotificationTag of
      DATAdelivered:
         begin
            TotalBytes := TotalBytes + Note.BytesDelivered;
            {issue next TcpFReceive                                 }
            TcpFReceive (ConnectionInfo.Connection, BufferAddress,
                   BUFFERlength, ReturnCode);
            if ReturnCode <> OK then begin
               Restore('TcpFReceive: ', ReturnCode);
               return;
            end;
         end;
      FSENDresponse:
         begin
            if Note.SendTurnCode <> OK then begin
               Restore('FSENDresponse: ',Note.SendTurnCode);
               return;
            end
            else begin
               {issue next TcpFSend                         }
               TotalBytes := TotalBytes + BUFFERlength;
               TcpFSend (ConnectionInfo.Connection, BufferAddress,
                   BUFFERlength, PushFlag, UrgentFlag, ReturnCode);
               if ReturnCode <> OK then begin
                  Restore('TcpFSend: ', ReturnCode);
                  return;
               end;
            end;
         end;
      OTHERWISE
         begin
            Restore('UnExpected Notification ',OK);
            return;
         end;
   end; { Case on Note.NotificationTag }
```

```


      { is it time to print transfer rate? }
      if TotalBytes < 1048576 then
         continue;

      { Print transfer rate after every 1M bytes of data transferred }
      DoubleSubtract (ClockTime, StartingTime, Difference);
      DoubleDivide (Difference, CLOCKunitsPERthousandth, Thousandths,
                    Ignored);
      RealRate := (TotalBytes/Thousandths) * 1000.0;
   {* Writeln('Transfer Rate ', RealRate:1:0,' Bytes/sec.'); *}
      RoundRealRate := Round(RealRate);
      Msg1(Output,8, addr(RoundRealRate) );

      StartingTime := ClockTime;
      TotalBytes   := 0;
      Count        := Count + 1;
   end; {Loop while Count < 5 }

   { Close TCP connection and wait till partner also drops connection }
   TcpClose (ConnectionInfo.Connection, ReturnCode);
   if ReturnCode <> OK then begin
      Restore ('TcpClose: ', ReturnCode);
      return;
   end;

   { when partner also drops connection, program will receive         }
   {  CONNECTIONstateCHANGED notification with NewState = NONEXISTENT }
   repeat
      GetNextNote (Note, True, ReturnCode);
      if ReturnCode <> OK then begin
         Restore ('GetNextNote: ', ReturnCode);
         return;
      end;
   until (Note.NotificationTag = CONNECTIONstateCHANGED) &
         ((Note.NewState = NONEXISTENT) ]
            (Note.NewState = CONNECTIONclosing));

   Restore ('Program terminated successfully. ', OK);
end.
```

Part 4. Appendixes

Appendix A. Multitasking C Socket Sample Program

The first sample program is the server in the C language. It allocates a socket, binds to a port, calls listen() to perform a passive open, and uses select() to block until a client request arrives. When a client requests a connection, select() returns and accept() is called to establish the connection.

Note: Some hosts have more than one network address. By specifying a particular network address for the bind() call, a server specifies that it wants to honor connections from one particular network address only. If the server specifies the constant INADDR_ANY for this address, it accepts connections from any of the machine's network addresses.

This program uses the Multitasking Facility (MTF). The server has started a number of subtasks with the MTF task initialization service tinit(). When the server has accepted a connection, it calls tsched() to start the subtask which will handle the client. The server then uses givesocket() and takesocket() to pass the connection to the subtask. When the connection has been passed to the subtask, the main loop blocks in select() waiting for another client.

The second program is the subtask in C. When it begins, it does a takesocket(). It was passed two 8-byte names that define the parent task from which it will obtain the socket. After it gets the socket, it sends a message to this new client and then waits for the client to send a message back.

The third program is the client in C. It allocates a socket, binds to a port, and connects to a server port that is passed as the second parameter port number 691. Then it has a conversation with the server (actually the server's subtask) sending and receiving messages alternatively.

Notes:

1. When you compile the C sample programs, use DEF(MVS) in the CPARM list.
2. When you run the server program, specify PARM='9999' to use port 9999.
3. When you run the client program, specify PARM='MVSF 9999' to use port 9999. Replace MVSF with the host name of your MVS system.

The C examples are followed by an example of an assembler language program that processes the same functions as the C client sample program.

Server Sample Program in C

The following C socket server program is the MTCSRVR member in the *hlq*.SEZAINST data set.

```
/*** IBMCOPYR ********************************************************/
/*                                                                   */
/* Component Name: MTCSRVR (alias EZAEC047)                          */
/*                                                                   */
/* Copyright:                                                        */
/*   Licensed Materials - Property of IBM                            */
/*   This product contains "Restricted Materials of IBM"             */
/*   5655-HAL (C) Copyright IBM Corp. 1994, 1996.                    */
/*   All rights reserved.                                            */
/*   US Government Users Restricted Rights -                         */
/*   Use, duplication or disclosure restricted by GSA ADP Schedule   */
/*   Contract with IBM Corp.                                         */
/*   See IBM Copyright Instructions.                                 */
/*                                                                    */
/*  TCP/IP for MVS                                                    */
/*  SMP/E Distribution Name: EZAEC049                                 */
/*                                                                    */
/*                                                                    */
/*** IBMCOPYR *********************************************************/
/**********************************************************************/
/* C socket Server Program                                            */
/*                                                                    */
/* This code performs the server functions for multitasking, which    */
/* include                                                            */
/*       . creating subtasks                                          */
/*       . socket(), bind(), listen(), accept()                       */
/*       . getclientid                                                */
/*       . givesocket() to TCP/IP in preparation for the subtask      */
/*                      to do a takesocket()                          */
/*       . select()                                                   */
/*                                                                    */
/* There are three test tasks running:                                */
/*       . server master                                              */
/*       . server subtask - separate TCB within server address space  */
/*       . client                                                     */
/*                                                                    */
/**********************************************************************/
static char ibmcopyr[] =
   "MTCSRVR - Licensed Materials - Property of IBM. "
   "This module is \"Restricted Materials of IBM\" "
   "5655-HAL (C) Copyright IBM Corp. 1994, 1996. "
   "See IBM Copyright Instructions.";
#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>
#include <netdb.h>
#include <socket.h>
#include <inet.h>
#include <fcntl.h>
#include <errno.h>
#include <tcperrno.h>
#include <bsdtime.h>
#include <mtf.h>
#include <stdio.h>
int  dotinit(void);
void getsock(int *s);
int  dobind(int *s, unsigned short port);
int  dolisten(int *s);
int  getname(char *myname, char *mysname);
int  doaccept(int *s);
int  testgive(int *s);
int  dogive(int *clsocket, char *myname);
/*
 * Server Main.
 */
main(argc, argv)
int argc;
char **argv;
{
    unsigned short port;       /* port server for bind                */
    int s;                     /* socket for accepting connections    */
    int rc;                    /* return code                         */
    int count;                 /* counter for number of sockets       */
    int clsocket;              /* client socket                       */
    char myname[8];            /* 8 char name of this addres space    */
    char mysname[8];           /* my subtask name                     */
    /*
     * Check arguments. Should be only one: the port number to bind to.
     */
    if (argc != 2) {
```

```
    fprintf(stderr, "Usage: %s port\n", argv[0]);
    exit(1);
}
/*
 * First argument should be the port.
 */
port = (unsigned short) atoi(argv[1]);
fprintf(stdout, "Server: port = %d \n", port);
/*
 * Create subtasks
 */
rc = dotinit();
if (rc < 0)
    perror("Srvr: error for tinit");
printf("rc from tinit is %d\n", rc);
getsock(&s);
printf("Srvr: socket = %d\n", s);
rc = dobind(&s, port);
if (rc < 0)
    tcperror("Srvr: error for bind");
printf("Srvr: rc from bind is %d\n", rc);
rc = dolisten(&s);
if (rc < 0)
    tcperror("Srvr: error for listen");
printf("Srvr: rc from listen is %d\n", rc);
/***************************************
 * To do nonblocking mode,
 *  uncomment out this code.
 *
rc = fcntl(s, F_SETFL, FNDELAY);
if (rc != 0)
    tcperror("Error for fcntl");
printf("rc from fcntl is %d\n", rc);
***************************************/
rc = getname(myname, mysname);
if (rc < 0)
    tcperror("Srvr: error for getclientid");
printf("Srvr: rc from getclientid is %d\n", rc);
/*----------------------------------------------------------------*/
/* . issue accept(), waiting for client connection                */
/* . issue givesocket() to pass client's socket to TCP/IP         */
/* . issue select(), waiting for subtask to complete takesocket() */
/* . close our local socket associated with client's socket       */
/* . loop on accept(), waiting for another client connection      */
/*----------------------------------------------------------------*/
rc    = 0;
count = 0;          /* number of sockets */
while (rc == 0) {
    clsocket = doaccept(&s);
    printf("Srvr: clsocket from accept is %d\n", clsocket);
    count = count + 1;
    printf("Srvr: ###number of sockets is %d\n", count);
    if (clsocket != 0) {
        rc = dogive(&clsocket, myname);
        if (rc < 0)
            tcperror("Srvr: error for dogive");
        printf("Srvr: rc from dogive is %d\n", rc);
        if (rc == 0) {
            rc = tsched(MTF_ANY,"csub", &clsocket,
                            myname, mysname);
            if (rc < 0)
                perror("error for tsched");
            printf("Srvr: rc from tsched is %d\n", rc);
            rc = testgive(&clsocket);
            printf("Srvr: rc from testgive is %d\n", rc);
            sleep(60); /*** do simplified situation first ***/
            printf("Srvr: closing client socket %d\n", clsocket);
```

```
                rc = close(clsocket);   /* give back this socket */
                if (rc < 0)
                    tcperror("error for close of clsocket");
                printf("Srvr: rc from close of clsocket is %d\n", rc);
                /****************************************************/
                exit(0); /*** do  this simplified situation first ***/
                /****************************************************/
            } /** end of if (rc == 0)       ****/
        }   /**** end of if (clsocket != 0) ****/
    }   /******** end of while (rc == 0)    ****/
}   /************ end of main           ********/
/*--------------------------------------------------------------------*/
/*    dotinit()                                                       */
/*    Call tinit() to ATTACH subtask and fetch() subtask load module  */
/*--------------------------------------------------------------------*/
int dotinit(void)
{
    int rc;
    int numsubs = 1;
    printf("Srvr: calling __tinit\n");
    rc = __tinit("mtccsub", numsubs);
    return rc;
}
/*--------------------------------------------------------------------*/
/*    getsock()                                                       */
/*    Get a socket                                                    */
/*--------------------------------------------------------------------*/
void getsock(int *s)
{
    int temp;
    temp = socket(AF_INET, SOCK_STREAM, 0);
    *s = temp;
    return;
}
/*--------------------------------------------------------------------*/
/*    dobind()                                                        */
/*    Bind to all interfaces                                          */
/*--------------------------------------------------------------------*/
int dobind(int *s, unsigned short port)
{
    int rc;
    int temps;
    struct sockaddr_in tsock;
    memset(&tsock, 0, sizeof(tsock));   /* clear tsock to 0's */
    tsock.sin_family      = AF_INET;
    tsock.sin_addr.s_addr = INADDR_ANY; /* bind to all interfaces */
    tsock.sin_port        = htons(port);
    temps = *s;
    rc = bind(temps, (struct sockaddr *)&tsock, sizeof(tsock));
    return rc;
}
/*--------------------------------------------------------------------*/
/*    dolisten()                                                      */
/*    Listen to prepare for client connections.                       */
/*--------------------------------------------------------------------*/
int dolisten(int *s)
{
    int rc;
    int temps;
    temps = *s;
    rc = listen(temps, 10);     /* backlog of 10 */
    return rc;
}
/*--------------------------------------------------------------------*/
/*    getname()                                                       */
/*    Get the identifiers by which TCP/IP knows this server.          */
/*--------------------------------------------------------------------*/
```

