Programmer's Guide

TCP/IP for VSE

Version 2 Release 2

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

This manual describes the application programming interfaces available with TCP/IP FOR VSE.

Published October 2017

CSI INTERNATIONAL

"Delivering what the competition can only promise"

www.csi-international.com • info@csi-international.com • 800.795.4914

TCP/IP FOR VSE Programmer's Guide
Version 2 Release 2
October 2017

Published by CSI International

Phone:	800-795-4914
Fax:	740-986-6022
Internet:	http://www.csi-international.com
Product questions:	info@csi-international.com
Technical support:	support@csi-international.com
Review comments:	documentation@csi-international.com

CSI International Technical Support

During Business Hours

Monday through Friday, 9:00 A.M. through 5:00 P.M. EST/EDT.

Telephone:	Toll Free in the USA	800-795-4914
	Worldwide	740-420-5400

Email: support@csi-international.com

Web: http://csi-international.com/problemreport_vse.htm

Emergency Service 24/7

After business hours and 24 hours on Saturday and Sunday:

Telephone:	Toll Free in the USA:	800-795-4914
	Worldwide:	740-420-5400

CSI International provides support to address each issue according to its severity.

Updates to This Manual

The following table describes updates to this manual. Updates may be identified by a fix number in CSI International's support database.

October 2017

ID	Change Description	Page
	Chapter 6, SSL/TLS for VSE APIs, "Secure Socket Layer API":	
	• For each function, added a reference to Appendix C: TLS 1.2 Enhancement.	125
	• Added a note about TLS 1.2 to "Hardware Assist Options Settings."	125
	• gsk_initialize( ): Added the TLS 1.2 protocol to sec_types.	132
	• gsk_secure_soc_init( ): Added the TLS 1.2 protocol to sec_type.	138
	Appendix A: $SOCKOPT Options Phase:	
	• Updated options for TLS 1.2.	196
	• Added section: "New $SOCKOPT Settings for TLS 1.2."	201
	Appendix C: TLS 1.2 Enhancement:	
	Added Appendix C to cover TLS 1.2 support.	208

Table of Contents

To return from a hyperlink jump, press <Alt> <◄>

1

SOCKET Assembler API

SOCKET Macro

The SOCKET macro is the lowest level interface to TCP/IP FOR VSE. Because it is an Assembler interface, it gives the Assembler programmer more flexibility than any other programming interface. You can use the SOCKET macro to communicate with TCP, UDP, TELNET, and FTP. The SOCKET macro also acts as a general-purpose client interface.

Syntax

The syntax of the SOCKET macro follows. The defaults are underlined.

```
label     SOCKET function,type,
                ACTIVE=[YES|NO],
                CICS=[YES|NO],
                DATA=[(address,length)|NULL],
                DESC=descriptor,
                ECB=resultarea,
                ECB2=2ndecbaddr,
                FAST=[YES|NO],
                FOIP=[0|foreignipaddr],
                FOPORT=[0|foreignportnum],
                ID=[00|nn],
                LOPORT=[0|localportnum],
                MF=[E|L],
                MFG=soparea,
                NEGOT=[YES|NO],
                TIMED=[YES|NO],
                TIMEOUT=[36000|timevalue],
                USESYS=[YES|NO],
                WAIT=[YES|NO]
```

The caller must meet the following requirements:

- AMODE must be 24 or 31.
- RMODE must be 24 or ANY.

Also, the SOCKET macro issues SVCs that must be executed in an enabled state.

function Operand

The first operand of the SOCKET macro is *function*, which indicates the function of the macro. The following table describes the valid values.

function **Value**	**Description**
ABORT	When you specify ABORT, all pending sends and receives are aborted and the socket is closed. This is an abnormal close and may cause a reset message to appear at the other side of the connection.
CLOSE	When you specify CLOSE, the specified connection is closed. The close operation is intended to be graceful in the sense that outstanding SENDs are transmitted until all data is sent. It is acceptable to issue several SEND calls followed by a CLOSE call and to expect all data to be sent to the destination. The user can close the connection at any time. Because closing a connection requires communication with the foreign socket, connections can remain in the closing state for a short time.
DSECT	When you specify DSECT, the SOCKET macro produces the SRBLOK DSECT. The SRBLOK DSECT maps the 56-byte result area specified by the ECB operand.
OPEN	When you specify OPEN, you must also specify whether the connection is PASSIVE or ACTIVE. If you set PASSIVE=YES, you are listening for an incoming connection. A passive open can have either a fully specified foreign socket (FOIP=*nnn.nnn.nnn.nnn* and FOPORT=*nnnnn*) waiting for a specific client connection, or an unspecified foreign socket (FOIP=0 and FOPORT=0) waiting for any client connection. A server application normally begins with a passive open on a preassigned port number and then waits for client connections. To connect to the server application, a client must know the server's IP address and port number. If you set ACTIVE=YES, the procedure to establish the connection begins immediately. A client application normally begins with an active open and then attempts to connect to a specific server application that is passively listening (waiting for client connections). The client’s IP address and port number are sent to the server during active open processing, thus enabling the server to send data back to the client.

function **Value**	**Description**
RECEIVE	When you specify RECEIVE, a receiving buffer is filled with all available data or as much data as it can hold. For example, if you specify 2000 bytes and only 100 arrive, then 100 bytes are received; the remaining buffer is empty. Also, if you specify 100 bytes but 2000 bytes are sent, then 100 bytes are received and the remaining 1900 bytes require additional RECEIVES. The SRECB of the result area is posted when the receiving buffer contains data, and the SRCOUNT field of the result area contains the length of the received data. There is no inherent structure in the data stream. The application must issue multiple RECEIVEs until the correct amount of data is retrieved. Also, if you expect and specify a given amount, the amount received may still be less if the sending stack divides the data. You must design the application program to dynamically determine when incoming data segments are complete. To do this, you can include a length prefix or use unique delimiters.
SEND	When you specify SEND, the data contained in the specified user buffer is sent. The SRECB of the result area is posted when the data buffer is accepted by the TCP/IP FOR VSE partition.
SET_SYSID	When you specify SET_SYSID, you enable the macro to alter the default TCP/IP FOR VSE stack ID to which the application program connects. If you specify USESYS=YES on the OPEN request and the // OPTION SYSPARM='*xx*' contains a two-byte value, that specification overrides the SET_SYSID command. To make application testing easier, we recommend that you use the // OPTION SYSPARM=xx and USESYS=YES instead of the SET_SYSID command to specify the TCP/IP FOR VSE stack.
STATUS	When you specify STATUS, you receive a data block containing the following information: local socket, foreign socket, local connection name, receive window, send window, connection state, number of buffers awaiting acknowledgment, number of buffers pending receipt, urgent state, and transmission timeout. Based on the connection state or the implementation itself, some of this information may not be available or meaningful. The DSECT that maps this data is named STATBLK. It is mapped by the STATBLK macro.

type **Operand**

The SOCKET macro's second operand is *type*, which indicates the connection type you are using. The following table describes the valid values.

***type* Value**	**Description**
CLIENT	Connects to the general-purpose client manager. The client manager executes in the TCP/IP FOR VSE partition and controls the functions of some independent protocols. For example, an application program using the general client manager can manipulate LPR.
CONTROL	Connects to a control manager. The control manager allows an application program to request system services from the TCP/IP FOR VSE partition. For example, you can use the control interface to translate a domain name to an IP address.
FTP	Connects to FTP. The FTP client manager executes in the TCP/IP FOR VSE partition and manages the FTP session. After the open request is complete, an application program can send FTP commands and receive responses to those commands across the connection. This enables an application program to manipulate an FTP client session.
TCP	Uses the TCP interface to connect and communicate. This enables you to establish a TCP connection in client or server mode so you can transfer a TCP data stream to or from an application program.
TELNET	Connects to a TELNET client manager. The TELNET client manager executes in the TCP/IP FOR VSE partition and manages the TELNET protocol. After a TELNET open completes, data is sent and received as a TELNET stream. The application program does not need to worry about ASCII-to-EBCDIC translation or the details of TELNET option negotiations.
UDP	Allows you to transmit or receive a UDP datagram from a foreign port and IP address. Because UDP is not a connection-based system, it passes data in the form of blocks called *UDP datagrams*.

Keyword Parameters

The remaining parameters in the SOCKET macro are keyword operands. These are described in the following table:

Keyword	**Description**
ACTIVE= [YES|NO]	Indicates whether TCP/IP FOR VSE should attempt to connect with a foreign host or wait for a foreign host to connect with it (listen state). If *YES*, the connection is established. If you specify UDP as your *connect* value, no actual connection is established. The default is NO.
CICS=[YES|NO]	Indicates whether the SOCKET macro should generate code that works in the CICS environment. If YES, the appropriate EXEC CICS calls are generated to replace the standard operating system GETVIS and WAIT calls. The default is NO.
DATA= [(*address,length*)| NULL]	Identifies either a block of data to be transmitted or an area to be used for a receive operation. You can specify a simple '*string*' for a transmission. When you refer to a data buffer, you must specify the address and length as shown in the macro format. These fields can refer to a fullword containing the address or to a register containing the address ((reg1),(reg2)). If you specify the keyword value NULL with a receive request, you receive a completion signal without any transfer of data, and you can schedule subsequent receives to retrieve the data. There is no default.
DESC=*descriptor*	Identifies the thread of operation. This field must point to a fullword in storage in which the descriptor value can be stored, and this descriptor must be passed to all subsequent SOCKET calls for the open connection. In other words, after a connection is established, any number of independent SOCKET calls can be made to the connection, but all of them must use the DESC field to identify the open connection. The descriptor is established by the OPEN call and is the address of the socket work area (SOWORK). This area is obtained dynamically, used for all additional SOCKET calls, and then released by the CLOSE call. There is no default.
ECB=*resultarea*	Points to a 14-fullword area containing SOCKET macro results. After a SOCKET request completes, the results are stored in the specified area. This area is initialized by the SOCKET call. There is no default.

Keyword	**Description**
ECB2=*2ndecbaddr*	Points to a second ECB. This fullword ECB is posted at SOCKET macro completion. To control the wait on SOCKET operations, an application can use either the original ECB result area or this second ECB area. There is no default. Unlike the ECB= parameter, this ECB is not initialized by the SOCKET call. This means that many concurrent operations may share the same ECB2= value, and when it is posted, the application can examine the individual ECB= fields to determine the completed operation(s).
FAST=[YES|NO]	For FAST=NO (the default), the send or close ECB is posted when the remote host acknowledges the request. If FAST=YES is specified on either the CLOSE or SEND functions, the ECB is posted immediately and there is no waiting for the remote host to respond.
FOIP= [0|*foreignipaddr*]	Identifies the foreign IP address that TCP/IP FOR VSE is to connect with or transmit to. This keyword must be used in an active open call. The default is 0. If you code a value for a PASSIVE connection (ACTIVE=NO), then that value is used as a mask to limit incoming connections to a specific host.
FOPORT= [0|*foreignportnum*]	Identifies the foreign port number that TCP/IP FOR VSE is to connect with or transmit to. This keyword must be used in an active open call. The default is 0.
ID=[00|*nn*]	Identifies a two-digit system ID assigned to a TCP/IP FOR VSE partition. When you start a TCP/IP FOR VSE partition, you can use the parameter string on the EXEC statement to specify a system ID. When you use the ID keyword on the SOCKET macro, all SOCKET requests are transmitted to the TCP/IP FOR VSE partition that matches the ID you specify. The default is 00.
LOPORT= [0|*localportnum*]	Identifies the local port number to be used when establishing a TCP connection. If you specify 0, which is the default, TCP/IP FOR VSE assigns an available local port number. Assignment begins at 4096 and proceeds upward. Port numbers should not be reused with the same IP addresses for at least several minutes. This ensures that stale datagrams from a closed connection do not pollute a later connection. In general, clients (ACTIVE=YES) should specify 0 and servers (ACTIVE=NO) should specify the number of their well-known port.