```
int getname(char *myname, char *mysname)
{
    int rc;
    struct clientid cid;
    memset(&cid, 0, sizeof(cid));
    rc = getclientid(AF_INET, &cid);
    memcpy(myname,  cid.name,        8);
    memcpy(mysname, cid.subtaskname, 8);
    return rc;
}
/*--------------------------------------------------------------------*/
/*    doaccept()                                                      */
/*    Select() on this socket, waiting for another client connection. */
/*    If connection is pending, issue accept() to get client's socket */
/*--------------------------------------------------------------------*/
int doaccept(int *s)
{
    int temps;
    int clsocket;
    struct sockaddr clientaddress;
    int addrlen;
    int maxfdpl;
    struct fd_set readmask;
    struct fd_set writmask;
    struct fd_set excpmask;
    int rc;
    struct timeval time;
    temps = *s;
    time.tv_sec  = 1000;
    time.tv_usec = 0;
    maxfdpl = temps + 1;
    FD_ZERO(&readmask);
    FD_ZERO(&writmask);
    FD_ZERO(&excpmask);
    FD_SET(temps, &readmask);
    rc = select(maxfdpl, &readmask, &writmask, &excpmask, &time);
    printf("Srvr: rc from select is %d\n", rc);
    if (rc < 0) {
        tcperror("error from select");
        return rc;
    }
    else if (rc == 0) {  /* time limit expired */
        return rc;
    }
    else {              /* this socket is ready */
        addrlen = sizeof(clientaddress);
        clsocket = accept(temps, &clientaddress, &addrlen);
        return clsocket;
    }
}
/*--------------------------------------------------------------------*/
/*    testgive()                                                      */
/*    Issue select(), checking for an exception condition, which      */
/*    indicates that takesocket() by the subtask was successful.      */
/*--------------------------------------------------------------------*/
int testgive(int *s)
{
    int temps;
    struct sockaddr clientaddress;
    int addrlen;
    int maxfdpl;
    struct fd_set readmask;
    struct fd_set writmask;
    struct fd_set excpmask;
    int rc;
    struct timeval time;
    temps = *s;
    time.tv_sec  = 1000;
    time.tv_usec = 0;
    maxfdpl = temps + 1;
    FD_ZERO(&readmask);
    FD_ZERO(&writmask);
    FD_ZERO(&excpmask);
 /* FD_SET(temps, &readmask); */
 /* FD_SET(temps, &writmask); */
    FD_SET(temps, &excpmask);
    rc = select(maxfdpl, &readmask, &writmask, &excpmask, &time);
    printf("Srvr: rc from select for testgive is %d\n", rc);
    if (rc < 0) {
        tcperror("Srvr: error from testgive");
    }
    else
        rc = 0;
    return rc;
}
/*--------------------------------------------------------------------*/
/*    dogive()                                                        */
/*    Issue givesocket() for giving client's socket to subtask.       */
/*--------------------------------------------------------------------*/
int dogive(int *clsocket, char *myname)
{
    int rc;
    struct clientid cid;
    int temps;
    temps = *clsocket;
    memset(&cid, 0, sizeof(cid));
    cid.domain = AF_INET;
    memcpy(cid.name,       myname,     8);
    memcpy(cid.subtaskname,"        ", 8);
    printf("Srvr: givesocket socket is %d\n", temps);
    printf("Srvr: givesocket name is %s\n", cid.name);
    rc = givesocket(temps, &cid);
    return rc;
}
```

The Subtask Sample Program in C

The following C socket server program is the MTCCSUB member in the *hlq.*SEZAINST data set.

```
/*** IBMCOPYR *********************************************************/
/*                                                                    */
/* Component Name: MTCCSUB                                            */
/*                                                                    */
/* Copyright:                                                         */
/*   Licensed Materials - Property of IBM                             */
/*   This product contains "Restricted Materials of IBM"              */
/*   5655-HAL (C) Copyright IBM Corp. 1994, 1996.                     */
/*   All rights reserved.                                             */
/*   US Government Users Restricted Rights -                          */
/*   Use, duplication or disclosure restricted by GSA ADP Schedule   */
/*   Contract with IBM Corp.                                          */
/*   See IBM Copyright Instructions.                                  */
/*                                                                    */
/*  TCP/IP for MVS                                                    */
/*  SMP/E Distribution Name: EZAEC048                                 */
/*                                                                    */
/*                                                                    */
/*** IBMCOPYR *********************************************************/
/**********************************************************************/
/* C Socket Server Subtask Program                                    */
/*                                                                    */
/* This code is started by the tsched() routine of C/370 MTF.         */
/* Its purpose is to do a takesocket() and then send/recv with the   */
/* client process.                                                    */
/**********************************************************************/
#pragma runopts(noargparse,plist(mvs),noexecops)
static char ibmcopyr[] =
   "MTCCSUB - Licensed Materials - Property of IBM. "
   "This module is \"Restricted Materials of IBM\" "
   "5655-HAL (C) Copyright IBM Corp. 1994, 1996. "
   "See IBM Copyright Instructions.";
#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>
#include <netdb.h>
#include <socket.h>
#include <inet.h>
#include <fcntl.h>
#include <errno.h>
#include <tcperrno.h>
#include <bsdtime.h>
#include <stdio.h>
/*
 * Server subtask
 */
csub(int *clsock,         /* address of socket passed  */
     char *tskname,       /* address of caller's name  */
     char *tsksname)      /* address of caller's sname */
{
    int temps;                 /*                                      */
    int sendbytes;             /* # bytes sent                         */
    int recvbytes;             /* # bytes received                     */
    int clsocket;              /* client socket                        */
    int rc;                    /*                                      */
    char xtskname[8];          /* caller's name                        */
    char xtsksname[8];         /* caller's subtask name                */
    clsocket = *clsock;
    memcpy(&xtskname,  tskname,  8);      /* local copy */
    memcpy(&xtsksname, tsksname, 8);      /* local copy */
    rc = doget(&clsocket, xtskname, xtsksname);
    printf("Csub: returned from doget()\n");
    if (rc < 0)
        tcperror("Csub: Error from doget");
    printf("Csub: rc from doget is %d\n", rc);
    temps = rc;         /* new socket number */
    if (temps > -1) do {
        sendbytes = dosend(&temps);
        recvbytes = dorecv(&temps);
    } while (0);
 /* } while (recvbytes > 0); do simplified situation first ***/
 fflush(stdout);
 sleep(30);
}
/*-----------------*/
/*    doget()      */
/*-----------------*/
int doget(int *clsocket, char *xtskname, char *xtsksname)
{
    int rc;
    int temps;
    struct clientid cid;
    memset(&cid, 0, sizeof(cid));
    temps = *clsocket;
    memcpy(cid.name,        xtskname,  8);
    memcpy(cid.subtaskname, xtsksname, 8);
    cid.domain = AF_INET;
    rc = takesocket(&cid, temps);
    *clsocket = temps;
    return rc;
}
```

```
/*-----------------*/
/*    dosend()     */
/*-----------------*/
int dosend(int *clsocket)
{
    int temps;
    int sendbytes;
    char data[80] = "Message from subtask: I sent this data";
    /*****************************************************
          note: stream mode means that data is not sent
                as a record and can therefore flow in
                variable sized chunks across the network.
                This example is a simplified situation.
    ******************************************************/
    temps = *clsocket;
    sendbytes = send(temps, data, sizeof(data), 0);
    printf("Csub: sendbytes = %d\n", sendbytes);
    return sendbytes;
}
/*-----------------*/
/*    dorecv()     */
/*-----------------*/
int dorecv(int *clsocket)
{
    int temps;
    int recvbytes;
    char data[80];
    char *datap;
    /*****************************************************
          note: stream mode means that data is not sent
                as a record and can therefore flow in
                variable sized chunks across the network.
                This example is a simplified situation.
    ******************************************************/
    temps = *clsocket;
    recvbytes = recv(temps, data, sizeof(data), 0);
    if (recvbytes > 0)
        printf("Csub: data recv: %s\n", data);
    else
        printf("Csub: client stopped sending data\n");
    printf("Csub: recvbytes = %d\n", recvbytes);
    return recvbytes;
}
```

The Client Sample Program in C

The following C socket server program is the MTCCLNT member in the *hlq.*SEZAINST data set.

```
/*** IBMCOPYR ********************************************************/
/*                                                                   */
/* Component Name: MTCCLNT                                           */
/*                                                                   */
/* Copyright:                                                        */
/*   Licensed Materials - Property of IBM                            */
/*   This product contains "Restricted Materials of IBM"             */
/*   5655-HAL (C) Copyright IBM Corp. 1994, 1996.                    */
/*   All rights reserved.                                            */
/*   US Government Users Restricted Rights -                         */
/*   Use, duplication or disclosure restricted by GSA ADP Schedule   */
/*   Contract with IBM Corp.                                         */
/*   See IBM Copyright Instructions.                                 */
/*                                                                   */
/*  TCP/IP for MVS                                                   */
/*  SMP/E Distribution Name: EZAEC047                                */
/*                                                                   */
/*                                                                   */
```