Keyword	Description
MF=[E|L]	Sets the macro format. Specify L to create the global constant area and E to execute all other functions. The default is E.
MFG=*soparea*	Identifies the address of the SOCKET parameter area, which is used to pass the addresses of the request parameters to TCP/IP FOR VSE. To make your program fully reentrant, you can define the area in which you want the parameter list to be built. The length of this area is 72 bytes. It can be reserved by using the following code: `SOPAREA   DS   (SOPLEN)X   PARAMETER AREA` There is no default.
NEGOT=[YES|NO]	Indicates whether TELNET connections are permitted to negotiate session attributes. When you specify YES, the negotiation is permitted. The default is YES.
TIMED=[YES|NO]	Indicates whether this socket operation is subject to the TIMEOUT= value. When you specify YES, monitoring is activated. If the time specified in the TIMEOUT keyword elapses without data being received from the remote host, the receive is posted as complete and control returns to the calling program. This is especially useful for limiting RECEIVE and Passive OPEN (listen) socket requests, which can require an indefinite amount of time to complete. The default is NO.
TIMEOUT= [36000|*timevalue*]	Specifies the time allowed for the stack to respond to a socket request. The value is the number of 300th-second intervals. A value of 18000 specifies 60 seconds. The default is 36000, or 2 minutes.
USESYS= [YES|NO]	Indicates whether the application's JCL can use the // OPTION SYSPARM=*xx* statement to identify the TCP/IP FOR VSE partition to connect to. If YES, the statement is permitted. If the SET_SYSID command has been issued, USESYS=YES overrides it. The default is NO.
WAIT=[YES|NO]	Indicates whether the SOCKET macro should generate an appropriate WAIT for the completion of the operation. If YES, a WAIT is generated. If NO, the SOCKET macro completes as soon as the request has been queued to the stack partition. In this case, you must code a separate WAIT to ensure that the operation completes. The default is NO.

Global Constant Area

The Global Constant area is required for all SOCKET macro operations within an application. This area contains constant information required by all SOCKET operations for communication with the TCP/IP FOR VSE partition. The following SOCKET macro creates the global constant area and sets the ID of the desired TCP/IP FOR VSE partition:

```
label    SOCKET   ID=00,MF=L
```

Connecting to TCP/IP FOR VSE

TCP/IP FOR VSE can execute in multiple partitions at one time. To use the SOCKET macro, you must identify a specific TCP/IP FOR VSE stack ID in the global constant area. To change the partition you are communicating with, you must modify this area. The following statement shows how to change the TCP/IP FOR VSE stack identifier to which your application connects.

```
label    SOCKET   SET_SYSID,newid
```

In this statement, *newid* refers to a two-byte area containing the new stack ID for SOCKET operations. All SOCKET operations that follow are directed to the specified stack. For example, using

```
SOCKET   SET_SYSID,=C'01'
```

connects to the partition running stack ID 01. (Of course, instead of using `=C'01'`, you can specify a two-byte field name that contains the value. No length checking is done by the macro.) Specifying USESYS=YES on a SOCKET open request overrides the SET_SYSID specification if a // OPTION SYSPARM='01' statement is used in the JCL of the partition running the application.

Return Codes

Register 15 contains the return codes. You should check the return code after each SOCKET call to determine whether your request processed successfully. The codes are defined with equates in the SOCKET macro.

You must determine whether register 15 contains zero or a non-zero value. If zero is returned, this means that you have been able to use the interface. It does not mean that your request was successful. You must wait for the stack to return a response by issuing a WAIT against the SRECB field. After the SRECB is posted, check the return code in the SRCODE field that is contained in the SRBLOK response area. This value indicates whether the request succeeded. The SRCODE values are explained in the next section.

If register 15 contains a non-zero value, it means that your socket request could not be scheduled. This may indicate invalid or inconsistent parameters, logic errors, or that the requested stack partition is not available. For some register 15 return codes, you must also examine the value in register 0. This value may indicate the specific problem.

The register 15, register 0 code combinations and their meanings are described in the following table.

Reg 15 Value	Reg 0 Value	Meaning
0		The request is successfully scheduled. After the ECB= field is posted, SRCODE contains the completion status.
4	*x*	Failure to obtain storage. Register 0 contains the return code from the storage call (GETVIS).
8		The requested stack is unavailable. See the register 0 value (1, 2, or 3).
	1	SCBLOK was not found. This indicates that TCP/IP FOR VSE has not been initialized since the last IPL.
	2	The stack is not running or is shutting down.
	3	The SQBLOK eyecatcher is wrong. This indicates corruption in one of the key TCP/IP control blocks. You may need to cycle the stack partition to recover.
9	2	The SCBLOK eyecatcher is wrong. This may indicate a storage corruption problem with a critical TCP/IP control block. Cycling one of the TCP/IP FOR VSE stacks may correct the problem.
10	3	The SQBLOK pointer in SCBLOK is zero. This means that the socket request could not find an active stack with a matching SYSID value.
11	4	The stack is shutting down.
12		The DESCRIPTOR field is zero (except for OPEN)
14	6	The SOQUEUE field in SOWORK is zero. This may indicate that the socket call was issued (1) before the OPEN completed, (2) after a CLOSE was issued, or (3) with an invalid DESCRIPTOR value.
15	1	The SQSOCKET field in SQBLOK is zero. This generally means that the stack is initializing.
16	x	Cannot obtain storage. Register 0 contains the GETVIS return code.
17	x	Storage release error. Register 0 contains the FREEVIS or IPSTOR return code.

Reg 15 Value	Reg 0 Value	Meaning
24	x	The SUBSID call failed. Register 0 contains the SUBSID return code.
28		The SOPARM parameter list is invalid. See the register 0 value (1, 2, 3, or 4).
	1	The eyecatcher (“SO”) is invalid.
	2	The version is invalid (not 1).
	3	The release is invalid (not 2, 3, or 4).
	4	The descriptor field is zero (all calls).

SRBLOK DSECT

Every SOCKET request generates a response that is returned in a 56-byte area that you provide. The first fullword of this area serves as an ECB that allows the application program to wait for the request to complete.

The format of the SRBLOK can be found in the SRBLOK macro and is subject to change from release to release. Typically, new fields are added to the end of the block for compatibility.

The DSECT fields are described in the following table.

Field	Description
SRECB	This fullword is the primary ECB for the request operation. When the SOCKET request completes, the ECB is posted. The SOCKET macro also provides a mechanism for a second ECB to be posted along with the SRECB. The second ECB is not contained within the response area.
SRLOPORT	After an open request is complete, this halfword contains the local port number that is assigned to the connection.
SRFOPORT	After an open request is complete, this halfword contains the foreign port number that is assigned to the connection.
SRFOIP	After an open request is complete, this fullword contains the foreign IP address of the connecting system.
SRCOUNT	After a receive request completes, this halfword field contains the number of bytes that were transferred.
SRFLAGS	These flags alert the application program to a specific action or operation that has occurred. They are discussed in more detail where necessary.

<table>
<tr><th>Field</th><th colspan="2">Description</th></tr>
<tr><td rowspan="16">SRCODE</td><td colspan="2">This field contains the return code for each SOCKET request. The codes and their meanings are listed below. The hexadecimal equivalent of each code is in parentheses.</td></tr>
<tr><td><b>Code</b></td><td><b>Meaning</b></td></tr>
<tr><td>0 (0)</td><td>The socket request completed normally.</td></tr>
<tr><td>4 (4)</td><td>Connection not found (does not apply to OPEN). The connection has terminated and the related control blocks are gone. For a RECEIVE, this means that the remote host issue issued a FIN and all data on the connection has been retrieved.</td></tr>
<tr><td>8 (8)</td><td>Connection has been RESET.</td></tr>
<tr><td>12 (0C)</td><td>A timed OPEN or RECEIVE reached the specified time-out value without completing.</td></tr>
<tr><td>16 (10)</td><td>The descriptor field contains nulls. This request should have failed to schedule. This may indicate a wrong use of the SOCKET macro or other API.</td></tr>
<tr><td>20 (14)</td><td>Duplicate use of an ECB. This request shares an ECB and SRBLOK with another running socket request.</td></tr>
<tr><td>24 (18)</td><td>General failure during an OPEN.</td></tr>
<tr><td>28 (1C)</td><td>The TCP/IP FOR VSE stack is shutting down.</td></tr>
<tr><td>32 (20)</td><td>For a STATUS call, the return area is too short to return the STATBLK.</td></tr>
<tr><td>36 (24)</td><td>A SEND was issued after a CLOSE or a FIN.</td></tr>
<tr><td>40 (28)</td><td>The connection was not established when a SEND or a RECEIVE was issued.</td></tr>
<tr><td>44 (2C)</td><td>A CLOSE was issued, but there was still data waiting to be received (by a RECEIVE). The queued data is discarded.</td></tr>
<tr><td>48 (30)</td><td>There was a storage shortage in the TCP/IP FOR VSE partition.</td></tr>
<tr><td>52 (34)</td><td>The operation was terminated because an ABORT was issued.</td></tr>
<tr><td>SRTERMTY</td><td colspan="2">Contains the agreed-on terminal type after a TELNET open request is negotiated.</td></tr>
</table>

Opening a Connection

Before you can communicate, you need to open a connection. A connection can be active or passive. An active connection seeks out the specified partner and actively negotiates the connection. A passive connection takes no action and simply waits to receive a connection request from the remote end.

Active Connect

The following example opens an active connection.

```
label    SOCKET   OPEN,TCP,                                *
               ACTIVE=YES,                                 *
               FOIP=IPADDR,                                *
               FOPORT=65,                                  *
               DESC=SOCKDESC,                              *
               ECB=RESULTS
         LTR      R15,R15            Test the return
         BNZ      ERROR              Failure, then...
         WAIT     RESULTS
         USING    SRBLOK,RESULTS
         CLI      SRCODE,0
         BNE      ERROR
```

In this example, TCP/IP FOR VSE is asked to establish a connection with a foreign system whose IP address is held in the fullword IPADDR and whose foreign port number is 65. After the connection is established, the fullword SOCKDESC must be passed to all subsequent SOCKET calls for this connection. This is the only call that is required to establish a complete connection with the foreign system.

After the SRECB contained in the result area is posted, the connection is ready for send and receive activity. An active connection request must complete within a timeout period. The timeout keyword is omitted, so the timeout value defaults to two minutes. If the connection is not complete within two minutes, the SRECB is posted and an error condition is set in the SRCODE field of the result area.

Passive Connect

The following example opens a passive connection.

```
label    SOCKET   OPEN,TCP,                                *
               PASSIVE=YES,                                *
               LOPORT=65,                                  *
               DESC=SOCKDESC,                              *
               ECB=RESULTS
         LTR      R15,R15                 Test the return
         BNZ      ERROR                   Failure, then...
         WAIT     RESULTS
         USING    SRBLOK,RESULTS
         CLI      SRCODE,0
         BNE      ERROR
```

In this example, the application program directs TCP/IP FOR VSE to listen for a connection request to arrive at port 65. When a connection request arrives, the connection is completed, and the request is posted as complete.

Notice that the foreign port and IP address are not specified. This allows any remote user's connection request to be accepted.

After the connection is established, further connection requests for this local port number are rejected unless there are other server programs waiting on this same local port number. In this example, the listen connection established waits forever for a connection request. If this is not desirable, TIMEOUT= may be used to force a completion even if no connection request is received.

Receiving Data

After a connection completes, the next logical step is to receive or send data. The following example uses the SOCKET macro to receive data from a foreign host.

```
label    SOCKET   RECEIVE,TCP,                              *
               DATA=(INBUFF,INLEN),                         *
               DESC=SOCKDESC,                               *
               ECB=RESULTS
         LTR      R15,R15               Test the return
         BNZ      ERROR                 Failure, then...
         WAIT     RESULTS
         USING    SRBLOK,RESULTS
         CLI      SRCODE,0
         BNE      ERROR
         MVC      INGOT(2),SRCOUNT
           .
           .
           .
INBUFF   DC       A(BUFFER)
INLEN    DC       F'4096'
INGOT    DC       H'0'
BUFFER   DS       CL4096
```

In this example, the SOCKET macro queues a request to receive information from the foreign host. You can have many receive requests queued on the same connection. Each request must provide its own separate result area (ECB=), but it must refer to the same descriptor word (DESC=). Each request is processed in the order it was queued, and data is passed to the application as it arrives. The maximum information that can be received in one request is 65,535 bytes. Because it is unusual for 64K of information to arrive at one time, such a specification would waste memory. Generally, information is received in pieces no larger than the MTU size of the link. The amount of data received is returned in the SRCOUNT field.

It is important to note that TCP is a stream-oriented protocol, and iteration is necessary when receiving a stream of data. This is a significant difference from most VSE I/O operations and requests that are record oriented or block oriented. A disk or tape VSE I/O operation issues a read request, waits on an ECB for completion, checks for errors, and then has the entire record or block available for processing. A TCP stream application may send 4096 bytes of data to a receiving application. The receiving application may get the full 4096 bytes in one receive, or it may get 2000 bytes, then 1000 bytes, and then 1096 bytes in three separate receive requests. TCP guarantees that it delivers bytes in sequence, but it does not guarantee that it delivers bytes in the same groups in which they are sent.

To handle a TCP data stream, the receiving application must continue to issue receive requests to the input stream until the agreed upon data structure indicates that all data is received. For example, TELNET and FTP protocols use a carriage return/line feed indicator in the data stream to indicate the end of a command or record.

The receiving application continues to loop through receive requests and to add data to a buffer until the carriage return/line feed indicators are detected in the data stream. At that point, the receiving application knows that it has received a complete command or record that can be processed. You must carefully preserve left-over data, as this is the first part of the next record.