```
/*** IBMCOPYR ********************************************************/
/*********************************************************************/
/* C Socket Client Program                                           */
/*                                                                   */
/* This code sends and receives mgs with the server subtask.         */
/*********************************************************************/
static char ibmcopyr[] =
   "MTCCLNT - Licensed Materials - Property of IBM. "
   "This module is \"Restricted Materials of IBM\" "
   "5655-HAL (C) Copyright IBM Corp. 1994, 1996. "
   "See IBM Copyright Instructions.";
#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>
#include <netdb.h>
#include <socket.h>
#include <inet.h>
#include <errno.h>
#include <tcperrno.h>
#include <bsdtime.h>
#include <stdio.h>
int dosend(int *s);
int dorecv(int *s);
int doconn(int *s, unsigned long *octaddrp, unsigned short port);
void getsock(int *s);
/*
 * Client
 */
main(int argc, char **argv)
{
    int gotbytes;              /* number of bytes received            */
    int sndbytes;              /* number of bytes sent                */
    int s;                     /* socket descriptor                   */
    int rc;                    /* return code                         */
    struct in_addr octaddr;    /* host internet address (binary)      */
    unsigned short port;       /* port number sent as parameter       */
    char * charaddr;           /* host internet address (dotted dec)  */
    struct hostent *hostnm;    /* server host name information        */
    /*
     * Check Arguments Passed. Should be hostname and port.
     */
    if (argc != 3) {
        fprintf(stderr, "Usage: %s hostname port\n", argv[0]);
        exit(1);
    }
    /*
     * The host name is the first argument. Get the server address.
     */
    hostnm = gethostbyname(argv[1]);
    if (hostnm == (struct hostent *) 0) {
        fprintf(stderr, "Gethostbyname failed\n");
        exit(2);
    }
    octaddr.s_addr = *((unsigned long *)hostnm->h_addr);
    /*
     * The port is the second argument.
     */
    port = (unsigned short) atoi(argv[2]);
    fprintf(stdout, "Clnt: port = %d\n", port);
    getsock(&s);
    printf("Clnt: our socket is %d\n", s);
    charaddr = inet_ntoa(octaddr);
    printf("Clnt: address of host is %8s\n", charaddr);
    rc = doconn(&s, &octaddr.s_addr, port);
    if (rc < 0)
        tcperror("Clnt: error for connect");
    else {
        printf("Clnt: rc from connect is %d\n", rc);
        do {
            gotbytes = dorecv(&s);
            sndbytes = dosend(&s);
        } while (0);
     /* } while (sndbytes > 0); do simplified situation first ***/
        sleep(15);
    }
}
/*-----------------*/
/*    getsock()    */
/*-----------------*/
void getsock(int *s)
{
    int temp;
    temp = socket(AF_INET, SOCK_STREAM, 0);
    *s = temp;
    return;
}
/*-----------------*/
/*    doconn()     */
/*-----------------*/
int doconn(int *s, unsigned long *octaddrp, unsigned short port)
{
    int rc;
    int temps;
    struct sockaddr_in tsock;
    memset(&tsock, 0, sizeof(tsock));
    tsock.sin_family      = AF_INET;
    tsock.sin_port        = htons(port);
    tsock.sin_addr.s_addr = *octaddrp;
    temps = *s;
    rc = connect(temps, (struct sockaddr *)&tsock, sizeof(tsock));
    return rc;
}
/*-----------------*/
/*    dorecv()     */
/*-----------------*/
int dorecv(int *s)
{
    int temps;
    int gotbytes;
    char data[100];
    temps = *s;
    gotbytes = recv(temps, data, sizeof(data), 0);
    if (gotbytes < 0) {
        tcperror("Clnt: error for recv");
    }
    else
        printf("Clnt: data recv: %s\n", data);
    return gotbytes;
}
/*-----------------*/
/*    dosend()     */
/*-----------------*/
int dosend(int *s)
{
    int temps;
    int sndbytes;
    char data[50];
    temps = *s;
    gets(data);
    printf("clnt: data to send: %s\n", data);
    sndbytes = send(temps, data, sizeof(data), 0);
    if (sndbytes < 0) {
        tcperror("Clnt: error for send");
    }
    else
        printf("Clnt: sent %d bytes to server subtask\n", sndbytes);
    return sndbytes;
}
```

Appendix B. Return Codes

This appendix contains system error return codes for socket calls, they apply to all socket APIs in this book. It also contains sockets extended return codes that apply only to the macro, call instruction, and REXX socket APIs.

If the return code is not listed in this appendix, it is a return code that is received from OS/390 UNIX. Refer to *OS/390 UNIX System Services Messages and Codes* for the OS/390 UNIX ERRNOs.

See "User Abend U4093" on page 561 for a description of user abend U4093.

System Error Codes for Socket Calls

This section contains the error codes and the message names that refer to the following APIs:

- C sockets
- Macro
- Call instruction
- REXX sockets

The names in the Socket Type column are identifiers that apply to all of the above APIs and do not follow the naming convention for any specific API. These message numbers and codes are in the TCPERRNO.H include file.

When a socket call is processed, both a return code and an error number are returned to your program. If the return code is zero or a positive number, the call completed normally. If the return code is a negative number, the call did not complete normally and an error number is returned. See the following table for the meaning of the error number that is returned.

For the following error conditions, a name is returned by C socket calls and a number is returned by the sockets extended interface calls. The error condition return codes can originate from the socket application programming interface or from a peer server program.

Sockets ERRNOs

Table 20 shows the ERRNOs returned by TCP/IP for MVS.

Table 20. Sockets ERRNOs

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.

Table 20. Sockets ERRNOs (continued)

Error Number	Message Name	Socket Type	Error Description	Programmer's Response
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.
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.

Table 20. Sockets ERRNOs (continued)

Error Number	Message Name	Socket Type	Error Description	Programmer's Response
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 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.

Table 20. Sockets ERRNOs (continued)

Error Number	Message Name	Socket Type	Error Description	Programmer's Response
34	ERANGE	All	The result is too large.	Check function parameter values.
35	EWOULDBLOCK	Accept	The socket is in nonblocking mode and connections are not queued. This is not an error condition.	Reissue Accept().
35	EWOULDBLOCK	Read Recvfrom	The socket is in nonblocking mode and read data is not available. This is not an error condition.	Issue a select on the socket to determine when data is available to be read or reissue the Read()/Recvfrom().
35	EWOULDBLOCK	Send Sendto Write	The socket is in nonblocking mode and buffers are not available.	Issue a select on the socket to determine when data is available to be written or reissue the Send(), Sendto(), or Write().
36	EINPROGRESS	Connect	The socket is marked nonblocking and the connection cannot be completed immediately. This is not an error condition.	See the Connect() description for possible responses.
37	EALREADY	Connect	The socket is marked nonblocking and the previous connection has not been completed.	Reissue Connect().
37	EALREADY	Maxdesc	A socket has already been created calling Maxdesc() or multiple calls to Maxdesc().	Issue Getablesize() to query it.
37	EALREADY	Setibmopt	A connection already exists to a TCP/IP image. A call to 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. It exceeds the IP limit of 64K or the limit set by the setsockopt() call.	Either correct the length parameter, or send the message in smaller pieces.
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.

Table 20. Sockets ERRNOs (continued)

Error Number	Message Name	Socket Type	Error Description	Programmer's Response
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.
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.

Table 20. Sockets ERRNOs (continued)

Error Number	Message Name	Socket Type	Error Description	Programmer’s Response
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.	Call your system administrator.
55	ENOBUFS	Send Sendto Write	Not enough buffer space is available to send the new message.	Call your system administrator.
55	ENOBUFS	Takesocket	Not enough buffer space is available to create the new socket.	Call your system administrator.
56	EISCONN	Connect	The socket is already connected.	Correct the socket descriptor on Connect() or do not issue a Connect() twice for the socket.
57	ENOTCONN	All	The socket is not connected.	Connect the socket before communicating.
58	ESHUTDOWN	All	A Send cannot be processed after socket shutdown.	Issue read/receive before shutting down the read side of the socket.
59	ETOOMANYREFS	All	There are too many references. A splice cannot be completed.	Call your system administrator.
60	ETIMEDOUT	Connect	The connection timed out before it was completed.	Ensure the server application is available.
61	ECONNREFUSED	Connect	The requested connection was refused.	Ensure server application is available and at specified port.
62	ELOOP	All	There are too many symbolic loop levels.	Reduce symbolic links to specified file.
63	ENAMETOOLONG	All	The file name is too long.	Reduce size of specified file name.
64	EHOSTDOWN	All	The host is down.	Restart specified host.

Table 20. Sockets ERRNOs (continued)

Error Number	Message Name	Socket Type	Error Description	Programmer's Response
65	EHOSTUNREACH	All	There is no route to the host.	Set up network path to specified host and verify that host name is valid.
66	ENOTEMPTY	All	The directory is not empty.	Clear out specified directory and reissue call.
67	EPROCLIM	All	There are too many processes in the system.	Decrease the number of processes or increase the process limit.
68	EUSERS	All	There are too many users on the system.	Decrease the number of users or increase the user limit.
69	EDQUOT	All	The disk quota has been exceeded.	Call your system administrator.
70	ESTALE	All	An old NFS** data set handle was found.	Call your system administrator.
71	EREMOTE	All	There are too many levels of remote in the path.	Call your system administrator.
72	ENOSTR	All	The device is not a stream device.	Call your system administrator.
73	ETIME	All	The timer has expired.	Increase timer values or reissue function.
74	ENOSR	All	There are no more stream resources.	Call your system administrator.
75	ENOMSG	All	There is no message of the desired type.	Call your system administrator.
76	EBADMSG	All	The system cannot read the message.	Verify that 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.
79	ENOLCK	All	No record locks are available.	Call your system administrator.

Table 20. Sockets ERRNOs (continued)

Error Number	Message Name	Socket Type	Error Description	Programmer's Response
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 0–49.	Reissue function with corrected socket descriptor.
113	EBADF	Bind socket	The socket descriptor is already being used.	Correct the socket descriptor.
113	EBADF	Givesocket	The socket has already been given. The socket domain is not AF_INET.	Correct the socket descriptor.
113	EBADF	Select	One of the specified descriptor sets is an incorrect socket descriptor.	Correct the socket descriptor. Set on Select() or Selectex().
113	EBADF	Takesocket	The socket has already been taken.	Correct the socket descriptor.
113	EBADF	Accept	A Listen() has not been issued before the Accept().	Issue Listen() before Accept().
121	EINVAL	All	An incorrect argument was specified.	Check and correct all function parameters.
145	E2BIG	All	The argument list is too long.	Eliminate excessive number of arguments.
156	EMVSINITIAL	All	Process initialization error.	Attempt to initialize again.

Table 20. Sockets ERRNOs (continued)

Error Number	Message Name	Socket Type	Error Description	Programmer's Response
1002	EIBMSOCKOUTOFRANGE	Socket	A socket number assigned by the client interface code is out of range.	Check the socket descriptor parameter.
1003	EIBMSOCKINUSE	Socket	A socket number assigned by the client interface code is already in use.	Use a different socket descriptor.
1004	EIBMIUCVERR	All	The request failed because of an IUCV error. This error is generated by the client stub code.	Ensure IUCV/VMCF is functional.
1008	EIBMCONFLICT	All	This request conflicts with a request already queued on the same socket.	Cancel the existing call or wait for its completion before reissuing this call.
1009	EIBMCANCELLED	All	The request was cancelled by the CANCEL call.	Informational, no action needed.
1011	EIBMBADTCPNAME	All	A TCP/IP name that is not valid was detected.	Correct the name specified in the IBM_TCPIMAGE structure.
1011	EIBMBADTCPNAME	Setibmopt	A TCP/IP name that is not valid was detected.	Correct the name specified in the IBM_TCPIMAGE structure.
1011	EIBMBADTCPNAME	INITAPI	A TCP/IP name that is not valid was detected.	Correct the name specified on the IDENT option TCPNAME field.
1012	EIBMBADREQUESTCODE	All	A request code that is not valid was detected.	Contact your system administrator.
1013	EIBMBADCONNECTIONSTATE	All	A connection token that is not valid was detected; bad state.	Verify TCP/IP is active.
1014	EIBMUNAUTHORIZEDCALLER	All	An unauthorized caller specified an authorized keyword.	Ensure user ID has authority for the specified operation.
1015	EIBMBADCONNECTIONMATCH	All	A connection token that is not valid was detected. There is no such connection.	Verify TCP/IP is active.
1016	EIBMTCPABEND	All	An abend occurred when TCP/IP was processing this request.	Verify that TCP/IP has restarted.
1026	EIBMINVDELETE	All	Delete requestor did not create the connection.	Delete the request from the process that created it.
1027	EIBMINVSOCKET	All	A connection token that is not valid was detected. No such socket exists.	Call your system programmer.
1028	EIBMINVTCPCONNECTION	All	Connection terminated by TCP/IP. The token was invalidated by TCP/IP.	Reestablish the connection to TCP/IP.
1032	EIBMCALLINPROGRESS	All	Another call was already in progress.	Reissue after previous call has completed.

Table 20. Sockets ERRNOs (continued)

Error Number	Message Name	Socket Type	Error Description	Programmer's Response
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.
1040	EIBMSELECTEXPOST	SELECTEX	SELECTEX passed an ECB that was already posted.	Check whether the user's ECB was already posted.
2001	EINVALIDRXSOCKETCALL	REXX	A syntax error occurred in the RXSOCKET parameter list.	Correct the parameter list passed to the REXX socket call.
2002	ECONSOLEINTERRUPT	REXX	A console interrupt occurred.	Retry the task.
2003	ESUBTASKINVALID	REXX	The subtask ID is incorrect.	Correct the subtask ID on the INITIALIZE call.
2004	ESUBTASKALREADYACTIVE	REXX	The subtask is already active.	Only issue the INITIALIZE call once in your program.
2005	ESUBTASKALNOTACTIVE	REXX	The subtask is not active.	Issue the INITIALIZE call before any other socket call.
2006	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.

Table 20. Sockets ERRNOs (continued)

Error Number	Message Name	Socket Type	Error Description	Programmer's Response
2016	EHOSTNOTFOUND	REXX	The host is not found.	Call your MVS system programmer.
2017	EIPADDRNOTFOUND	REXX	Address not found.	Call your MVS system programmer.

OS/390 UNIX Return Codes

Table 21 contains the OS/390 UNIX error condition codes that are not translated to a TCP/IP errno.

Table 21. OS/390 UNIX Return Codes

Error condition	Error number
ENOSYS	134
EILSEQ	147
EOVERFLOW	149
EMVSNOTUP	150
EMVSDYNALC	151
EMVSCVAF	152
EMVSCATLG	153
EMVSINITIAL	156
EMVSERR	157
EMVSPARM	158
EMVSPFSFILE	159
EMVSPFSPERM	162
EMVSSAFEXTRERR	163
EMVSSAF2ERR	164
EMVSNORTL	167
EMVSEXPIRE	168
EMVSPASSWORD	169
EMVSWLMERROR	170

For more information about OS/390 UNIX error codes, refer to *OS/390 UNIX System Services Messages and Codes*.

Additional Return Codes

The following section contains the error condition codes that are returned in the ERRNO field by the API when you use the sockets extended interfaces. The RETCODE field contains a −1 when an error condition is returned.

Note: The return codes 10119 through 10130 return the IPUSER variable.

Sockets Extended ERRNOs

Table 22 on page 558 shows the ERRNOs that are returned by the sockets extended interface.

Table 22. Sockets Extended ERRNOs

Error Code	Problem Description	System Action	Programmer's Response
10100	An ESTAE macro did not complete normally.	End the call.	Call your MVS system programmer.
10101	A STORAGE OBTAIN failed.	End the call.	Increase MVS storage in the application's address space.
10108	The first call from TCP/IP was not INITAPI or TAKESOCKET.	End the call.	Change the first TCP/IP call to INITAPI or TAKESOCKET.
10110	LOAD of EZBSOH03 (alias EZASOH03) failed.	End the call.	Call the IBM Software Support Center.
10154	Errors were found in the parameter list for an IOCTL call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the IOCTL call. You might have incorrect sequencing of socket calls.
10155	The length parameter for an IOCTL call is less than or equal to zero.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the IOCTL call. You might have incorrect sequencing of socket calls.
10156	The length parameter for an IOCTL call is 3200 (32 x 100).	Disable the subtask for interrupts. Return an error code to the caller.	Correct the IOCTL call. You might have incorrect sequencing of socket calls.
10159	A zero or negative data length was specified for a READ or READV call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the length in the READ call.
10161	The REQARG parameter in the IOCTL parameter list is zero.	End the call.	Correct the program.
10163	A 0 or negative data length was found for a RECV, RECVFROM, or RECVMSG call.	Disable the subtask for interrupts. Sever the DLC path. Return an error code to the caller.	Correct the data length.
10167	The descriptor set size for a SELECT or SELECTEX call is less than or equal to zero.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the SELECT or SELECTEX call. You might have incorrect sequencing of socket calls.
10168	The descriptor set size *in bytes* for a SELECT or SELECTEX call is greater than 252. A number greater than the maximum number of allowed sockets (2000 is maximum) has been specified.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the descriptor set size.
10170	A zero or negative data length was found for a SEND or SENDMSG call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the data length in the SEND call.
10174	A zero or negative data length was found for a SENDTO call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the data length in the SENDTO call.

Table 22. Sockets Extended ERRNOs (continued)

Error Code	Problem Description	System Action	Programmer's Response
10178	The SETSOCKOPT option length is less than the minimum length.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the OPTLEN parameter.
10179	The SETSOCKOPT option length is greater than the maximum length.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the OPTLEN parameter.
10184	A data length of zero was specified for a WRITE call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the data length in the WRITE call.
10186	A negative data length was specified for a WRITE or WRITEV call.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the data length in the WRITE call.
10190	The GETHOSTNAME option length is less than 24 or greater than the maximum length.	Disable the subtask for interrupts. Return an error code to the caller.	Correct the length parameter.
10193	The GETSOCKOPT option length is less than the minimum or greater than the maximum length.	End the call.	Correct the length parameter.
10197	The application issued an INITAPI call after the connection was already established.	Bypass the call.	Correct the logic that produces the INITAPI call that is not valid.
10198	The maximum number of sockets specified for an INITAPI exceeds 2000.	Return to the user.	Correct the INITAPI call.
10200	The first call issued was not a valid first call.	End the call.	For a list of valid first calls, refer to the section on special considerations in the chapter on general programming .
10202	The RETARG parameter in the IOCTL call is zero.	End the call.	Correct the parameter list. You might have incorrect sequencing of socket calls.
10203	The requested socket number is a negative value.	End the call.	Correct the requested socket number.
10205	The requested socket number is a duplicate.	End the call.	Correct the requested socket number.
10208	The NAMELEN parameter for a GETHOSTBYNAME call was not specified.	End the call.	Correct the NAMELEN parameter. You might have incorrect sequencing of socket calls.
10209	The NAME parameter on a GETHOSTBYNAME call was not specified.	End the call.	Correct the NAME parameter. You might have incorrect sequencing of socket calls.
10210	The HOSTENT parameter on a GETHOSTBYNAME or GETHOSTBYADDR call was not specified.	End the call.	Correct the HOSTENT parameter. You might have incorrect sequencing of socket calls.

Table 22. Sockets Extended ERRNOs (continued)

Error Code	Problem Description	System Action	Programmer's Response
10211	The HOSTADDR parameter on a GETHOSTBYNAME or GETHOSTBYADDR call is incorrect.	End the call.	Correct the HOSTADDR parameter. You might have incorrect sequencing of socket calls.
10212	The resolver program failed to load correctly for a GETHOSTBYNAME or GETHOSTBYADDR call.	End the call.	Check the JOBLIB, STEPLIB, and linklib datasets and rerun the program.
10213	Not enough storage is available to allocate the HOSTENT structure.	End the call.	Increase the user storage allocation for this job.
10214	The HOSTENT structure was not returned by the resolver program.	End the call.	Ensure that the domain name server is available. This can be a nonerror condition indicating that the name or address specified in a GETHOSTBYADDR or GETHOSTBYNAME call could not be matched.
10215	The APITYPE parameter on an INITAPI call instruction was not 2 or 3.	End the call.	Correct the APITYPE parameter.
10218	The application programming interface (API) cannot locate the specified TCP/IP.	End the call.	Ensure that an API that supports the performance improvements related to CPU conservation is installed on the system and verify that a valid TCP/IP name was specified on the INITAPI call. This error call might also mean that EZASOKIN could not be loaded.
10219	The NS parameter is greater than the maximum socket for this connection.	End the call.	Correct the NS parameter on the ACCEPT, SOCKET or TAKESOCKET call.
10221	The AF parameter of a SOCKET call is not AF_INET.	End the call.	Set the AF parameter equal to AF_INET.
10222	The SOCTYPE parameter of a SOCKET call must be stream, datagram, or raw (1, 2, or 3).	End the call.	Correct the SOCTYPE parameter.
10223	No ASYNC parameter specified for INITAPI with APITYPE=3 call.	End the call.	Add the ASYNC parameter to the INITAPI call.
10224	The IOVCNT parameter is less than or equal to zero, for a READV, RECVMSG, SENDMSG, or WRITEV call.	End the call.	Correct the IOVCNT parameter.
10225	The IOVCNT parameter is greater than 120, for a READV, RECVMSG, SENDMSG, or WRITEV call.	End the call.	Correct the IOVCNT parameter.
10226	Invalid COMMAND parameter specified for a GETIBMOPT call.	End the call.	Correct the COMMAND parameter of the GETIBMOPT call.
10229	A call was issued on an APITYPE=3 connection without an ECB or REQAREA parameter.	End the call.	Add an ECB or REQAREA parameter to the call.

Table 22. Sockets Extended ERRNOs (continued)

Error Code	Problem Description	System Action	Programmer's Response
10300	Termination is in progress for either the CICS transaction or the sockets interface.	End the call.	None.
10330	A SELECT call was issued without a MAXSOC value and a TIMEOUT parameter.	End the call.	Correct the call by adding a TIMEOUT parameter.
10331	A call that is not valid was issued while in SRB mode.	End the call.	Get out of SRB mode and reissue the call.
10332	A SELECT call is invoked with a MAXSOC value greater than that which was returned in the INITAPI function (MAXSNO field).	End the call.	Correct the MAXSOC parameter and reissue the call.
10999	An abend has occurred in the subtask.	Write message EZY1282E to the system console. End the subtask and post the TRUE ECB.	If the call is correct, call your system programmer.
20000	An unknown function code was found in the call.	End the call.	Correct the SOC-FUNCTION parameter.
20001	The call passed an incorrect number of parameters	End the call	Correct the parameter list.
20002	The CICS Sockets Interface is not in operation.	End the call	Start the CICS Sockets Interface before executing this call.

User Abend U4093

An abend U4093 indicates that a sockets extended call that is not valid has been detected. It is issued by EZASOKET following a call to EZASOKFN if EZASOKFN has detected an error in the socket call parameter list. The registers at the time of the abend are:

- R2 contains the address of the save area containing the calling program registers.
- R11 contains the error code passed to EZASOKET by EZASOKFN.

  Code Description

  X'4E20' (20000)
  Indicates EZASOKFN could not find the requested CALL function name

  X'4E21' (20001)
  Indicates that EZASOKFN found an incorrect number of parameters in the parameter list for the requested function
- R12 contains the address of the incorrect parameter list.

Figure 148 on page 562 is an example of abend U4093:

```
USER COMPLETION CODE=4093
TIME=15.01.58  SEQ=00074  CPU=0000  ASID=000E
PSW AT TIME OF ERROR  078D1000   80018F14  ILC 2  INTC 0D
ACTIVE LOAD MODULE=DLSV2AS2  ADDRESS=00018670  OFFSET=000008A4
DATA AT PSW  00018F0E - 00181610  0A0D4100  35185000
GPR  0-3  80000000  80000FFD  000189E4  00018DC0
GPR  4-7  00019DC0  00018CE0  00018AA6  00018D18
GPR  8-11 00013780  00019378  00019088  00004E21
GPR 12-15 000187D4  0001902C  80018EF4  00004E21
```

Figure 148. Example of Abend U4093

Appendix C. Address Family Cross Reference

This appendix contains AF_INET and AF_IUCV address family cross reference information for the major APIs in this book.

Address families define different styles of addressing. All hosts in the same addressing family understand and use the same method for addressing socket endpoints. TCP/IP supports two addressing families; AF_INET and AF_IUCV. The AF_INET family defines addressing in the internet domain. The AF_IUCV family defines addressing in the IUCV domain. In the IUCV domain, address spaces can use the socket interface to communicate with other address spaces on the same host.

The INET and IUCV column entries are:

yes The call applies to this address family.

no If you use this call with this address family, an error is returned.

n/a If you use this call with this address family, no error is returned and the call is not processed.

blank The call does not apply to this API.

Table 23. Socket Address Families Cross Reference

Application Programming Interface (API)								
Function	C SOCKETS		MACRO		CALL		REXX	
	INET	IUCV	INET	IUCV	INET	IUCV	INET	IUCV
accept()	yes	yes	yes	no	yes	no	yes	no
bind()	yes	yes	yes	no	yes	no	yes	no
cancel()			yes	no				
close()	yes	yes	yes	no	yes	no	yes	no
connect()	yes	yes	yes	no	yes	no	yes	no
endhostent()	yes	n/a						
endnetent()	yes	n/a						
endprotoent()	yes	n/a						
endservent()	yes	n/a						
fcntl()	yes	no	yes	no	yes	no	yes	no
getclientid()	yes	no	yes	no	yes	no	yes	no
getdomainname							yes	no
getdtablesize()	yes	yes						
gethostbyaddr()	yes	no	yes	no	yes	no	yes	no
gethostbyname()	yes	n/a	yes	no	yes	no	yes	no
gethostent()	yes	n/a						
gethostid()	yes	no	yes	no	yes	no	yes	no
gethostname()	yes	no	yes	no	yes	no	yes	no
getibmopt()	yes	no	yes	no	yes	no		
getibmsockopt()	yes	no						

Table 23. Socket Address Families Cross Reference (continued)

Application Programming Interface (API)								
API TYPE	C SOCKETS		MACRO		CALL		REXX	
	INET	IUCV	INET	IUCV	INET	IUCV	INET	IUCV
getnetbyaddr()	yes	n/a						
getnetbyname()	yes	n/a						
getnetent()	yes	n/a						
getpeername()	yes	yes	yes	no	yes	no	yes	no
getprotobyname()	yes	n/a					n/a	n/a
getprotobynumber()	yes	n/a					yes	no
getprotoent()	yes	n/a						
getservbyname()	yes	n/a					yes	no
getservbyport()	yes	n/a					yes	no
getservent()	yes	n/a						
getsockname()	yes	yes	yes	no	yes	no	yes	no
getsockopt()	yes	no	yes	no	yes	no	yes	no
givesocket()	yes	no	yes	no	yes	no	yes	no
global			yes	no	yes	no		
htonl()	yes	n/a						
htons()	yes	n/a						
initapi			yes	no	yes	no		
inet_addr()	yes	n/a						
inet_inaof()	yes	n/a						
inet_makeaddr()	yes	n/a						
inet_netof()	yes	n/a						
inet_network()	yes	n/a						
inet_ntoa()	yes	n/a						
ioctl()	yes	no	yes	no	yes	no	yes	no
lasterrno()								
listen()	yes	yes	yes	no	yes	no	yes	no
maxdesc()	yes	yes						
ntohl()	yes	n/a						
ntohs()	yes	n/a						
read()	yes	yes	yes	no	yes	no	yes	no
readv()	yes	yes	yes	no	yes	no		
recv()	yes	yes	yes	no	yes	no	yes	no
recvfrom()	yes	yes	yes	no	yes	no	yes	no
recvmsg()	yes	yes	yes	no	yes	no		
resolve							yes	no
select()	yes	yes	yes	no	yes	no	yes	no
selectex()	yes	yes	yes	no	yes	no		
send()	yes	yes	yes	no	yes	no	yes	no

Table 23. Socket Address Families Cross Reference (continued)

Application Programming Interface (API)								
sendmsg()	yes	no	yes	no	yes	no		
API TYPE	C SOCKETS		MACRO		CALL		REXX	
	INET	IUCV	INET	IUCV	INET	IUCV	INET	IUCV
sendto()	yes	no	yes	no	yes	no	yes	no
setibmopt()	yes	no						
setibmsockopt()	yes	no						
sethostent()	yes	n/a						
setnetent()	yes	n/a						
setprotoent()	yes	n/a						
setservent()	yes	n/a						
setsockopt()	yes	no	yes	no	yes	no	yes	no
shutdown()	yes	yes	yes	no	yes	no	yes	no
sock_debug()	yes	yes						
sock_do_teststor()	yes	yes						
socket()	yes	yes	yes	no	yes	no	yes	no
takesocket()	yes	no	yes	no	yes	no	yes	no
task			yes	no	yes	no		
tcperror()	yes	yes						
termapi			yes	no	yes	no		
version							yes	no
write()	yes	yes	yes	no	yes	no	yes	no
writev()	yes	yes	yes	no	yes	no		

Appendix D. Related Protocol Specifications (RFCs)

This appendix lists the related protocol specifications for TCP/IP for MVS. The internet protocol suite is still evolving through requests for comments (RFC). New protocols are being designed and implemented by researchers and are brought to the attention of the internet community in the form of RFCs. Some of these are so useful that they become recommended protocols. That is, all future implementations for TCP/IP are recommended to implement these particular functions or protocols. These become the *de facto* standards, on which the TCP/IP protocol suite is built.

Many features of TCP/IP for MVS are based on the following RFCs:

RFC	**Title and Author**
768	*User Datagram Protocol* J.B. Postel
791	*Internet Protocol* J.B. Postel
792	*Internet Control Message Protocol* J.B. Postel
793	*Transmission Control Protocol* J.B. Postel
821	*Simple Mail Transfer Protocol* J.B. Postel
822	*Standard for the Format of ARPA Internet Text Messages* D. Crocker
823	*DARPA Internet Gateway* R.M. Hinden, A. Sheltzer
826	*Ethernet Address Resolution Protocol or Converting Network Protocol Addresses to 48.Bit Ethernet Address for Transmission on Ethernet Hardware* D.C. Plummer
854	*Telnet Protocol Specification* J.B. Postel, J.K. Reynolds
856	*Telnet Binary Transmission* J.B. Postel, J.K. Reynolds
857	*Telnet Echo Option* J.B. Postel, J.K. Reynolds
862	*Echo Protocol* J.B. Postel
863	*Discard Protocol* J.B. Postel
864	*Character Generator Protocol* J.B. Postel
877	*Standard for the Transmission of IP Datagrams over Public Data Networks* J.T. Korb
885	*Telnet End of Record Option* J.B. Postel
903	*Reverse Address Resolution Protocol* R. Finlayson, T. Mann, J.C. Mogul, M. Theimer
904	*Exterior Gateway Protocol Formal Specification* D.L. Mills
919	*Broadcasting Internet Datagrams* J.C. Mogul
922	*Broadcasting Internet Datagrams in the Presence of Subnets* J.C. Mogul
950	*Internet Standard Subnetting Procedure* J.C. Mogul, J.B. Postel
952	*DoD Internet Host Table Specification* K. Harrenstien, M.K. Stahl, E.J. Feinler
959	*File Transfer Protocol* J.B. Postel, J.K. Reynolds
974	*Mail Routing and the Domain Name System* C. Partridge
1006	*ISO Transport Service on tiop of the TCP Version:3* M.T.Rose, D.E. Cass

1009 *Requirements for Internet Gateways* R.T. Braden, J.B. Postel

1013 *X Window System Protocol, Version 11: Alpha Update* R.W. Scheifler

1014 *XDR: External Data Representation Standard* Sun Microsystems Incorporated

1027 *Using ARP to Implement Transparent Subnet Gateways* S. Carl-Mitchell, J.S. Quarterman

1032 *Domain Administrators Guide* M.K. Stahl

1033 *Domain Administrators Operations Guide* M. Lottor

1034 *Domain Names—Concepts and Facilities* P.V. Mockapetris

1035 *Domain Names—Implementation and Specification* P.V. Mockapetris

1042 *Standard for the Transmission of IP Datagrams over IEEE 802 Networks* J.B. Postel, J.K. Reynolds

1044 *Internet Protocol on Network System's HYPERchannel: Protocol Specification* K. Hardwick, J. Lekashman

1055 *Nonstandard for Transmission of IP Datagrams over Serial Lines: SLIP* J.L. Romkey

1057 *RPC: Remote Procedure Call Protocol Version 2 Specification* Sun Microsystems Incorporated

1058 *Routing Information Protocol* C.L. Hedrick

1091 *Telnet Terminal-Type Option* J. VanBokkelen

1094 *NFS: Network File System Protocol Specification* Sun Microsystems Incorporated

1118 *Hitchhikers Guide to the Internet* E. Krol

1122 *Requirements for Internet Hosts—Communication Layers* R.T. Braden

1123 *Requirements for Internet Hosts—Application and Support* R.T. Braden

1155 *Structure and Identification of Management Information for TCP/IP-Based Internets* M.T. Rose, K. McCloghrie

1156 *Management Information Base for Network Management of TCP/IP-based Internets* K. McCloghrie, M.T. Rose

1157 *Simple Network Management Protocol (SNMP),* J.D. Case, M. Fedor, M.L. Schoffstall, C. Davin

1179 *Line Printer Daemon Protocol* The Wollongong Group, L. McLaughlin III

1180 *TCP/IP Tutorial,* T.J. Socolofsky, C.J. Kale

1183 *New DNS RR Definitions* C.F. Everhart, L.A. Mamakos, R. Ullmann, P.V. Mockapetris, (Updates RFC 1034, RFC 1035)

1187 *Bulk Table Retrieval with the SNMP* M.T. Rose, K. McCloghrie, J.R. Davin

1188 *Proposed Standard for the Transmission of IP Datagrams over FDDI Networks* D. Katz

1198 *FYI on the X Window System* R.W. Scheifler

1207 *FYI on Questions and Answers: Answers to Commonly Asked "Experienced Internet User" Questions* G.S. Malkin, A.N. Marine, J.K. Reynolds

1208 *Glossary of Networking Terms* O.J. Jacobsen, D.C. Lynch

1213 *Management Information Base for Network Management of TCP/IP-Based Internets: MIB-II,* K. McCloghrie, M.T. Rose

1215 *Convention for Defining Traps for Use with the SNMP* M.T. Rose

1228 *SNMP-DPI Simple Network Management Protocol Distributed Program Interface* G.C. Carpenter, B. Wijnen

1229 *Extensions to the Generic-Interface MIB* K. McCloghrie

1230 *IEEE 802.4 Token Bus MIB IEEE 802 4 Token Bus MIB* K. McCloghrie, R. Fox

1231 *IEEE 802.5 Token Ring MIB IEEE 802.5 Token Ring MIB* K. McCloghrie, R. Fox, E. Decker

1267 *A Border Gateway Protocol 3 (BGP-3)* K. Lougheed, Y. Rekhter

1268 *Application of the Border Gateway Protocol in the Internet* Y. Rekhter, P. Gross

1269 *Definitions of Managed Objects for the Border Gateway Protocol (Version 3)* S. Willis, J. Burruss

1270 *SNMP Communications Services* , F. Kastenholz, ed.

1325 *FYI on Questions and Answers: Answers to Commonly Asked "New Internet User" Questions* G.S. Malkin, A.N. Marine

1340 *Assigned Numbers* J.K. Reynolds, J.B. Postel

1350 *TFTP Protocol* K.R. Sollins

1351 *SNMP Administrative Model* J. Davin, J. Galvin, K. McCloghrie

1352 *SNMP Security Protocols* J. Galvin, K. McCloghrie, J. Davin

1353 *Definitions of Managed Objects for Administration of SNMP Parties* K. McCloghrie, J. Davin, J. Galvin

1354 *IP Forwarding Table MIB* F. Baker

1356 *Multiprotocol Interconnect on X.25 and ISDN in the Packet Mode* A. Malis, D. Robinson, R. Ullmann

1374 *IP and ARP on HIPPI* J. Renwick, A. Nicholson

1381 *SNMP MIB Extension for X.25 LAPB* D. Throop, F. Baker

1382 *SNMP MIB Extension for the X.25 Packet Layer* D. Throop

1387 *RIP Version 2 Protocol Analysis* G. Malkin

1388 *RIP Version 2 — Carrying Additional Information* G. Malkin

1389 *RIP Version 2 MIB Extension* G. Malkin

1390 *Transmission of IP and ARP over FDDI Networks* D. Katz

1393 *Traceroute Using an IP Option* G. Malkin

1397 *Default Route Advertisement In BGP2 And BGP3 Versions of the Border Gateway Protocol* D. Haskin

1398 *Definitions of Managed Objects for the Ethernet-like Interface Types* F. Kastenholz

1540 *IAB Official Protocol Standards* J.B. Postel

These documents can be obtained from:

Government Systems, Inc.
Attn: Network Information Center
14200 Park Meadow Drive
Suite 200
Chantilly, VA 22021

Many RFCs are available online. Hard copies of all RFCs are available from the NIC, either individually or by subscription. Online copies are available using FTP from the NIC at `nic.ddn.mil`. Use FTP to download the files, using the following format:

```
RFC:RFC-INDEX.TXT
RFC:RFCnnnn.TXT
RFC:RFCnnnn.PS
```

where:

nnnn	Is the RFC number.
TXT	Is the text format.
PS	Is the PostScript format.

You can also request RFCs through electronic mail, from the automated NIC mail server, by sending a message to `service@nic.ddn.mil` with a subject line of `RFC` *nnnn* for text versions or a subject line of `RFC` *nnnn*`.PS` for PostScript versions. To request a copy of the RFC index, send a message with a subject line of `RFC INDEX`.

For more information, contact `nic@nic.ddn.mil`.

Appendix E. Abbreviations and Acronyms

This appendix lists the abbreviations and acronyms used throughout this book.

AIX	Advanced Interactive Executive
ANSI	American National Standards Institute
API	Application program interface
APPC	Advanced Program-to-Program Communications
APPN	Advanced Peer-to-Peer Networking
ARP	Address Resolution Protocol
ASCII	American National Standard Code for Information Interchange
ASN.1	Abstract Syntax Notation One
AUI	Attachment Unit Interface
BIOS	Basic Input/Output System
BNC	Bayonet Neill-Concelman
CCITT	Comite Consultatif International Telegraphique et Telephonique. The International Telegraph and Telephone Consultative Committee
CETI	Continuously Executing Transfer Interface
CLAW	Common Link Access to Workstation
CLIST	Command List
CMS	Conversational Monitor System
CP	Control Program
CPI	Common Programming Interface
CREN	Corporation for Research and Education Networking
CSD	Corrective Service Diskette
CTC	Channel-to-Channel
CU	Control Unit
CUA	Common User Access
DASD	Direct Access Storage Device
DBCS	Double Byte Character Set
DLL	Dynamic Link Library
DNS	Domain Name System
DOS	Disk Operating System
DPI	Distributed Program Interface
EBCDIC	Extended Binary-Coded Decimal Interchange Code
EISA	Enhanced Industry Standard Adapter
ELANS	IBM Ethernet LAN Subsystem

ESCON	Enterprise Systems Connection
FAT	File Allocation Table
FDDI	Fiber Distributed Data Interface
FTAM	File Transfer Access Management
FTP	File Transfer Protocol
FTP API	File Transfer Protocol Applications Programming Interface
GCS	Group Control System
GDDM	Graphical Data Display Manager
GDF	Graphics Data File
HCH**	HYPERchannel device**
HIPPI	High Performance Parallel Interface
HPFS	High Performance File System
ICAT	Installation Configuration Automation Tool
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronic Engineers
IETF	Internet Engineering Task Force
ILANS	IBM Token-Ring LAN Subsystem
IP	Internet Protocol
IPL	Initial Program Load
ISA	Industry Standard Adapter
ISDN	Integrated Services Digital Network
ISO	International Organization for Standardization
IUCV	Inter-User Communication Vehicle
JES	Job Entry Subsystem
JIS	Japanese Institute of Standards
JCL	Job Control Language
LAN	Local Area Network
LAPS	LAN Adapter Protocol Support
LCS	IBM LAN Channel Station
LPD	Line Printer Daemon
LPQ	Line Printer Query
LPR	Line Printer Client
LPRM	Line Printer Remove
LPRMON	Line Printer Monitor
LU	Logical Unit

MAC	Media Access Control
Mbps	Megabits per second
MBps	Megabytes per second
MCA	Micro Channel Adapter
MHS	Message Handling System
MIB	Management Information Base
MIH	Missing Interrupt Handler
MILNET	Military Network
MTU	Maximum Transmission Unit
MVS	Multiple Virtual Storage
MX	Mail Exchange
NCP	Network Control Program
NCS	Network Computing System
NDIS	Network Driver Interface Specification
NFS**	Network File System**
NIC	Network Information Center
NLS	National Language Support
NSFNET	National Science Foundation Network
OS/2	Operating System/2
OSF**	Open Software Foundation**, Inc.
OSI	Open Systems Interconnection
OSIMF/6000	Open Systems Interconnection Messaging and Filing/6000
OV/MVS	OfficeVision/MVS
OV/VM	OfficeVision/VM
PAD	Packet Assembly/Disassembly
PC	program call
PCA	Parallel Channel Adapter
PDN	Public Data Network
PDU	Protocol Data Units
PING	Packet Internet Groper
PIOAM	Parallel I/O Access Method
POP	Post Office Protocol
PROFS	Professional Office Systems

PSCA	Personal System Channel Attach
PSDN	Packet Switching Data Network
PU	Physical Unit
PVM	Passthrough Virtual Machine
RACF	Resource Access Control Facility
RARP	Reverse Address Resolution Protocol
REXEC	Remote Execution
REXX	Restructured Extended Executor Language
RFC	Request For Comments
RIP	Routing Information Protocol
RISC	Reduced Instruction Set Computer
RPC	Remote procedure call
RSCS	Remote Spooling Communications Subsystem
SAA	System Application Architecture
SBCS	Single Byte Character Set
SDLC	Synchronous Data Link Control
SLIP	Serial Line Internet Protocol
SMI	Structure for Management Information
SMTP	Simple Mail Transfer Protocol
SNA	Systems Network Architecture
SNMP	Simple Network Management Protocol
SOA	Start of Authority
SPOOL	Simultaneous Peripheral Operations Online
SQL	IBM Structured Query Language
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
TFTP	Trivial File Transfer Protocol
TSO	Time Sharing Option
TTL	Time-to-Live
UDP	User Datagram Protocol
VGA	Video Graphic Array
VM	Virtual Machine
VMCF	Virtual machine communication facility
VM/SP	Virtual Machine/System Product

VM/XA	Virtual Machine/Extended Architecture
VTAM	Virtual Telecommunications Access Method
WAN	Wide Area Network
XDR	eXternal Data Representation

Appendix F. How to Read a Syntax Diagram

The syntax diagram shows you how to specify a command so that the operating system can correctly interpret what you type. Read the syntax diagram from left to right and from top to bottom, following the horizontal line (the main path).

Symbols and Punctuation

The following symbols are used in syntax diagrams:

►► Marks the beginning of the command syntax.

► Indicates that the command syntax is continued.

| Marks the beginning and end of a fragment or part of the command syntax.

►◄ Marks the end of the command syntax.

You must include all punctuation such as colons, semicolons, commas, quotation marks, and minus signs that are shown in the syntax diagram.

Parameters

The following types of parameters are used in syntax diagrams.

Required
Required parameters are displayed on the main path.

Optional
Optional parameters are displayed below the main path.

Default
Default parameters are displayed above the main path.

Parameters are classified as keywords or variables. Keywords are displayed in uppercase letters and can be entered in uppercase or lowercase. For example, a command name is a keyword.

Variables are italicized, appear in lowercase letters, and represent names or values you supply. For example, a data set is a variable.

Syntax Examples

In the following example, the USER command is a keyword. The required variable parameter is *user_id*, and the optional variable parameter is *password*. Replace the variable parameters with your own values.