Sending Data

The following example uses the SOCKET macro to send data across a connection.

```
         MVC      OUTLEN(4),=F'45'
         SOCKET   SEND,TCP,                                *
               DATA=(OUTBUFF,OUTLEN),                      *
               DESC=SOCKDESC,                              *
               ECB=RESULTS
         LTR      R15,R15                 Test the return
         BNZ      ERROR                   Failure, then...
         WAIT     RESULTS
         USING    SRBLOK,RESULTS
         CLI      SRCODE,0
         BNE      ERROR

           .
           .
           .

OUTBUFF  DC       A(BUFFER)
OUTLEN   DC       F'4096'
BUFFER   DS       CL4096
```

In this example, 45 bytes of data are sent across the connection to the foreign system. The SRECB is posted in the result area when the data has arrived at the desired location. The send request must refer to the descriptor word (DESC=) that was created during OPEN processing.

Send requests are processed in the order in which they are issued. The maximum buffer size that can be used is 65,535 bytes. Regardless of size, each send request accepted by the TCP/IP FOR VSE partition is broken into different size pieces for actual transmission.

To speed-up data transmission, you may not want to wait for each SEND request to be acknowledged by the remote host before proceeding. In this case, you would fill a large buffer, issue a SEND (without waiting), refill the buffer, and *then* wait for the first to complete. For maximum efficiency, use two ECB= areas and alternate them. This ensures that there is always data ready to be sent.

Status

Sometimes you need to obtain information about a connection or operation prior to its completion. To do this, use the STATUS call.

```
         SOCKET   OPEN,TCP,                                  *
               DESC=DATADESC,                                *
               PASSIVE=YES,                                  *
               ECB=DATAECB
         LTR      R15,R15
         BNZ      PASVF421
*
*        Wait for 1 second
*
PASVLIST XC       PASVTECB(4),PASVTECB
         SETIME   1,PASVTECB
         WAIT     PASVTECB
*
*        Let's do a status call
*
         LA       R1,PASVCCBL
         ST       R1,PASVADDR
         MVC      PASVLEN(4),=A(STBLOKLN)
         SOCKET   STATUS,TCP,                                *
               DESC=DATADESC,                                *
               DATA=(PASVADDR,PASVLEN),                      *
               ECB=PASVRECB
         LTR      R15,R15
         BNZ      PASVF421
         LA       R1,PASVRECB
         WAIT     (1)
         USING    SRBLOK,PASVRECB
         CLI      SRCODE,SRSTGOOD
         BNE      ERROR
*
*        Check that the connection...
*
         USING    STATBLK,PASVCCBL
         LA       R5,60
TRYAGIN  DS       0H
         CLI      STSTATE,STSTLIST
         BE       GOTPORT
         LA       R3,TIMEECB
         SETIME   2,TIMEECB
         WAIT     TIMEECB
         BCT      R5,TRYAGIN
```

In this example, a status call is used to determine the local port number that was assigned and the local IP address of TCP/IP FOR VSE. The local port number could have been determined when the first foreign client connected to the VSE server application. When the VSE server application issues a passive OPEN, the application program can wait until a foreign client connects and the OPEN is complete. At this point, the result area contains the local port number (SRLOPORT), the foreign port number (SRFOPORT), and the foreign IP address (SRFOIP).

For some server applications, for example, FTP, you need to know the local port number or the local IP address before clients are connected. Also, the local IP address is not returned in the result area and must be obtained by a status call.

In the example above, we issue a status request before any foreign clients have connected to our VSE server application, and we save the local port number and local VSE IP address after the connection is in a listening state.

An example of an application that uses a status call is the FTP protocol. It uses a control connection and a data connection on different ports. The control connection is usually on standard port 21 and foreign clients connect to it there. When a user PUTs or GETs a file, however, a separate data connection is opened and a STATUS call is issued to determine the port number assigned to the data connection. After the port number is determined, the FTP port command is sent on the control connection to the foreign client so that it knows the port number assigned to the data connection. The foreign client then opens its side of the data connection, and FTP commands can continue to be sent and received on the control connection (theoretically) even while a large file is being transferred on the separate data connection.

You can use the STATBLK macro to map information returned from a STATUS call. The STATBLK macro completely replaces the CCBLOK macro that was distributed with prior releases of TCP/IP FOR VSE. For compatibility, adding the “CC=YES” parameter to the STATBLK macro call equates the old CCBLOK field names to their STATBLK counterparts.

STATBLK DSECT

The STATBLK DSECT maps the connection control block from the TCP/IP FOR VSE partition to the caller’s local storage. It contains many fields that are intended for internal use and are not helpful in this situation. The fields listed in the following table are useful.

Field	Description
STSTATE	Connection status
STLOIP	Local IP address
STLOPORT	Local port number

Field	Description
STFOIP	Foreign IP address
STFOPORT	Foreign port number

Close Connection

After you finish using a connection, you should close it. The following example shows a close operation:

```
         SOCKET   CLOSE,TCP,                                *
               DESC=DATADESC,                               *
               ECB=DATAECB
         LTR      R15,R15
         BNZ      ERROR
*
*        Wait for completion
*
         WAIT     (1)
         LA       R1,DATAECB
         USING    SRBLOK,DATAECB
         CLI      SRCODE,0
         BNE      ERROR
```

This is a simple operation that only requires a descriptor and a result area. Although the close operation is queued behind any outstanding send operations, it is good practice to allow previously queued requests to complete before you issue the close.

Control Connection

We have discussed how you can use the SOCKET macro for TCP connections. You can also use it to communicate directly with TCP/IP FOR VSE, which maintains a control manager in its partition. Using the SOCKET facility, you can send requests to the Control Manager and obtain the results.

Resolving Symbolic Names

TCP/IP FOR VSE contains a local name server (see DEFINE NAME in the *TCP/IP FOR VSE Command Reference*) and a domain name client (see the SET DNS1 command). This enables you to assign symbolic names to actual TCP/IP addresses. Symbolic names are easier to remember than IP addresses, and you can reassign a name to a different address.

To request the IP address associated with a symbolic name, send the following command to the control manager:

```
GETHOSTBYNAME name
```

The control manager looks for the name in the local name server and then in the domain name client, and it returns this block of data:

```
IPADDR   DS    F             IP Address in binary
IPADDRC  DS    CL15          IP Address in display format
```

You can request an IP address or host name using the commands described in the following table. Text input should contain the command and operand and should be delimited with a newline (x15). The command and the operand data should be separated by one or more blanks.

Command	**Input**	**Output**
GETHOSTBYADDR	IPv4 address in EBCDIC text characters	Blank-delimited domain name
GETHOSTBYNAME	Domain name in EBCDIC text characters	19 bytes of data containing: • 4-byte binary IPv4 address • 15-byte text IPv4 domain address (*nnn.nnn.nnn.nnn*)
GETHOSTNAME	None	Blank-delimited text name of the local VSE system

Command	Input	Output
GETHOSTID	None	19 bytes of data containing: • 4-byte binary IPv4 address • 15-byte text IPv4 address of the local VSE system

Like all SOCKET operations, the first order of business is to open a connection. This is shown in the following example.

```
HOSTNAME SOCKET   OPEN,CONTROL,         Open the connection *
               DESC=CONTDESC,           Descriptor          *
               ECB=CONTECB              ECB Address
         LTR      R15,R15               Test the return code
         BNZ      ERROR                 Bad, then error
         LA       R1,CONTECB            Address the ECB
         WAIT     (1)
*
* * Check the results
*
         USING    SRBLOK,R1
         CLI      SRCODE,SROPGOOD       Was the open good?
         BNE      ERROR                 If no, exit with error
         DROP     R1
```

At this point, the connection is established, and the control manager is waiting for the application to transmit a command for execution. No IP address or port information is specified in the open call. This is because no connection to a foreign host is necessary. The only connection required is with the TCP/IP FOR VSE partition.

The next step is to code the command to be passed to the control manager. This is shown in the following example.

```
* *  Set up the command
         MVC   CONTBUFF(CMDL),CMD       Move command to buffer
         LA    R3,CMDL                  Command length (w/NL)

...

         CMD   DC     C'GETHOSTBYNAME CSI-INTERNATIONAL.COM'
         DC    X'15'                    End-of-command (NL)
         CMDL  EQU    *-CMD             Length of entire cmd
```

Note that an end-of-command character (an EBCDIC newline) is included.

Next, send the command to the control manager. You do this exactly as if the data is destined for a foreign host.

```
*
* * Send the command
         ST       R3,CONTLEN               Save the length
         LA       R4,CONTBUFF              Address the buffer
         ST       R4,CONTADDR              Save the address
         SOCKET   SEND,CONTROL,            Send the command    *
               DATA=(CONTADDR,CONTLEN),    Identify the data   *
               DESC=CONTDESC,              Descriptor          *
               ECB=CONTECB                 ECB Address
         LTR      R15,R15                  Test the return code
         BNZ      ERROR                    Bad, then error
         LA       R1,CONTECB               Address the ECB
         WAIT     (1)
*
* * Check the response
         USING    SRBLOK,R3
         CLI      SRCODE,SRSEGOOD          Was the send good?
         BNE      ERROR                    No, then exit w/error
         DROP     R1
```

At this point, the control manager has received the command and is executing it. The results of the command are returned as data.

To obtain the results, issue a receive request. Although you can issue the receive before the send (leaving it queued), the following example shows it issued in sequence:

```
*
* * Let's get the response
         MVC      CONTLEN(4),=F'100'       Set the length
         LA       R4,CONTBUFF              Address the buffer
         ST       R4,CONTADDR              Save the address
         SOCKET   RECEIVE,CONTROL,         Receive the response*
               DATA=(CONTADDR,CONTLEN),    Identify the data   *
               DESC=CONTDESC,              Descriptor          *
               ECB=CONTECB                 ECB Address
         LTR      R15,R15                  Test the return code
         BNZ      ERROR                    Bad, then error
         LA       R1,CONTECB               Address the ECB
         WAIT     (1)
*
* * Check the results
*
         USING    SRBLOK,R1
         CLI      SRCODE,SRREGOOD          Was the receive good?
         BNE      ERROR                    No, then exit w/error
         DROP     R1
```

Now that you have received the results of the command execution, you can extract them from the reply.

This is shown in the following example:

```
*
* * Copy the data from the buffer
         MVC     IPADDR(4),CONTBUFF        Copy the IP Address
         MVC     IPADDRC(15),CONTBUFF+4    Copy the Char IP Addrs
```

You must now issue a close operation to terminate the connection with the control manager. If you leave the connection open, you can make additional requests over the same connection. You can close the connection as shown in the following example:

```
*
* * Close the control
         SOCKET   CLOSE,CONTROL,         Close the connection *
               DESC=CONTDESC,            Descriptor           *
               ECB=CONTECB               ECB Address
         LTR      R15,R15                Test the return code
         BNZ      ERROR                  Bad, then OK, move on
         LA       R1,CONTECB             Address the ECB
         WAIT     (1)
```

Sample Programs

The source code for the following sample programs can be downloaded from CSI International's website, www.csi-international.com.

Program	Description
Assembler Server SAMSERVR	This sample server program is written in assembler language and performs the following tasks: • Initializes and attaches three VSE subtasks • Opens a passive TCP connection and waits for a command from a client application • Dispatches one of the attached VSE subtasks to process the command and sends a reply to the client application that sent the command
Assembler Client SAMCLINT	This sample client program is written in assembler language and performs the following tasks: • Opens an active TCP connection to the server • Reads a command from SYSIPT • Sends the command to the server • Issues a receive for the reply and displays the reply on SYSLST
Microsoft Visual Basic Client for Windows®	This sample client performs the following tasks: • Opens an active TCP connection to the SAMSERVR on VSE • Displays a Windows GUI interface for sending commands to VSE • Sends a command from the Windows GUI to VSE • Displays the VSE reply in a list view format on the Windows GUI

2

BSD Socket Interface

Overview

The *de facto* standard application programming interface (API) for TCP/IP applications is the BSD socket interface. Although this API was developed for the BSD Unix operating system in the early 1980s, it has been implemented on a wide variety of non-Unix systems. For TCP/IP FOR VSE, the BSD socket interface is simply referred to as the BSD interface.

There was no official RFC standard for the BSD Interface for IPv4, but RFC3493, "Basic Socket Interface Extensions for IPv6," now contains a standard definition for IPv6 and addresses compatibility with the IPv4 *de facto* standard. VSE's implementation of the interface does not provide all of the variations of the defined functions, but it attempts to parallel the z/VM and z/OS interfaces as needed.