```
►►──USER──user_id──┬──────────┬──────────────────────────────────────────►◄
                   └─password─┘
```

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.

```
►►──┤ First Line ├──OPERAND1──OPERAND2──OPERAND3──OPERAND4──OPERAND5──OPERAND6───►
```

```
►─┤ Second Line ├──────────────────────────────────────────────────►◄
```

Required operands: Required operands and values appear on the main path line.

```
►►──REQUIRED_OPERAND───────────────────────────────────────────────►◄
```

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.

```
►►───┬─REQUIRED_OPERAND_OR_VALUE_1─┬───────────────────────────────►◄
     └─REQUIRED_OPERAND_OR_VALUE_2─┘
```

Optional values: Optional operands and values appear below the main path line.

```
►►───┬─────────┬───────────────────────────────────────────────────►◄
     └─OPERAND─┘
```

You can choose not to code optional operands and values.

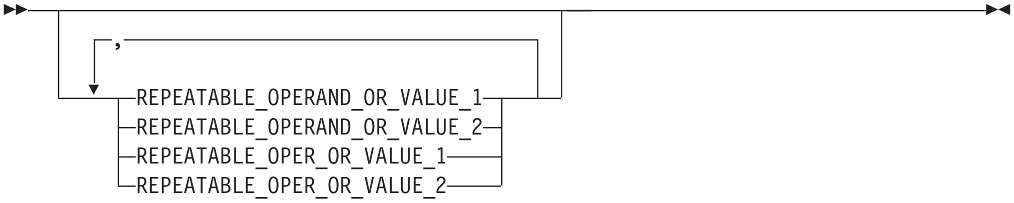
Choose one optional operand from a stack: If there is more than one mutually exclusive optional operand or value to choose from, they are stacked vertically in alphanumeric order below the main path line.