Languages

The BSD interface can be used in C, Assembler (often referred to as Basic Assembler Language, or BAL), COBOL, and other languages that support standard call/save linkage conventions. It is also valid for use with either batch or CICS programs. The interface determines at run time whether the program is under CICS control and, if it is, uses CICS services. The language environment (LE) is also dynamically detected for LE-conforming applications.

What is a Socket?

The BSD interface is the universally accepted standard for writing client and server socket applications that use the TCP/IP protocol. Applications using the BSD interface are commonly referred to as socket applications. Much information is available on how to create socket applications using the interface’s well-known functions. Before you begin using these functions to develop socket applications, you must understand the concept of a socket.

Definition

The term *socket* has been used in many ways, and it is important to understand the meaning within the context of TCP/IP FOR VSE’S BSD interface. A socket is a dynamically allocated control block that is assigned a unique socket number. An application passes the socket number to various functions (listen, accept, send, receive, close, etc.) to control a single TCP/IP connection. In essence, the socket number is a handle that allows the application to function independently of TCP/IP control block structures.

The socket number is dynamically associated with a specific port when the connection is established. You can use a socket to request information from TCP/IP, wait for an incoming connection request, or to send and receive data across the network. A unique socket number is required for each connection used by an application. When the application closes the connection, the socket number (and corresponding socket control block) is returned to the pool of available sockets.

Socket vs. Port

The term *socket* is sometimes confused with *port number*. In this discussion of the BSD interface, the two terms have completely different meanings. This difference may be easier to understand by describing each term in the context of a BSD server and a client application.

In a BSD server application, the socket function is used to allocate and assign a unique socket number. The bind function is then used to associate that socket number with a specific TCP/IP port. The BSD server would have multiple connections with remote clients using unique socket numbers, but it would have the same port number for all the client connections. The port number for a server is usually a constant between 1 and 65,535 that is known by the client applications wanting to connect to the server. By comparison, socket numbers are unique between 1 and 8,192 and are known only by the server application.

In a BSD client application, the socket function is also used to dynamically allocate and obtain a unique socket number. The connect( ) function is then called with a remote IP address and port number parameters. The client application must know or obtain the port number of the server application before it issues its connect request.

The local port number of a client application usually is assigned dynamically, and most client applications do not know or care what local port number is used. They simply use the dynamically assigned socket number and remote port number to communicate with the remote server application. The client really only needs to know the IP address and port number of the remote server application to connect to it.

Using Socket Functions

Socket functions allow application programs to communicate with the TCP/IP FOR VSE partition. You can use the socket functions for both server and client applications. The examples in this section show the order in which various socket functions are called so that you can see how the functions work in a BSD socket program. This section also includes information on connecting to a specific TCP/IP FOR VSE stack.

The examples below are not detailed but convey a general understanding of the concepts involved. Each function is described in a later section.

TCP Server

A typical TCP server application will take the following actions:

- Allocate a socket with a socket( ) call.
- Initialize the sockaddr structure only with the desired port number. The remainder of the IP address is cleared to zeros.
- Bind the allocated socket to a specific port number with a bind( ) call.
- Issue a listen( ) call to send the initial passive open request to the TCP/IP FOR VSE partition.
- Issue an accept( ) call so that the server waits (blocks) until a client connects. When a client application sends a connect request to the server port, the accept allocates a new socket for the client, reissues the passive open on the original socket number, and then returns control to the BSD server application with the new socket number.
- Process the client request by taking the following actions:
    1. Issue a receive( ) call to retrieve client data.
    2. Process the client data.
    3. Issue a send( ) call to respond to the client.
    4. Issue a close( ) call to close the new socket.
- Return to the accept( ) call to wait for another client request.
- Issue a close( ) call when the server is shut down to close the original socket and terminate the server application.

TCP Client

A typical TCP client application will take the following actions:

- Obtain a socket number with a socket( ) call.
- Connect to a remote host with a connect( ) call using the socket number obtained from the socket( ) call.

- Send data to the remote host with a send( ) call using the same socket number that was used for the connect( ) and obtained from the socket( ) call.
- Issue a receive( ) call to wait for a reply using the same socket number that was used for the send( ) request.
- Process the received data.
- Close the connection using the same socket number that was used for the connect( ) call.

Connecting to TCP/IP

By default, your program connects with the TCP/IP FOR VSE partition assigned to stack ID 00. The ID assignment is made in the parameter field of the TCP/IP FOR VSE EXEC statement. The default value for the ID is 00.

If you want to connect to a different TCP/IP FOR VSE partition for testing or other purposes, you can use the setsysid( ) function to change the stack ID. This function is described in the next section.

Another way to specify a stack ID is to include the following // OPTION statement in your program's JCL:

```
// OPTION SYSPARM='01'
```

In this example, 01 is the two-digit ID of the desired stack.

You can override the default ID and the one defined in the // OPTION SYSPARM statement by specifying a stack ID in a custom options phase ($SOCKOPT). You can create this phase by modifying default settings. See "Appendix A: $SOCKOPT Options Phase," page 196, for details on setting options in a custom options phase.

Function Descriptions

This section lists and describes the socket functions and their return codes. The entry point for Assembler, COBOL and other languages is also listed for each function. For information on the *errno* variable, see the section “Error Handling” on page 58.

abort( )

The abort( ) function immediately terminates the connection with a reset (RST). Any outstanding send or receive requests are also terminated and posted complete with a negative result. After the abort( ) is completed, the connection is closed and the socket is available to be reused. The syntax is as follows:

```
int abort(int);
rc = abort(socket);
```

The variables are described in the following table:

Variable	Description
rc	The return code. A 0 indicates that the abort completed successfully. A -1 indicates an error, and *errno* contains the reason.
socket	The socket number to be aborted.

For Assembler, COBOL and other languages, call the IPNRABRT entry point.

accept( )

The accept( ) function accepts a connect request from a remote client and returns the socket number to be used for the session. The syntax is as follows:

```
int accept(int,struct sockaddr *,int *);
rc = accept(listen,&sockaddr,&length);
```

The variables are described in the following table.

Variable	Description
rc	The socket number assigned to the accepted session is returned. A value of -1 indicates that an error has occurred, and the variable *errno* contains additional information.
listen	The socket number used by a listen( ) call.

Variable	Description
sockaddr	Points to a sockaddr structure or NULL. If a pointer is provided, it contains the address of the accepted host.
length	Points to an integer that contains the length of the sockaddr structure. This value is required if sockaddr is specified. The system replaces the original contents of *length* with the sockaddr length.

Usage Notes:

- There is no correspondence between the listen socket number and the returned session socket number.
- Control is not returned to your program until a session request is received on the listen socket. If you do not want to be suspended, use the select( ) function.
- The UDP protocol does not recognize the accept( ) function.
- You must accept( ) sessions with all requesters. You can, however, close undesirable connections immediately.

For Assembler, COBOL and other languages, call the IPNRACCP entry point.

bind( )

The bind( ) function assigns an IP address or port number to a socket.

```
int bind(int,struct sockaddr *,int);
rc = bind(socket,&sockaddr,length);
```

The variables are described in the following table:

Variable	Description
rc	The return code. A 0 indicates success. A -1 indicates that an error has occurred, and the variable *errno* has more information.
socket	The socket number to be labeled.
sockaddr	Points to a sockaddr structure that contains a port and/or IP address.
length	The length of the sockaddr structure.

Usage Notes:

- If you specify port 0 in a bind request, the system assigns an available port.

- This function is affected by the $OPTBNDX option in $SOCKOPT. See “Appendix A: $SOCKOPT Options Phase,” page 196, for details on setting options in a custom options phase.

For Assembler, COBOL and other languages, call the IPNRBIND entry point.

close( )

The close( ) function closes a connection and releases allocated resources. The socket is returned to the system.

```
int close(int);
rc = close(socket);
```

The variables are described in the following table:

Variable	Description
rc	The return code. A 0 indicates that the close completed successfully. A -1 indicates that an error has occurred, and the variable *errno* contains more information.
socket	The socket number to be closed.

Usage Notes:

- If you are reading from a socket and you close it while data is pending, the connection is reset rather than closed. This alerts the other host to an error condition. This situation occurs only with TCP connections and not with UDP connections.
- This function is affected by the $OPTCNFW option in $SOCKOPT. See “Appendix A: $SOCKOPT Options Phase,” page 196, for details on setting options in a custom options phase.

For Assembler, COBOL and other languages, call the IPNRCLOS entry point.

connect( )

The connect( ) function establishes a connection with a remote client or server. Status information is not returned until the connection completes.

```
int connect(int,struct sockaddr *,int);
rc = connect(socket,&sockaddr,length);
```

The variables are described in the following table:

Variable	Description
rc	The return code. A 0 indicates that the connect completed successfully. A -1 indicates that an error has occurred, and the variable *errno* has more information.
socket	The socket number to be used for the connection.
sockaddr	Points to a sockaddr structure that contains the address of the remote client or server you want to connect to. This value can be NULL if the bind( ) function has already supplied the information.
length	The length of the sockaddr structure. This value can be NULL if sockaddr is NULL.

For Assembler, COBOL and other languages, call the IPNRCONN entry point.

getclientid( )

The getclientid( ) function returns the identifier by which the calling application is known to the TCP/IP FOR VSE partition. The clientid that is returned is used in the givesocket( ) and takesocket( ) functions.

```
int getclientid(int domain,struct clientid *,int);
rc = getclientid(domain,&clientid);
```

The variables are described in the following table:

Variable	Description
rc	The return code. A 0 indicates that the function completed successfully. A -1 indicates that an error has occurred, and the variable *errno* has more information.
domain	The requested address domain.
clientid	Pointer to a clientid structure that is to contain the identifier.

For Assembler, COBOL and other languages, call the IPNRGETC entry point.

gethostbyaddr( )

The gethostbyaddr( ) function takes an IP address and returns the symbolic name associated with it.

```
unsigned long gethostbyaddr(char *,int,int);
rc = gethostbyaddr(ipaddr,ipaddr_len,domain);
```

The variables are described in the following table:

Variable	Description
rc	The symbolic name associated with an IP address, or a -1 if the address is not found.
ipaddr	The pointer to the IP address.
ipaddr_len	The length of *ipaddr* in bytes.
domain	The pointer to the address domain support (AF_INET).

For Assembler, COBOL and other languages, call the IPNRGHBA entry point.

gethostbyname( )

The gethostbyname( ) function looks up a symbolic name and returns its IP (network) address.

```
unsigned long gethostbyname(char *);
rc = gethostbyname(&string);
```

The variables are described in the following table:

Variable	Description
rc	The IP (network) address associated with the name or a -1 if the name is not found.
string	A zero-delimited text string.

For Assembler, COBOL and other languages, call the IPNRGETN entry point.

gethostid( )

The gethostid( ) function returns the IP address currently used by the local TCP/IP FOR VSE host.

```
unsigned long gethostid( );
ipaddr = gethostid( );
```

The address returned is the one established by the SET IPADDR command. This is also true of multi-homed hosts.

For Assembler, COBOL and other languages, call the IPNRGETH entry point.

gethostname( )

The gethostname( ) function acquires the local host's assigned name, if one exists.

```
int gethostname(char *, int);
rc = gethostname(&buffer,length);
```

The variables are described in the following table:

Variable	Description
rc	The return code. A 0 indicates that the local host name has been returned. A -1 indicates that an error has occurred or that no name is assigned to the local host.
buffer	Points to a buffer in which the zero-delimited name is returned. Under TCP/IP FOR VSE, this buffer must be at least 17 bytes long.
length	The length of the buffer.

The function determines the local host's name by locating the local host's address as determined by the SET IPADDR command. The function then scans the name table that was created with command DEFINE NAME,ID=*xxx*,IPADDR=*xxx* until it finds a match for the local host's address. If it finds a match, it returns the associated name entry.

For Assembler, COBOL and other languages, call the IPNRGETA entry point.

getpeername( )

The getpeername( ) function returns the foreign IP address and port of the peer connected to a socket.

```
int getpeername(int, struct sockaddr *,int);
rc = getpeername(socket,&sockaddr,length);
```

The variables are described in the following table:

Variable	Description
rc	The return code. A 0 indicates that the function completed successfully. A -1 indicates that an error has occurred, and the variable *errno* has more information.
socket	The socket number.
sockaddr	Points to a sockaddr structure that is to contain the foreign IP address and port.
length	The sockaddr structure's length in bytes.

For Assembler, COBOL and other languages, call the IPNRGETP entry point.

getsockname( )

The getsockname( ) function returns the local IP address and port of a socket.

```
int getsockname(int,struct sockaddr *,int);
rc = getsockname(socket,&sockaddr,&length);
```

The variables are described in the following table:

Variable	Description
rc	The return code. A 0 indicates that the function completed successfully. A -1 indicates that an error has occurred, and the variable *errno* has more information.
socket	The socket number.
sockaddr	Points to the sockaddr structure that is to contain the local IP address and port.
length	Points to a variable containing the length of the sockaddr structure.

For Assembler, COBOL and other languages, call the IPNRGETS entry point.

getsockopt( )

The getsockopt( ) function updates the return area and length with the value of the requested option.

```
int getsockopt(int *,int *,int *,char *,int *)
rc = getsockopt(&p1,&p2,&p3,&p4,&p5)
```

The variables are described in the following table:

Variable	Description
rc	The return code. A 0 indicates that the function completed successfully. A -1 indicates that an error has occurred, and the variable *errno* has more information.
p1	Pointer to the socket number.
p2	Pointer to the code level for the related option.
p3	Pointer to the requested option.
p4	Pointer to the return area for the requested option's value.
p5	Pointer to the return area length.

For Assembler, COBOL and other languages, call the IPNRGETO entry point.

getversion( )

The getversion( ) function updates the return area with the version of TCP/IP FOR VSE that is currently active. The value returned is dependent on the passed return area length.

Eight-Byte Return Area. When the passed return area length is eight bytes, the returned area consists of the following parts:

- Four bytes containing the displayable characters "1.5E," "1.5F," "1.5G," "2.1*x*," or "2.2x," where *x* is the modification level (0 to 9).
- Four bytes containing the binary hexadecimal value that is internally associated with the four-byte character string that precedes it.

The product version values that may be output are as follows.

Character Format	Binary Format
`1.5E`	`X'00010501'`
`1.5F`	`X'00010502'`
`1.5G`	`X'00010503'`
`2.1x`	`X'000201xx'` (see text)
`2.2x`	`X'000202xx'` (see text)

For version 2 in binary format, *xx* is the modification level and ranges from 00 to 99, depending on the maintenance level active on the system.

Twenty-Byte Return Area. When the passed return area length is 20 bytes, the returned area contains the following:

- 8 bytes: character version (*vv.rr.mm*) of the TCP/IP stack
- 8 bytes: character version (*vv.rr.mm*) of the BSD interface (IPNRBSDC)
- 4 bytes: hexadecimal version (00*vvrrmm*) of the TCP/IP stack

```
int getversion(int *,int *)
rc = getversion(&p1,&p2)
```

The variables are described in the following table:

Variable	Description
rc	The return code. A 0 indicates that the function completed successfully. A -1 indicates that an error has occurred, and the variable *errno* contains more information.
p1	Pointer to the return area for the version.
p2	Pointer to the return area length. The length is a fullword containing either 8 or 20.

For Assembler, COBOL and other languages, call the IPNRVERS entry point.

givesocket( )

The givesocket( ) function makes the specified socket available to a takesocket( ) function issued by another program. Any socket can be given. Typically, givesocket( ) is used by a master program that obtains sockets by means of an accept( ) call and gives them to application programs that handle one socket at a time.

```
int givesocket(int,struct clientid *);
rc = givesocket(socket,&clientid);
```

The variables are described in the following table:

Variable	Description
rc	The return code. A 0 indicates that the function completed successfully. A -1 indicates that an error has occurred, and the variable *errno* contains more information.
socket	The socket number.
clientid	Pointer to a clientid structure specifying the program that is to receive the socket.

Givesocket/takesocket processing is affected by the $OPTGTSP option in $SOCKOPT. See "Appendix A: $SOCKOPT Options Phase," page 196, for details on setting options in a custom options phase.

For Assembler, COBOL and other languages, call the IPNRGIVE entry point.

listen( )

The listen( ) function instructs the system to monitor a port for connection requests from remote hosts.

```
int listen(int,int);
rc = listen(socket,backlog);
```

The variables are described in the following table.

Variable	Description
rc	The return code. A 0 indicates that a listen condition has been established and will continue for the requested number of events. A -1 indicates that an error has occurred, and the variable *errno* contains more information.
socket	The socket number to be associated with the listen. This socket contains the port number to be used.
backlog	The maximum number of incoming connections to be queued. See the Usage Notes for more information.

Usage Notes:

- This function is used with TCP requests; it does not apply to UDP requests.

- The value of *backlog* is overridden in the following cases:

  1. If QUEDMAX in $SOCKOPT equals 0, then *backlog* is ignored and the value is set by the PORTQUEUE command.
  2. If QUEDMAX in $SOCKOPT is greater than 0 but less than the value of *backlog*, then the value of QUEDMAX is used.

  By default, the QUEDMAX keyword is set to 0. See “Appendix A: $SOCKOPT Options Phase,” page 196, for details on setting options in a custom options phase.

- If you are going to use the PORTQUEUE command, then you must ensure that QUEDMAX=0 in $SOCKOPT to allow the command to control the queuing of inbound connection requests (SYNs). See “Port Queuing” in the “Performance” chapter of the *TCP/IP FOR VSE Installation Guide* for information on the PORTQUEUE command.

  When clients attempt to connect to a port in a listen state, the request can be rejected. Most clients then retry the connection attempt at some configured number of times for a specific time period before giving up.

  The TCP/IP DIAGNOSE CONNREJ command can be used to diagnose whether inbound connection requests are being rejected. Rejected requests may be retried and still be successful, but connection queuing can be implemented to avoid client connection rejections by a server on a specific port.

For Assembler, COBOL and other languages, call the IPNRLIST entry point.

receive( ), recv( )

The receive( ) function enables your program to receive data sent from a remote host. This function does not return control until the data is transferred.

```
int receive(int,char *,int,int);
result = receive(socket,&buffer,length,flags);
```

The variables are described in the following table.

Variable	Description
result	If the function completes successfully, this field contains the length of the received data. A value of -1 indicates that an error has occurred, and the variable *errno* contains more information.
socket	The socket number to be used for the receive( ) function.
buffer	Points to the buffer to be used for receiving the data.

Variable	Description
length	The length of the buffer.
flags	Provided for compatibility. Code zero as a placeholder.

Usage Notes:

- If you are using UDP and a datagram is too large to fit in the area provided, the excess bytes are discarded.
- After you issue the receive( ) function, control is not returned until the data is placed in your buffer and is available. To test for availability, use one of the select( ) functions.
- This function is affected by the $OPTXNBK option in $SOCKOPT. See "Appendix A: $SOCKOPT Options Phase," page 196, for details on setting options in a custom options phase.

For Assembler, COBOL, other languages, call IPNRRECV entry point.

recvfrom( )

The recvfrom( ) function is similar to the receive( ) function, but it includes additional parameters and is used only in UDP applications.

```
int recvfrom(int,char *,int,int,struct sockaddr *,length);
result = recvfrom(socket,&buffer,len,flags,&sockaddr,s_len);
```

The variables are described in the following table:

Variable	Description
result	If the function completes successfully, this field contains the length of the received data. A value of -1 indicates that an error has occurred, and the variable *errno* contains additional information.
socket	The socket number to be used for the receive( ) function.
buffer	Points to the buffer to be used for receiving the data.
len	The length of the buffer.
flags	Provided for compatibility. Code zero as a placeholder.
sockaddr	Points to a sockaddr structure that contains a port and/or IP address.
s_len	The length of the sockaddr structure.

For Assembler, COBOL and other languages, call the IPNRREFR entry point.

select( )

The select( ) function examines a subset of your program's sockets and indicates which ones are ready to be processed.

```
int select(int,fd_set *,fd_set *,fd_set *,struct timeval *);
tot = select(num,&read,&write,&exc,&time);
```

The variables are described in the following table:

Variable	Description
tot	After completion, the function returns the total number of sockets that are ready for processing. Only sockets flagged for examination are included. A value of -1 indicates that an error has occurred, and the variable *errno* contains more information.
num	Specifies the highest socket number to be examined. Socket numbers up to 8000 are valid, so setting a reasonable value reduces overhead.
read	Points to a bit string that indicates which sockets are to be examined for available data for reading. Each 1 bit causes the corresponding socket to be examined. If the socket has no read-eligible data, the bit is turned off. NULL indicates that the string is not used.
write	Points to a bit string that indicates which sockets are to be examined for availability for writing. Each 1 bit causes the corresponding socket to be examined. If the socket is not available for writing, the bit is turned off. NULL indicates that the string is not used.
exc	Points to a bit string that indicates which sockets are to be examined for pending exceptions. Each 1 bit causes the corresponding socket to be examined. If the socket has no pending exception, the bit is turned off. NULL indicates that the string is not used.
time	Points to a time structure that indicates how long the select( ) function should wait. If this value is NULL, select( ) does not return control until at least one socket is ready to process. Otherwise, control is returned when one or more sockets are ready or when the time interval expires, whichever is sooner.

For Assembler, COBOL and other languages, call the IPNRSELE entry point.

selectecb( )

The selectecb( ) function examines a subset of your program’s sockets and determines which ones are ready for processing. This function is similar to the select( ) function except that an ECB indicates that a socket is ready.

```
int selectecb(int,fd_set *,fd_set *,fd_set *,struct timeval *,int *);
tot = selectecb(num,&read,&write,&exc,&time,&ecb);
```

The variables are described in the following table.

Variable	Description
tot	At completion, *tot* contains the total number of sockets ready for processing. Only sockets flagged for examination are included. A value of -1 indicates that an error has occurred, and the variable *errno* contains additional information.
num	Specifies the highest socket number to be examined. Numbers up to 8000 are valid, so setting a reasonable value reduces overhead.
read	Points to a bit string that indicates which sockets are to be examined for read-eligible data. Each 1 bit causes the corresponding socket to be examined. If the socket has no read-eligible data, the bit is turned off. NULL indicates that the string is not used.
write	Points to a bit string that indicates which sockets are to be examined for write-eligible data. Each 1 bit causes the corresponding socket to be examined. If the socket has no write-eligible data, the bit is turned off. NULL indicates that the string is not used.
exc	Points to a bit string that indicates which sockets are to be examined for pending exceptions. Each 1 bit causes the corresponding socket to be examined. If the socket has no pending exception, the bit is turned off. NULL indicates that the string is not used.
time	Points to a time structure that indicates how long the selectecb( ) can remain outstanding. If this value is NULL, selectecb( ) monitors the sockets indefinitely or until one becomes ready. Otherwise, the ECB is posted when at least one socket is ready or when the time interval expires, whichever is sooner.
ecb	Points to an ECB that is posted when at least one socket is ready to process or when the optional time interval has expired.

Usage Notes:

- The selectecb( ) function always returns control immediately. There is no implied wait even when no sockets are ready for processing.
- When you issue selectecb( ), the ECB address is noted within each socket selected by the bit strings. The ECB is posted each time a selected socket is ready for processing. To determine which sockets are ready, use any select( ) function.
- If you reissue selectecb( ), all sockets specified are updated to point to a new ECB.
- After an ECB is assigned to a socket, it cannot be removed. The only way to work around this restriction is to assign a different (or dummy) ECB to the socket.
- If you specify a time value and it ends before a socket becomes ready, the pending selectecb( ) function terminates and the ECB is posted.

For Assembler, COBOL, and similar languages, call the IPNRSECB entry point.

selectex( )

The selectex( ) function examines a subset of your program’s sockets and determines which ones are ready for processing. The selectex( ) is similar to select( ), but it includes an ECB that can be used to terminate the implied wait.

```
int selectex(int,fd_set *,fd_set *,fd_set *,struct timeval *,int *);
tot = selectex(num,&read,&write,&exc,&time,&ecb);
```

The variables are described in the following table:

Variable	Description
tot	After completion, the selectex( ) function returns the total number of sockets that are ready for processing. Only sockets flagged for examination are included. If an error occurs, a -1 is returned, and variable *errno* contains additional information.
num	Specifies the highest socket number to be examined. Socket numbers up to 8000 are valid, so setting a reasonable value reduces overhead.
read	Points to a bit string that indicates which sockets are to be examined for read-eligible data. Each 1 bit causes the corresponding socket to be examined. If the socket has no read-eligible data, the bit is turned off. NULL indicates that the string is not used.

Variable	Description
write	Points to a bit string that indicates which sockets are to be examined for write-eligible data. Each 1 bit causes the corresponding socket to be examined. If the socket is not available for writing, the bit is turned off. NULL indicates that the string is not used.
exc	Points to a bit string that indicates which sockets are to be examined for pending exceptions. Each 1 bit causes the corresponding socket to be examined. If the socket has no pending exception, the bit is turned off. NULL indicates that the string is not used.
time	Points to a time structure that indicates how long the selectex( ) should wait. If this value is NULL, selectex( ) does not return until at least one socket is ready to process. Otherwise, control is returned when one or more sockets are ready or when the time interval expires, whichever is sooner.
ecb	Points to a single ECB or a list of ECBs. Multiple user ECBs are allowed in selectex( ) when the $OPTMECB option is set in $SOCKOPT. See "Appendix A: $SOCKOPT Options Phase," page 196, for details on setting options in a custom options phase. In addition to being terminated by a ready socket or time interval expiration, selectex( ) also returns when the ECB is posted.