```
►►───┬───────────────────┬─────────────────────────────────────────►◄
     ├─OPERAND_OR_VALUE_1─┤
     └─OPERAND_OR_VALUE_2─┘
```

Repeating an operand: An arrow returning to the left above an operand or value on the main path line means that the operand or value can be repeated. The command means that each operand or value must be separated from the next by a comma.

```
     ┌─,─────────────────┐
     ▼                   │
►►─────REPEATABLE_OPERAND─┴────────────────────────────────────────►◄
```

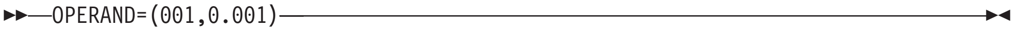
Selecting more than one operand: An arrow returning to the left above a group of operands or values means more than one can be selected, or a single one can be repeated.


,
REPEATABLE_OPERAND_OR_VALUE_1
REPEATABLE_OPERAND_OR_VALUE_2
REPEATABLE_OPER_OR_VALUE_1
REPEATABLE_OPER_OR_VALUE_2

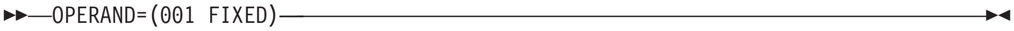
If an operand or value can be abbreviated, the abbreviation is described in the text associated with the syntax diagram.

Case Sensitivity: TCP/IP commands are not case sensitive. You can code them in upppercase or lowercase.

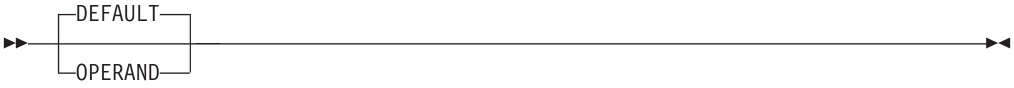
Nonalphanumeric characters: If a diagram shows a character that is not alphanumeric (such as parentheses, periods, commas, and equal signs), you must code the character as part of the syntax. In this example, you must code OPERAND=(001,0.001).



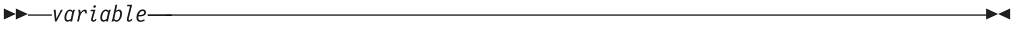
Blank spaces in syntax diagrams: If a diagram shows a blank space, you must code the blank space as part of the syntax. In this example, you must code OPERAND=(001 FIXED).



Default operands: Default operands and values appear above the main path line. TCP/IP uses the default if you omit the operand entirely.



Variables: A word in all lowercase italics is a *variable*. Where you see a variable in the syntax, you must replace it with one of its allowable names or values, as defined in the text.



Syntax fragments: Some diagrams contain syntax fragments, which serve to break up diagrams that are too long, too complex, or too repetitious. Syntax fragment names are in mixed case and are shown in the diagram and in the heading of the fragment. The fragment is placed below the main diagram.

```
►►──┤ Reference to Syntax Fragment ├──────────────────────────────────►◄
```

Syntax Fragment:

```
├──1ST_OPERAND,2ND_OPERAND,3RD_OPERAND──────────────────────────────────┤
```

References to syntax notes appear as numbers enclosed in parentheses above the line. Do not code the parentheses or the number.