For Assembler, COBOL and other languages, call the IPNRSELX entry point.

send( )

The send( ) function transmits data to a remote host.

```
int send(int,char *,int,int);
bytes_sent = send(socket,&msg,length,flags);
```

The variables are described in the following table:

Variable	Description
bytes_sent	If the call is successful, the number of bytes sent. A value of -1 indicates that an error has occurred, and the variable *errno* contains additional information.
socket	The socket number to be used for the transmission.
msg	Points to a buffer containing the data to be transmitted.

Variable	Description
length	The length of the data to transmit.
flags	Provided for compatibility. Code 0 as a placeholder.

By default, data is sent without waiting for an acknowledgement for improved performance. This can be overridden by removing the $OPTSNWT option setting in $SOCKOPT. See "Appendix A: $SOCKOPT Options Phase," page 196, for details on setting options in a custom options phase.

For Assembler, COBOL and other languages, call the IPNRSEND entry point.

sendto( )

The sendto( ) function can be used by UDP applications to send data to a remote host. It is similar to the send( ) function but has two additional parameters.

```
int sendto(int,char *,int,int,struct sockaddr *,length);
bytes_sent = sendto(socket,&buffer,len,flags,&sockaddr,ad_len);
```

The variables are described in the following table:

Variable	Description
bytes_sent	If the call is successful, this is the number of bytes sent. A value of -1 indicates that an error has occurred, and the variable *errno* contains additional information.
socket	The socket number to be used for the transmission.
buffer	Points to a buffer containing the data to be transmitted.
len	The length of the data to transmit.
flags	Provided for compatibility purposes. Code zero as a placeholder.
sockaddr	Points to a sockaddr structure that contains a port and/or IP address.
ad_len	The length of the sockaddr structure.

For Assembler, COBOL and other languages, call the IPNRSETO entry point.

seterrs_default( )

The seterrs_default( ) function indicates the default locations for the *errno* and *iprc* variables.

```
int seterrs_default(int *,int *);
rc = seterrs_default(&errno,&iprc);
```

The variables are described in the following table:

Variable	Description
rc	The return code. This is always zero.
errno	Points to the integer (fullword) variable that is to receive *errno* values. You can override this value for a particular socket with the seterrs_socket( ) function.
iprc	Points to the integer (fullword) variable that is to receive *iprc* values. You can override this value for a particular socket with the seterrs_socket( ) function.

By default, the *errno* and *iprc* variables reside in the socket driver program, and there is only one copy in each partition. If you want your C program to be reentrant, you must provide your own copies of these two variables.

For Assembler, COBOL and other languages, call the IPNRERRD entry point.

seterrs_socket( )

The seterrs_socket( ) function indicates the locations for the *errno* and *iprc* variables to be used for calls involving the specified socket.

```
int seterrs_socket(int,int *,int *);
rc = seterrs_socket(socket,&errno,&iprc);
```

The variables are described in the following table:

Variable	Description
rc	The return code for the operation. This is always zero.
socket	The socket to be associated with the variables.
errno	Points to the integer (fullword) variable that is to receive *errno* values.
iprc	Points to the integer (fullword) variable that is to receive *iprc* values.

By default, the *errno* and *iprc* variables reside in the socket driver program, and there is only one copy per partition. If you want your C program to be reentrant, you must provide your own copies of these two variables. The values coded with seterrs_socket( ) override those set with seterrs_default( ).

For Assembler, COBOL and other languages, call the IPNRERRS entry point.

setsockopt( )

The setsockopt( ) function allows you to set various option values available in a socket.

```
int setsockopt(int,int,int,char *,int);
rc = setsockopt(socket,sol_socket,option,&data,length);
```

The variables are described in the following table:

Variable	Description
rc	The return code. A 0 indicates success. A -1 indicates an error occurred, and variable *errno* has more information.
socket	The number of the socket to be modified.
sol_socket	The level of the call. This is the only permitted value.
option	The option to be set. Valid values are as follows: • SO_NONBLOCK: Sets the current socket to non-blocking for connects. When a connect( ) is issued without this option, the calling program is placed into a wait state until the connection request completes. This option enables the connect( ) to return immediately to the calling program, which uses a select for the socket write bit string to test for successful completion. • SO_LISTCHKC*:* Limits the number of connections allowed on a listening socket. The number of connections allowed is determined by the optional count parameter of the listen( ) function. If you do not set this option, the number of connections is unlimited (which is the default). • SO_REUSEADR: Allows multiple sockets to be bound to the same port. By default, a bind( ) function fails if another socket is already bound to the port specified in the bind request.
data	Points to an area containing option-dependent data. This field is ignored.

Variable	Description
length	The length of the option-dependent data. This field is ignored.

For Assembler, COBOL and other languages, call the IPNRSETS entry point.

setsysid( )

The setsysid( ) function can be used to set the ID of the TCP/IP FOR VSE stack you want your program to communicate with. This allows the program to override the default ID. See the section "Connecting to TCP/IP," page 27, for more information on setting the stack ID.

```
int setsysid(char);
rc = setsysid(id);
```

The variables are described in the following table:

Variable	Description
rc	The result contains the binary stack ID being used. A value of -1 indicates that an error has occurred, and the variable *errno* contains more information.
id	The stack id is the first and only parameter passed. The passed value must be 2 character bytes from 00 to 99. Optionally, you can pass the characters "GETS" to obtain the stack ID currently in use.

For Assembler, COBOL and other languages, call the IPNRSYID entry point.

shutdown( )

Under TCP/IP FOR VSE, the shutdown( ) function works exactly like the close( ) function, terminating all processing over a socket and returning the socket to the system.

```
int shutdown(int,int);
rc = shutdown(socket,how);
```

The variables are described in the following table:

Variable	Description
rc	The return code. A 0 indicates success. A -1 indicates that an error has occurred, and the variable *errno* contains more information.
socket	The socket number to be closed.

Variable	Description
how	Under some implementations, this value controls the functions of the sockets that are to be terminated. This parameter is ignored.

For Assembler, COBOL and other languages, call the IPNRSHUT entry point.

socket( )

The socket( ) function acquires a socket. The syntax is as follows:

```
int socket(int,int,int);
number = socket(domain,type,protocol);
```

The variables are described in the following table:

Variable	Description
number	If the call is successful, this is the number of the newly obtained socket. A value of -1 indicates that an error has occurred, and the variable *errno* contains additional information.
domain	The domain in which the socket is to be used. This must be coded as “AF_INET”.
type	The type of socket to be obtained. Supported types are `SOCK_STREAM` for TCP transmission `SOCK_DGRAM` for UDP transmission
protocol	The transmission protocol to be used. Supported protocols are `IPPROTO_TCP` for reliable TCP `IPPROTO_UDP` for unreliable UDP

For Assembler, COBOL and other languages, call the IPNRSOCK entry point.

takesocket( )

The takesocket( ) function acquires a socket from another program. Typically, the other program passes its client ID and socket descriptor and/or process ID to your program through your program's startup parameter list.

```
int takesocket(struct clientid *,int);
rc = takesocket(&clientid,socket);
```

The variables are described in the following table:

Variable	Description
rc	The return code. A 0 indicates success. A -1 indicates that an error has occurred, and the variable *errno* contains more information.
clientid	Points to the clientid structure that identifies the application from which you are taking a socket.
socket	The socket number.

For Assembler, COBOL and other languages, call the IPNRTAKE entry point.

Storage Functions

This section describes utility functions that provide subroutines for variable and bit manipulations. The entry point for Assembler and other high level languages is listed for each function.

bcopy( )

The bcopy( ) function copies one variable into another.

```
void bcopy(const void *,void *,int);
bcopy(&source,&target,length);
```

The variables are described in the following table:

Variable	**Description**
source	Points to the source variable
target	Points to the target variable
length	The integer length of both the source and the target

Usage Notes:

- The copy is performed with MVCL. Large variables are copied efficiently.
- The entire variable storage is copied, not just the occupied portion of character strings.
- The specified length cannot exceed the length of either the source or the target variable. If it does, storage overlays and random failures can occur.

For Assembler, COBOL and other languages, call the IPNRBCOP entry point.

bzero( )

The bzero( ) function clears the specified storage and fills it with binary zeros. No value is returned.

```
void bzero(void *,int);
bzero(&store,length);
```

The variables are described in the following table:

Variable	**Description**
store	Points to the variable to be filled with zeros
length	The length of the area to be cleared

The specified length cannot exceed the variable length. If it does, a larger storage area is cleared.

For Assembler, COBOL and other languages, call the IPNRBZER entry point.

htonl( )

The htonl( ) function translates a long integer (four bytes) from the computer’s internal byte order into network byte order. Because these are identical, the function executes as a simple assignment statement.

```
unsigned long htonl(unsigned long);
target = htonl(source);
```

For Assembler, COBOL and other languages, call the IPNRNILF entry point.

htons( )

The htons( ) function translates a short integer from the computer’s internal byte order into network byte order. Because these are identical, the function executes as a simple assignment statement.

```
unsigned short htons(unsigned short);
target = htons(source);
```

For Assembler, COBOL and other languages, call the IPNRNILH entry point.

inet_addr( )

The inet_addr( ) function converts a printable dotted-decimal IP (network) address to its binary equivalent.

```
unsigned long inet_addr(char *);
addr = inet_addr(&string);
```

The variables are described in the following table:

Variable	Description
addr	The returned network address. A value of -1 indicates that the source string could not be converted.
string	Points to a zero-delimited string containing the address to be converted.

The source string consists of one to four integers separated by periods. The resulting binary value is four bytes in length.

The integers are assigned to the individual bytes as explained in the following table:

Integers	Bytes
Four integers	Each integer is assigned to one byte in the final string.
Three integers	The left-most two integers are assigned to one byte each. The final integer is stored in two bytes as a 16-bit value.
Two integers	The left integer is stored as a byte. The second integer is stored in three bytes as a 24-bit value.
One integer	The value is stored as a single 32-bit value.

For Assembler, COBOL and other languages, call the IPNRINAD entry point.

inet_lnaof( )

The inet_lnaof( ) function examines a binary IP (network) address and returns only the host portion.

```
unsigned long inet_lnaof(struct in_addr);
host = inet_lnaof(ipaddr);
```

The variables are described in the following table:

Variable	Description
host	The host number
ipaddr	A binary network address

For Assembler, COBOL and other languages, call the IPNRINLN entry point.

C Definitions

Definitions File

C programs should include the socket.h file. This file contains all the function definitions, parameters, variables, and structures needed for creating a C-language socket application. These definitions include the address structures described in this section. Refer to the socket.h file for the complete set of definitions.

Address Tag Struct

The sockaddr variable maps an address tag for a socket. An address tag enables you to place information into a socket or retrieve existing information from a socket. This structure is as follows.

```
struct   sockaddr
    unsigned short    sa_family;   /* Set to AF_INET for IPv4 */
                                   /* or AF_INET6 for IPv6    */
    char              sa_data[14]; /* Address data            */
```

IPv4 Address Struct

For convenience, the IPv4 socket address structure is remapped by sockaddr_in with additional granularity. This structure is as follows:

```
struct   sockaddr_in         /* IPv4 socket address structure */
    short               sin_family;  /* Always set to AF_INET */
    unsigned short      sin_port;    /* Port number           */
    unsigned long       sin_addr;    /* Network (IP) address  */
    char                sin_zero[8]; /* Pad, binary zeros     */
```

Client ID Struct

The following clientid structure can be used to identify an application.

```
struct   clientid
    int                 domain(4)
    char                jobnmae(8)
    char                taskname(8)
    char                type(1)
    char                taskid(1)
    unsigned short      partitionid(2)
    int                 token(4)
```

Macros

The following macro instructions are included in the C socket.h file. These macros provide compatibility with other implementations and can simplify complex coding operations.

FD_SET

This macro turns on a single bit in a string. It is used in conjunction with the select( ) function.

```
FD_SET(n,p *)
```

The variables are described in the following table:

Variable	Description
n	The one-based bit number
p	Points to a bit string

FD_CLR

This macro turns off a single bit in a string. It is used in conjunction with the select( ) function.

```
FD_CLR(n,p *)
```

The variables are described in the following table:

Variable	Description
n	The one-based bit number
p	Points to a bit string

FD_ISSET

This macro tests a single bit in a string and determine its status. A value of 1 (true) is generated if the bit is turned on.

```
FD_ISSET(n,p *)
```

The variables are described in the following table:

Variable	Description
n	The one-based bit number
p	Points to a bit string

FD_COPY

This macro copies a bit string.

```
FD_COPY(f *,t *)
```

The variables are described in the following table:

Variable	Description
f	Points to the source string
t	Points to the target string, which must be the same length as the source string

FD_ZERO

This macro fills an entire bit string with binary zeros.

```
FD_ZERO(p *)
```

The variable is described in the following table:

Variable	Description
p	Points to a bit string

Assembler Definitions

Address Structures

The following address structures can be used to develop Assembler socket programs:

```
*
* * sockaddr_in IPv4 socket address structure
USOCKIN  DSECT
UFAMILY  DS       H
UPORT    DS       H
UIPADDR  DS       F
*
* * sockaddr_in6 IPv6 socket address structure
U6SOCKIN DSECT
U6LENGTH DS       XL1              Length of this structure
U6FAMILY DS       XL1              Protocl family
U6PORT   DS       XL2              UDP or TCP port number
U6FLOW   DS       XL4              IPv6 flow info(not used)
U6IPV4AD EQU      U6FLOW           IPv4 address(32-bits)
U6IPV6AD DS       XL16             IPv6 address(128-bits)
U6SCOPE  DS       XL4              Scoped interfaces(not used)
U6L      EQU      *-U6LENGTH       Length of this area
AF_INET  EQU      2                Protocol family for IPv4
AF_INET6 EQU      19               Protocol family for IPv6
SOCK_STREAM      DC F'1'           TCP stream socket
SOCK_DGRAM       DC F'2'           UDP datagram socket
*
*  For assembler programs, the errno is returned in:
         ENTRY IPERRNO
IPERRNO  DS      F
*
*  For assembler programs, the iprc is returned in R15 and:
         ENTRY IPRETCD
IPRETCD  DS      F
```

Sample Programs

Sample server and client programs are available to show how socket functions can be called from an Assembler language program. The source code for these programs is in the file SAMPBSDC.ZIP, which you can download from CSI International at www.csi-international.com.

These programs are described below.

Server

The BSDSERVR sample server program in SAMPBSDC.ZIP performs the following tasks:

- Initializes and attaches three VSE subtasks.
- Invokes the socket( ) function call to create a socket.
- Invokes the bind( ) function to port 6045 for the created socket.
- Invokes the listen( ) function to wait for a request from a client application named BSDCLINT.

- Dispatches one of the attached VSE subtasks to process the command and sends a reply to the client application that sent the command.

Client The BSDCLINT sample client program in SAMPBSDC.ZIP performs the following tasks:

- Invokes the socket( ) function call to create a socket.
- Invokes the connect( ) function to the BSDSERVR.
- Reads a command from SYSIPT.
- Invokes the send( ) function to send a command to BSDSERVR.
- Invokes the receive( ) function to receive a reply from BSDSERVR.
- Invokes the close( ) function to close the connection.

Error Handling

Return Codes and Error Numbers

The *iprc* and *errno* variables record the completion status of the functions. The *iprc* variable contains the return code, and *errno* contains the error number (reason) code. For most functions, a return code (*iprc*) of 0 indicates successful completion, and -1 indicates the request failed. For failed requests, the error number (*errno*) can be examined to determine the cause of the failure.

Error Number Descriptions

If a socket function returns with return code of -1, the error number in variable *errno* indicates why the function failed. The following table lists the error numbers, the associated Assembler equates, and their descriptions.

Errno	*errno* Equate	Message
100	EINTERNAL	/* Internal error */
113	EBADF	/* Parameter not valid */
118	EFAULT	/* Bad address or buffer N/A */
121	EINVAL	/* Invalid parameter */
122	EIO	/* Socket closed */
127	ENFILE	/* System file table is full */
129	ENOENT	/* No such socket */
132	ENOMEM	/* Not enough memory */
134	ENOSYS	/* Function not implemented */
158	EMVSPARM	/* Bad parameter */
183	EVSE	/* Not supported in VSE */
1102	EWOULDBLOCK	/* Request would block */
1103	EINPROGRESS	/* Socket connection in progress */
1104	EALREADY	/* Previous connection request incomplete */
1106	EDESTADDRREQ	/* Socket is not yet bound to a local */
1109	ENOPROTOOPT	/* No Option Recognized */
1112	EOPNOTSUPP	/* Socket call not supported */
1114	EAFNOSUPPORT	/* Address family not support */
1115	EADDRINUSE	/* Address or port in use */
1117	ENETDOWN	/* Ipnet Shutting Down */
1121	ECONNRESET	/* Connection reset/closed by peer */

Errno	*errno* Equate	Message
1122	ENOBUFS	/* No buffers available */
1123	EISCONN	/* Socket is already connected */
1124	ENOTCONN	/* Socket not connected */
1127	ETIMEDOUT	/* Connection request timed out */
1152	ECANCELED	/* Socket cancel */

Assembler and COBOL Programs

For Assembler and other high level languages, the *iprc* and *errno* values are returned into the IPRETCD and IPERRNO locations, respectively. Assembler programs must include the following storage definitions to reserve 4-byte areas for these entry points:

```
ENTRY IPRETCD
IPRC    DS     F   4-byte area reserved for BSD return code

ENTRY IPERRNO
ERRNO   DS     F   4-byte area reserved for BSD error number
```

If these external references are unresolved EXTRNs in the link-edit, then the return code and error number are not available to the application. Your application can then use the seterrs_default( ) and the seterrs_socket( ) functions to dynamically set the addresses for the return code (*iprc*) and error number (*errno*).

This means that there is one copy of each variable in each partition, and your program is not reentrant if it uses them. To make your application reentrant, you should still resolve IPRETCD and IPERRNO in static program storage, but then use the seterrs_default( ) or the seterrs_socket( ) function to assign dynamic storage for the *iprc* and *errno* variables.

Application Debugging

To help debug your socket program, or to analyze the functions and their results, you can create a debugging options phase ($SOCKDBG). You can use this phase without making any modifications to your program. When the trace is active, messages can be sent to the VSE system console (SYSLOG) and/or to the assigned printer (SYSLST) of the partition in which the application is executing.

See “Appendix B: $SOCKDBG Debugging Phase,” page 203, for details on enabling debugging messages and data dumps by specifying options in a custom $SOCKDBG debugging phase.

3

High Level Pre-Processor API

Overview

CSI International’s pre-processor application programming interface (API) provides a language-independent method of performing TCP/IP communications from your application program. The following steps describe how you use this facility.

1. Write your program as you normally would.
2. Insert language-independent calls to external routines provided by TCP/IP FOR VSE, if you want to use TCP/IP.
3. Pass your program through the TCP/IP FOR VSE pre-processor. The pre-processor replaces the language-dependent calls with code appropriate for the language you are using.
4. If necessary, pass your program through other pre-processors. These pre-processors may be provided by IBM or other vendors.
5. Compile your program.

In this chapter we show sample jobs that pre-process and compile a program. We then explain how to use the pre-processor and how to set up the connections and transmit the data. Finally, we provide sample programs coded in COBOL and PL/1.

Running Legacy JCL

The name of the pre-processor’s executable is IPNETPRE. Prior to release 1.5F, the phase name was IPNETRAN. If you use the IPNETRAN phase name in 1.5F and later releases, IPNETRAN simply loads IPNETPRE and passes control to it. This approach allows users to continue running JCLs that were written prior to release 1.5F.

IPNETPRE uses options that were not needed by IPNETRAN. To ensure that existing JCLs can be used without modification, you can store parameter values in the VSE library member IPNETPRE.L. These values are then used as the defaults. The IPNETPRE parameters are described in the section “Using the Pre-Processor” on page 65.

Pre-compiler Processing

After you code your application program and pass it through the TCP/IP FOR VSE pre-processor, you often must pass the output from CSI's pre-compiler through one or more pre-compilers provided by IBM or other vendors. The example in this section shows one way to do this.

The following JCL runs job SAMPLE1, which generates job SAMPLE2. Job SAMPLE2 places the pre-compiled output of the TCP/IP FOR VSE pre-processor in the library member $PRECOMP.WORK.

```
* $$ JOB JNM=SAMPLE1,CLASS=4,DISP=D
* $$ LST CLASS=P,DISP=D
* $$ PUN CLASS=A,DISP=I
// JOB SAMPLE1
// OPTION NOSYSDUMP,LOG,DECK
// LIBDEF *,SEARCH=(PRD2.TCPIP)
// EXEC ASSEMBLY
         PUNCH    '* $$ JOB JNM=SAMPLE2,CLASS=A,DISP=D'
         PUNCH    '* $$ LST CLASS=P,DISP=D'
         PUNCH    '// JOB SAMPLE2'
         PUNCH    '// EXEC LIBR'
         PUNCH    'ACCESS SUBLIB=PRD2.TCPIP'
         PUNCH    'CATALOG $PRECOMP.WORK REPLACE=YES'
         END
/*
// EXEC IPNETPRE,PARM='LANG=COBOL,ENV=CICS'
    Your Program...
    ...

/*
// OPTION DECK
// EXEC ASSEMBLY
         PUNCH    '/+'
         PUNCH    '/*'
         PUNCH    '/&&'
         PUNCH    '* $$ EOJ'
         END
/*
/&
* $$ EOJ
```

The example JCL below runs job SAMPLE3, which executes the IBM CICS COBOL pre-processor using the contents of library member $PRECOMP.WORK as input. This job generates job SAMPLE4, which replaces the contents of input library member $PRECOMP.WORK with the output from the pre-processor.

```
* $$ JOB JNM=SAMPLE3,CLASS=4,DISP=D
* $$ LST CLASS=P,DISP=D
* $$ PUN CLASS=A,DISP=I
// JOB SAMPLE3
// OPTION NOSYSDUMP,LOG,DECK
// LIBDEF *,SEARCH=(PRD2.TCPIP)
// EXEC ASSEMBLY
         PUNCH    '* $$ JOB JNM=SAMPLE4,CLASS=A,DISP=D'
         PUNCH    '* $$ LST CLASS=P,DISP=D'
         PUNCH    '// JOB SAMPLE4'
         PUNCH    '// EXEC LIBR'
         PUNCH    'ACCESS SUBLIB=PRD2.TCPIP'
         PUNCH    'CATALOG $PRECOMP.WORK REPLACE=YES'
         END
/*
// EXEC DFHECP1$,PARM='CICS'
* $$ SLI MEM=$PRECOMP.WORK,S=PRD2.TCPIP
/*
// OPTION DECK
// EXEC ASSEMBLY
         PUNCH    '/+'
         PUNCH    '/*'
         PUNCH    '/&&'
         PUNCH    '* $$ EOJ'
         END
/*
/&
* $$ EOJ
```

You must repeat the process for each preprocessor that you need to run.

Compiling Your Program

After all pre-processing is complete, the pre-processed library member is passed to a compiler, as shown in the following example. In this sample JCL, the library member $PRECOMP.WORK is passed to the COBOL compiler. The object deck is link-edited and cataloged to the library as phase SAMPLE. Note that any other language compiler or assembler could be invoked in place of COBOL.

Note: Do not use the “CBL DYNAM” option when compiling any COBOL program that will be invoking the API.

```
* $$ JOB JNM=SAMPLE5,CLASS=A,DISP=D
* $$ LST CLASS=P,DISP=D
// JOB SAMPLE5
// LIBDEF *,SEARCH=(PRD2.TCPIP,SPLIB2.PROD)
// LIBDEF PHASE,CATALOG=PRD2.TCPIP
// OPTION CATAL
 PHASE SAMPLE,*
/*
// OPTION LIST,LISTX,XREF
// EXEC FCOBOL
* $$ SLI MEM=$PRECOMP.WORK,S=PRD2.TCPIP
/*
/*
// EXEC LNKEDT
/&
* $$ EOJ
```

Using the Pre-Processor

This section explains how to use the pre-processor. It covers

- Execution order when using a CICS pre-processor
- Pre-processor parameters
- Pre-processor statements and syntax

Execution Order

Always run the TCP/IP pre-processor before the CICS pre-processor. The TCP/IP FOR VSE pre-processor generates EXEC CICS statements that are replaced by the CICS pre-processor.

Parameters

When you execute the pre-processing program (IPNETPRE), you can specify a number of parameters on the PARM field of the EXEC statement, along with an optional DEBUG parameter if you are encountering any problems.

Alternatively, you can include these parameter definitions within the IPNETPRE.L library member. You can use the CATALOG command to do this, as shown in the following example:

```
CATALOG IPNETPRE.L REP=Y EOD=/+
DEBUG=OFF
CC=ON
/+
```

The IPNETPRE parameters are described in the following table.

Parameter	Description
CC= [ON|OFF]	CC=ON forces an 81-byte output format with the first byte containing blanks. IPNETPRE is designed to output card images. If IJSYSPH is reassigned to a non-UR device type, it uses the appropriate length for that device type (80 bytes with no prefix for non-CC, and 81-bytes for CC). Some OEM products, however, may prevent the automatic length adjustment. The CC=ON setting allows you to correct this problem. Use this setting only if needed. The default is OFF.
DEBUG= [ON|OFF]	DEBUG=ON tells the API to output a large number of diagnostic messages when your application (not IPNETPRE) is running. It assists in debugging application problems with TCP/IP calls. To disable these messages, you must recompile your code with DEBUG=OFF. The default is OFF.