```
              (1)
►►──OPERAND──────────────────────────────────────────────────────────►◄
```

Notes:

1 An example of a syntax note.

Appendix G. Information Apars

This appendix lists information apars for IP-related books.

Notes:

1. Information apars contain updates to previous editions of the manuals listed below. Books updated for V2R10 contain all the updates except those contained in the information apars that may be issued after V2R10 books went to press.
2. Information apars are predefined for CS for OS/390 V2R10 and may not contain updates.

IP Information Apars

Table 24 lists information apars for IP-related books.

Table 24. IP Information Apars

Title	CS for OS/390 2.10	CS for OS/390 2.8	CS for OS/390 2.7	CS for OS/390 2.6	CS for OS/390 2.5	TCP/IP 3.3
High Speed Access Service User's Guide (GC31-8676)		ii11629	ii11566	ii11412	ii11181	
IP API Guide (SC31-8516)	II12371	ii11635	ii11558	ii11405	ii11144	
IP CICS Sockets Guide (SC31-8518)		ii11626	ii11559	ii11406	ii11145	
IP Configuration (SC31-8513)		ii11620 ii12068	ii11555 ii11637 ii11995	ii11402 ii11619 ii12066	ii11159 ii11979	ii10633
IP Configuration Guide (SC31-8725)	II12362					
IP Configuration Reference (SC31-8726)	II12363					
IP Diagnosis (SC31-8521)	II12366	ii11628	ii11565	ii11411	ii11160 ii11414	ii10637
IP Messages Volume 1 (SC31-8517)	II12367	ii11630	ii11562	ii11408		Messages and Codes ii10635
IP Messages Volume 2 (SC31-8570)	II12368	ii11631	ii11563	ii11409		
IP Messages Volume 3 (SC31-8674)	II12369	ii11632	ii11564	ii11410	ii11158	
IP Migration (SC31-8512)	II12361	ii11618	ii11554	ii11401		

Table 24. IP Information Apars (continued)

Title	CS for OS/390 2.10	CS for OS/390 2.8	CS for OS/390 2.7	CS for OS/390 2.6	CS for OS/390 2.5	TCP/IP 3.3
IP Network Print Facility (SC31-8522)		ii11627	ii11561	ii11407	ii11150	
IP Programmer's Reference (SC31-8515)		ii11634	ii11557	ii11404		ii10636
IP and SNA Codes (SC31-8571)	II12370	ii11917	Added TCP/IP codes to VTAM codes V2R6 ii11611	ii11361	ii11146 ii11097	
IP User's Guide (GC31-8514)	II12365	ii11625	ii11556	ii11403	ii11143	ii10634

Appendix H. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION ″AS IS″ WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Site Counsel
IBM Corporation
P.O.Box 12195
3039 Cornwallis Road
Research Triangle Park, North Carolina 27709-2195
U.S.A

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, photographs and color illustrations may not appear.

This book is available only in softcopy. You can obtain softcopy from the OS/390 Online Library Collection (SK2T-6700), the OS/390 PDF Library Collection (SK2T-6718), or the OS/390 Internet Library (*http://www.ibm.com/s390/os390/*).

Programming Interface Information

This publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM Communications Server for OS/390.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

ACF/VTAM
Advanced Peer-to-Peer Networking
AFP
AD/Cycle
AIX
AIX/ESA
AnyNet
APL2
APPN
AS/400
AT
BookManager
BookMaster
CBPDO
C/370
CICS
CICS/ESA
C/MVS
Common User Access
C Set ++
CT
CUA
DATABASE 2
DatagLANce
DB2
DFSMS
DFSMSdfp
DFSMShsm
DFSMS/MVS
Domino
DRDA
eNetwork
Enterprise Systems Architecture/370
ESA/390
ESCON
ES/3090
ES/9000
ES/9370
EtherStreamer
Extended Services
FAA

Micro Channel
MVS
MVS/DFP
MVS/ESA
MVS/SP
MVS/XA
MQ
Natural
NetView
Network Station
Nways
Notes
NTune
NTuneNCP
OfficeVision/MVS
OfficeVision/VM
Open Class
OpenEdition
OS/2
OS/390
Parallel Sysplex
Personal System/2
PR/SM
PROFS
PS/2
RACF
Resource Measurement Facility
RETAIN
RFM
RISC System/6000
RMF
RS/6000
S/370
S/390
SAA
SecureWay
Slate
SP
SP2
SQL/DS
System/360

FFST
FFST/2
FFST/MVS
First Failure Support Technology
GDDM
Hardware Configuration Definition
IBM
IBMLink
IMS
IMS/ESA
InfoPrint
Language Environment
LANStreamer
Library Reader
LPDA
MCS

System/370
System/390
SystemView
Tivoli
TURBOWAYS
UNIX System Services
Virtual Machine/Extended Architecture
VM/ESA
VM/XA
VSE/ESA
VTAM
WebSphere
XT
400
3090
3890

Lotus, Freelance, and Word Pro are trademarks of Lotus Development Corporation in the United States, or other countries, or both.

Tivoli and NetView are trademarks of Tivoli Systems Inc. in the United States, or other countries, or both.

The following terms are trademarks of other companies:

ATM is a trademark of Adobe Systems, Incorporated.

BSC is a trademark of BusiSoft Corporation.

CSA is a trademark of Canadian Standards Association.

DCE is a trademark of The Open Software Foundation.

HYPERchannel is a trademark of Network Systems Corporation.

UNIX is a registered trademark in the United States, other countries, or both and is licensed exclusively through X/Open Company Limited.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

ActionMedia, LANDesk, MMX, Pentium, and ProShare are trademarks of Intel Corporation in the United States, other countries, or both. For a complete list of Intel trademarks, see http://www.intel.com/tradmarx.htm.

Other company, product, and service names may be trademarks or service marks of others.

Bibliography

IBM Communications Server for OS/390 Publications

This bibliography contains descriptions of the books in the IBM Communications Server for OS/390 library.

Updates to books are available on RETAIN. See "Appendix G. Information Apars" on page 581 for a list of the books and the INFOAPARS associated with them.

These books are available online. Go to http://www.ibm.com/s390/os390/bkserv/ to access the OS/390 Internet Library web page.

Some books are available in both hard- and soft-copy, or soft-copy only. The following abbreviations follow each order number:

HC/SC	Both hard- and soft-copy are available
SC	Only soft-copy is available

Related Publications

For information about OS/390 products, refer to *OS/390 Information Roadmap* (GC28-1727-07 [HC/SC]). The Roadmap describes what level of documents are supplied with each release of CS for OS/390, as well as describing each OS/390 publication.

Firewall

OS/390 SecureWay Security Server Firewall Technologies Guide and Reference (SC24-5835 [HC/SC])

OSA-Express

OSA-Express Customer's Guide and Reference (SA22-7403 [HC/SC])

Softcopy Information

- *OS/390 Online Library Collection* (SK2T-6700).

  This collection contains softcopy unlicensed books for OS/390, Parallel Sysplex products, and S/390 application programs that run on OS/390. This collection is updated quarterly with any new or updated books that are available for the product libraries included in it.
- *OS/390 PDF Library Collection* (SK2T-6718).

  This collection contains the unlicensed books for OS/390 Version 2 Release 6 in Portable Document Format (PDF).
- *OS/390 Licensed Product Library* (LK2T-2499).

  This library contains unencrypted softcopy licensed books for OS/390 Version 2. If any of the books in this library are changed, it is updated quarterly. The OS/390 Licensed Product Library for Version 1 (LK2T-6702) is still available, but is no longer updated.
- *System Center Publication IBM S/390 Redbooks Collection* (SK2T-2177).

  This collection contains over 300 ITSO redbooks that apply to the S/390 platform and to host networking arranged into subject bookshelves.

Planning

OS/390 IBM Communications Server: SNA Migration (SC31-8622 [HC/SC]). This book is intended to help you plan for SNA, whether you are migrating from a previous version or installing SNA for the first time. This book also identifies the optional and required modifications needed to enable you to use the enhanced functions provided with SNA.

OS/390 IBM Communications Server: IP Migration (SC31-8512 [HC/SC]). This book is intended to help you plan for IP, whether you are migrating from a previous version or installing IP for the first time. This book also identifies the optional and required modifications needed to enable you to use the enhanced functions provided with IP.

Resource Definition, Configuration, and Tuning

OS/390 IBM Communications Server: IP Configuration Guide (SC31-8725 [HC/SC]). This book describes the major concepts involved in understanding and configuring an IP network. Familiarity with MVS operating, IP protocols, OS/390 UNIX System Services, and IBM Time Sharing Option (TSO) is recommended. Use this book in conjunction with the *OS/390 IBM Communications Server: IP Configuration Reference*.

OS/390 IBM Communications Server: IP Configuration Reference (SC31-8726 [HC/SC]). This book presents information for people who want to administer and maintain IP. Use this book in conjunction with the *OS/390 IBM Communications Server: IP Configuration Guide*. The information in this book includes:

- TCP/IP configuration data sets
- Configuration statements
- Operator commands
- Translation tables
- SMF records
- Protocol number and port assignments

OS/390 IBM Communications Server: SNA Network Implementation Guide (SC31-8563 [HC/SC]). This book presents the major concepts involved in implementing a SNA network. Use this book in conjunction with the *OS/390 IBM Communications Server: SNA Resource Definition Reference*.

OS/390 IBM Communications Server: SNA Resource Definition Reference (SC31-8565 [HC/SC]). This book describes each SNA definition statement, start option, and macroinstruction for user tables. It also describes NCP definition statements that affect SNA.Use this book in conjunction with the *OS/390 IBM Communications Server: SNA Network Implementation Guide*.

OS/390 IBM Communications Server: SNA Resource Definition Reference (SC31-8566 [HC/SC]). This book contains sample definitions to help you implement SNA functions in your networks, and includes sample major node definitions.

OS/390 eNetwork Communications Server: AnyNet SNA over TCP/IP (SC31-8578 [SC]). 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 [SC]). This guide provides information to help you install, configure, use, and diagnose Sockets over SNA. It also provides information to help you prepare application programs to use sockets over SNA.

Operation

OS/390 IBM Communications Server: IP User's Guide (GC31-8514 [HC/SC]). 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 IBM Communications Server: SNA Operation (SC31-8567 [HC/SC]). This book serves as a reference for programmers and operators requiring detailed information about specific operator commands.

OS/390 IBM Communications Server: Quick Reference (SX75-0121 [HC/SC]). This book contains essential information about SNA and IP commands.

OS/390 eNetwork Communications Server: High Speed Access Services Users Guide (GC31-8676 [SC]). This book is for end users and system administrators who want to use applications using a High Speed Access Services connection available in CS for OS/390.

Customization

OS/390 IBM Communications Server: SNA Customization (LY43-0110 [SC]). This book enables you to customize SNA, and includes:

- Communication network management (CNM) routing table
- Logon-interpret routine requirements
- Logon manager installation-wide exit routine for the CLU search exit
- TSO/SNA installation-wide exit routines
- SNA installation-wide exit routines

OS/390 eNetwork Communications Server: IP Network Print Facility (SC31-8522 [SC]). This book is for system programmers and network administrators who need to prepare their network to route SNA, JES2, or JES3 printer output to remote printers using TCP/IP.

Writing Application Programs

OS/390 IBM Communications Server: IP Application Programming Interface Guide (SC31-8516 [SC]). This book describes the syntax and semantics of program source code necessary to write your own application programming interface (API) into TCP/IP. You can use this interface as the communication base for writing your own client or server application. You can also use this book to adapt your existing applications to communicate with each other using sockets over TCP/IP.

OS/390 Secureway Communications Server: IP CICS Sockets Guide (SC31-8518 [SC]). This book is for people who want to set up, write application programs for, and diagnose problems with the socket interface for CICS using TCP/IP for MVS.

OS/390 eNetwork Communications Server: IP IMS Sockets Guide (SC31-8519 [SC]). 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 IBM Communications Server: IP Programmer's Reference (SC31-8515 [SC]). This book describes the syntax and semantics of a set of high-level application functions that you can use to program your own applications in a TCP/IP environment. These functions provide support for application facilities, such as user authentication, distributed databases, distributed processing, network management, and device sharing. Familiarity with the MVS operating system, TCP/IP protocols, and IBM Time Sharing Option (TSO) is recommended.

OS/390 IBM Communications Server: SNA Programming (SC31-8573 [SC]). This book describes how to use SNA macroinstructions to send data to and receive data from (1) a terminal in either the same or a different domain, or (2) another application program in either the same or a different domain.

OS/390 eNetwork Communications Server: SNA Programmers LU 6.2 Guide (SC31-8581 [SC]). This book describes how to use the SNA LU 6.2 application programming interface for host application programs. This book applies to programs that use only LU 6.2 sessions or that use LU 6.2 sessions along with other session types. (Only LU 6.2 sessions are covered in this book.)

OS/390 eNetwork Communications Server: SNA Programmers LU 6.2 Reference (SC31-8568 [SC]). This book provides reference material for the SNA LU 6.2 programming interface for host application programs.

OS/390 eNetwork Communications Server: CSM Guide (SC31-8575 [SC]). This book describes how applications use the communications storage manager.

OS/390 IBM Communications Server: CMIP Services and Topology Agent Guide (SC31-8576 [SC]). This book describes the Common Management Information Protocol (CMIP) programming interface for application programmers to use in coding CMIP application programs. The book provides guide and reference information about CMIP services and the SNA topology agent.

Diagnosis

OS/390 IBM Communications Server: IP Diagnosis (SC31-8521 [HC/SC]). This book explains how to diagnose TCP/IP problems and how to determine whether a specific problem is in the TCP/IP product code. It explains how to gather information for and describe problems to the IBM Software Support Center.

OS/390 IBM Communications Server: SNA Diagnosis V1 Techniques and Procedures(LY43-0079 [HC/SC]) and *OS/390 IBM Communications Server: SNA Diagnosis V2 FFST Dumps and the VIT* (LY43-0080 [HC/SC]). These books help you identify a SNA problem, classify it, and collect information about it before you call the IBM Support Center. The information collected includes traces, dumps, and other problem documentation.

OS/390 IBM Communications Server: SNA Data Areas Volume 1 (LY43-0111 [SC]) and *OS/390 IBM Communications Server: SNA Data Areas Volume 2* (LY43-0112 [SC]). These books describe SNA data areas and can be used to read a SNA dump. They are intended for IBM programming service representatives and customer personnel who are diagnosing problems with SNA.

Messages and Codes

OS/390 IBM Communications Server: SNA Messages (SC31-8569 [HC/SC]). This book describes the ELM, IKT, IST, ISU, IVT, IUT, and USS messages. Other information in this book includes:

- Command and RU types in SNA messages
- Node and ID types in SNA messages
- Supplemental message-related information

OS/390 IBM Communications Server: IP Messages Volume 1 (EZA) (SC31-8517 [HC/SC]). This volume contains TCP/IP messages beginning with EZA.

OS/390 IBM Communications Server: IP Messages Volume 2 (EZB) (SC31-8570 [HC/SC]). This volume contains TCP/IP messages beginning with EZB.

OS/390 IBM Communications Server: IP Messages Volume 3 (EZY-EZZ-SNM) (SC31-8674 [HC/SC]). This volume contains TCP/IP messages beginning with EZY, EZZ, and SNM.

OS/390 IBM Communications Server: IP and SNA Codes (SC31-8571 [HC/SC]). This book describes codes and other information that display in CS for OS/390 messages.

APPC Application Suite

OS/390 eNetwork Communications Server: APPC Application Suite User's Guide (GC31-8619 [SC]). This book documents the end-user interface (concepts, commands, and messages) for the AFTP, ANAME, and APING facilities of the APPC application suite. Although its primary audience is the end user, administrators and application programmers may also find it useful.

OS/390 eNetwork Communications Server: APPC Application Suite Administration (SC31-8620 [SC]). This book contains the information that administrators need to configure the APPC application suite and to manage the APING, ANAME, AFTP, and A3270 servers.

OS/390 eNetwork Communications Server: APPC Application Suite Programming (SC31-8621 [SC]). 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)

Redbooks

The following Redbooks may help you as you implement CS for OS/390.

- *OS/390 eNetwork Communication Server V2R7 TCP/IP Implementation Guide Volume 1: Configuration and Routing* (SG24–5227–01).

  This book provides examples of how to configure the base TCP/IP stack, routing daemons and the TELNET server. This book also provides information about national language support (NLS), routing, OSPF, network interfaces, diagnosis, multicasting, OS/390 UNIX System Services and security in an OS/390 UNIX System Services environment.
- *OS/390 eNetwork Communication Server V2R7 TCP/IP Implementation Guide Volume 2: UNIX Applications* (SG24–5228–01).

  This book provides information about implementing applications that run in the OS/390 UNIX environment, such as FTP, SNMP, BIND-based name server, DHCP, and SENDMAIL. This book also provides configuration samples and describes the implementation process.
- *OS/390 eNetwork Communication Server V2R7 TCP/IP Implementation Guide Volume 3: MVS Applications* (SG24–5229–01).

  This book provides information about TCP/IP applications that run in a legacy MVS environment, including CICS/IMS Sockets, and printing (NPF, LPR, and LPD.)
- *OS/390 Secureway Communication Server V2R8 TCP/IP Guide to Enhancements* (SG24-5631-00).

  This redbook provides information to facilitate the configuration and use of the new technologies and functions supported in SecureWay Communications Server for OS/390 V2R8. Special areas of interest in this book are security and Quality of Services.
- *TCP/IP in a Sysplex* (SG24–5235–01).

  The main goals of a Parallel Sysplex are high availability and high performance. This book demonstrates how these goals can be achieved in the particular environment of SecureWay Communications Server for OS/390 and its TCP/IP applications. This book describes the WLM/DNS functions, the Network Dispatcher and the Dynamic VIPA.
- *SNA and TCP/IP Integration* (SG24–5291–00).

  This book provides information about integrating current SNA network with future TCP/IP and Web-based communication requirements. This book concentrates on routing techniques.
- *SNA in a Parallel Sysplex Environment* (SG24–2113–01).

This book provides information about implementing a VTAM-based network on a Parallel Sysplex.

- *Subarea to APPN Migration : VTAM and APPN Implementation* (SG24–4656–01).

  This book is the first of two volumes. This book provides information about the migration of a subarea network to an APPN network. Some knowledge of SNA subarea networks and familiarity with the functions, terms and data flows of APPN networks is assumed.
- *Subarea to APPN Migration : HPR and DLUR Implementation* (SG24–5204–00).

  This book is the second of two volumes. This book provides information about the coverage of a network using HPR, DLUR and APPN/HPR routers. Some knowledge of SNA subarea networks and familiarity with the functions, terms and data flows of APPN networks is assumed.

Index

Readers' Comments — We'd Like to Hear from You

OS/390 IBM Communications Server
IP Application Programming Interface Guide
Version 2 Release 10

Publication No. SC31-8516-05

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
SC31-8516-05

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Software Reengineering
Department G7IA/ Bldg 503
Research Triangle Park, NC
27709-9990

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

Program Number: 5647–A01

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC31-8516-05

Spine information:

OS/390 IBM Communications
Server

OS/390 IBM CS V2R10.0: IP Application
Programming Interface Guide

Version 2
Release 10