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>END-EXEC=
[YES|NO]</td><td>END-EXEC=NO overrides the default requirement to end each “EXEC *protocol command*” statement with an “END-EXEC” line. Command operands are on separate lines. When the pre-compiler comes to a string that is not a valid operand, it assumes the EXEC has reached the end.
This parameter provides compatibility with how IBM processes the EXEC call. The default is YES. See “Line Termination,” page 79, for related information.
Note: When END-EXEC=NO, any invalid operand in your code signals that the EXEC is ended. This may lead to syntax errors.</td></tr>
<tr><td>ENV=*env*</td><td>ENV sets the environment in which the finished program is to execute. Valid values for *env* are as follows:
<table>
<tr><td>BATCH</td><td>The program is to execute in batch mode</td></tr>
<tr><td>CICS</td><td>The program is to execute under CICS</td></tr>
</table></td></tr>
<tr><td>LANG=*lang*</td><td>LANG sets the language to be processed. Valid values for *lang* are as follows:
<table>
<tr><td>COBOL</td><td>For F-level COBOL, COBOL II, COBOL for VSE</td></tr>
<tr><td>PL1</td><td>For PL/I</td></tr>
<tr><td>ASSEMBLER</td><td>For Assembler F or High-Level Assembler</td></tr>
</table>
Note: Although Assembler programs can use this pre-processor, we recommend using the SOCKET macro interface for finer controls within an Assembler environment.</td></tr>
<tr><td>QUIET</td><td>QUIET specifies that only important messages are output. This parameter is not set to a value.</td></tr>
<tr><td>TRACE=
[ON|OFF]</td><td>TRACE=ON enables an “EXEC CICS TRACE” call to be generated within TCP/IP calls. If an “EXEC *protocol* TRACE(*value*)” is inserted into the application’s code, then the value in the source code is used. Using this parameter provides a value when none is already set in the source code. The default is OFF.</td></tr>
<tr><td>UPPER</td><td>UPPER specifies that all output is in upper case rather than mixed case. This parameter is not set to a value.</td></tr>
</table>

Pre-Processor Return Codes

The IPNETPRE return codes are described in the following table.

Return Code	Description
0	No errors were found in the input. IPNETPRE translated your EXEC calls into code native to the language specified by the LANG parameter. The next step is for you to compile your program.
12	One of the following errors was detected: • LANG is set to an unsupported language. • ENV is set to an unsupported environment.
16	One or more errors were detected in your input. To find the errors, review the contents of the punch file produced by IPNETPRE.
24	IPNETPRE was unable to load its message file. You probably do not have PRD2.TCPIP or PRD1.BASE in the search chain for the partition running IPNETPRE.

Pre-Processor Statements

All pre-processor statements contain the following elements:

- An identifying tag that specifies the call's protocol.
- A command verb that identifies the TCP/IP operation to be performed.
- Optional operands specific to the chosen command verb. Each operand is placed on a separate line.
- A closing tag that indicates the end of the statement. This tag may include an optional terminator. See "Line Termination," page 79, for more information.

The following format is used for COBOL, PL/1, and Assembler:

```
EXEC id verb
         operand1
         operand2
END-EXEC[terminator]
```

Identifying Tag

The identifying tag (*id*) on the EXEC statement identifies the TCP/IP calls to the pre-processor and must be a valid protocol value. The pre-processor examines each "EXEC *id*" statement and replaces it with language-dependent code that performs the specified functions.

The valid values for *id* are described in the table below.

***id* Value**	**Description**
TCP	EXEC TCP connects and communicates using the TCP interface. This allows a TCP connection to be established in a client or server mode, and it allows a TCP data stream to be transferred to or from the application program.
FTP	EXEC FTP connects to FTP. An FTP client manager is loaded into the TCP/IP FOR VSE partition to handle the FTP session. After the open request finishes, the application program sends FTP commands and receives responses to those commands across the connection. This allows the application program to manipulate an FTP client session from a program.
CLIENT	EXEC CLIENT connects with the general-purpose client manager. This client manager, within the TCP/IP FOR VSE partition, controls the function of a number of independent protocols. For example, LPR can be manipulated from the application program using the general client manager accessed by this service type.
UDP	EXEC UDP connects and communicates using the UDP interface. This allows a UDP connection to be established in a client or server mode, and it allows a UDP "connectionless" data stream to be transferred to or from the application program.
TELNET	EXEC TELNET communicates using the TELNET interface and protocol. TELNET is not just TCP with translation added, but it is also a protocol. This allows a TCP-like connection to be established in a client or server mode, and it allows a TCP data stream to be transferred to or from the application program. Because it is TELNET, it also checks certain handshaking bits and may disable translation if the other end requires it.
CONTROL	EXEC CONTROL communicates with the stack's internal control client and is used to request DNS information, run TCP commands, and more. It is not a true protocol because it does not send a datagram outside of the stack's system. This protocol is not the same as the "EXEC *protocol* CONTROL" command.

Command Verbs

The command verbs you can use are summarized in the following table.

***verb* Value**	**Description**
CONTROL	Controls pre-compiler execution. It does not generate code as other commands do. It sets options internal to IPNETPRE.
OPEN	Opens a connection
CLOSE	Completes processing and closes a connection
SEND	Sends data
RECEIVE	Receives data
ABORT	Aborts the connection
STATUS	Gets the status of a specific connection

Each of these verbs and their operands are described in the following sections. Many of the verbs use an operand called RESULTAREA, which is described next.

RESULTAREA

The RESULTAREA operand is used by several command verbs and defines the 56-byte work field used by the application program. The following table shows the RESULTAREA structure. See the programming examples at the end of this chapter for more information on how to code these fields in COBOL.

Field Name	**Field Length**	**Field Description**
ECB	Fullword	An ECB that is posted when the operation is complete
Local Port	Halfword	The local port number assigned by the system
Foreign Port	Halfword	The foreign port number
Foreign Address	Fullword	The IP (network) address of the remote host
Bytes Returned	Halfword	The number of bytes placed in the input buffer
Flags	Byte	Internal flag settings
Return Code	Byte	The return code for the operation
Terminal	40 bytes	Terminal LU information

When you use RESULTAREA in your programs, you need to specify the definitions for fullword, halfword, and address correctly. The following table shows how to do this in COBOL, PL/1, and Assembler. See the programming examples at the end of the chapter for reference.

Field Type	COBOL	PL/1	Assembler
Fullword	PICTURE X(4)	FIXED BIN(31)	DS F
Halfword	PICTURE 9(4) COMP	FIXED BIN(15)	DS H
Address	DATA NAME	VARIABLE	DS XL*nn*

CONTROL Verb

The CONTROL verb allows you to specify options that control the operation of the pre-compiler. The syntax is as follows:

```
CONTROL      DOUBLE
            [TRACE(YES|NO)]
```

The operands are described in the following table:

Operand	Description
DOUBLE	Directs the pre-processor to generate literals using double quotes (" "). The default is to use single quotes.
TRACE([YES|NO])	YES directs the pre-processor to generate CICS trace table entries. The default is NO.

The return codes are described in the following table:

Return Code	Description
0	No syntax problems.
4	A syntax error was detected.

OPEN Verb

The OPEN verb allows you to open a connection. You can specify a complete IP address or just a port. The operands permit you to actively connect with a remote host or establish a passive listening connection that waits for the remote host to initiate the connection.

OPEN and the other commands do not obey the “CICS” parameter if you code it. This parameter is controlled completely by IPNETPRE and is not included in the list of subparameters below.

The OPEN verb's syntax is as follows:

```
OPEN     FOREIGNPORT(halfword for assembler,
                     or any number for COBOL or PL/1)
         FOREIGNIP(4-byte character)
         LOCALPORT(halfword for assembler,
                   or any number for COBOL or PL/1)
         RESULTAREA(field name)
         DESCRIPTOR(fullword)
        [ACTIVE|PASSIVE]
         TIMEOUT(fullword for assembler,
                 or any number for COBOL or PL/1)
        [WAIT(YES|NO)]
         ERROR(label)
         SYSID(2-byte character field)
```

Notice the phrase "*or any number for COBOL or PL/1*." The value entered causes a "MOVE *value* TO *field*" to be generated, and so the length of the holding area, or even the use of an actual number itself, is not a problem for COBOL or PL/1. For Assembler, however, the "MVC" is used. In that case, either the =F'*value*' or =H'*value*' constants can be used, or it should be a field with the exact characteristic noted in the syntax statement above. For example, a halfword is used to move a halfword, and a fullword is used to move a fullword.

The operands are described in the following table:

Operand	**Description**
FOREIGNPORT (*halfword*)	Identifies the number of the foreign port TCP/IP FOR VSE is to connect with or transmit to. It is used with the ACTIVE operand to specify the foreign port to connect with. It is a required operand and there is no default.
FOREIGNIP (*4-byte char*)	Use with the ACTIVE operand to specify the network address of the remote host to connect to. This is a 4-byte character field, not a numeric field. Use with the PASSIVE operand to specify a network address that may connect with this application. In passive mode, the address can be a specific IP address or a generic IP address. To code a generic IP address, specify a zero in any one of the octets that comprise the address. For example, specify a foreign IP address of 192.168.1.0 (X'C0A80100') to allow any address on the 192.168.1 subnet to connect to the application.

Operand	**Description**
LOCALPORT (*halfword*)	Specifies the port number to be used at the local end. If not specified or if set to zero, the system assigns you a port from the available pool. Generally, if you are writing a TCP server application (meaning that you are using PASSIVE mode), you need to specify the local port so that the foreign host knows which port to connect to. If you are connecting to another host (meaning that you are using ACTIVE mode), the local port generally is of no consequence.
RESULTAREA (*field name*)	Specifies a 56-byte structured area in your program that is to contain the results of various EXEC TCP operations. The structure is shown earlier in this chapter. Use the name of the field, not a pointer to a field.
DESCRIPTOR (*fullword*)	Identifies the thread of operation. This field must point to a fullword in storage where the descriptor value can be stored, and this descriptor must be passed to all subsequent socket calls for the open connection. In other words, once a connection has been established, any number of independent socket calls can be made to that connection, but all of them must use the descriptor field to reference the open connection.
[ACTIVE| PASSIVE]	ACTIVE (the default.) Actively and immediately establishes a connection to a remote host. A client application usually starts with an active open and attempts to connect to a specific server application that is passively listening (waiting for clients to connect to it). The client's IP address and port number are sent to the server application during the active open processing. The server can then transmit data to the client since it has the client's IP address and port number. PASSIVE. Listens for an incoming connection. A passive open may use either a fully specified foreign socket (FOREIGNIP(*n.n.n.n*) and FOREIGNPORT(0)) to wait for a specific client to connect with, or an unspecified foreign socket (FOREIGNIP(0) and FOREIGNPORT(0)) to wait for any client connection. A server application normally starts with a passive open on a preassigned port number and waits for clients to connect to it. The exception to this is UDP, which is "connectionless," and so the OPEN completes immediately. The user at that point should queue up a RECEIVE request. To connect to this server application, the client must know the server's IP address and port number. This operand is the opposite of the ACTIVE operand.

Operand	Description
TIMEOUT (*fullword*)	Specifies the time allowed for an active OPEN operation to complete. If the operation does not complete within this time, control is returned to your program. The value is the number of 300th-second intervals. 18000, for example, specifies one minute. To determine the value for a 2-minute timeout, use this calculation: 2 minutes x 60 x 300 = 36000 For a PASSIVE open, the timeout parameter specifies the maximum amount of time the foreign host is given to complete the connection once a SYNC request is received. A value of 0 means it never times out in this circumstance.
WAIT([YES|NO])	Indicates whether control is to be returned to your program before the requested operation is complete. Specify YES (the default) to return control only when the operation is complete. Specify NO to return control as soon as the operation is scheduled. If you specify NO, you must wait on the XOAECB yourself.
CICS	Indicates the program is a CICS application.
ERROR(*label*)	Specifies a branch location. If an error occurs, an immediate branch is taken to the specified label.
SYSID(*nn*)	Specifies the TCP/IP FOR VSE system identifier of the system making the request. The default is 00. You must specify a SYSID value if your installation is running TCP/IP with a system identifier other than 00. The value should be in a 2-byte character field. Coding it as a numeric may result in a connection failure (use `=X'00'` rather than `00` in Assembler, for example).

For return code information, see "Error Checking" on page 81.

CLOSE Verb

The CLOSE verb completes processing and closes an existing connection. Closing a connection is intended to be a graceful operation: outstanding SENDs continue to be transmitted until all data is sent. It is acceptable to issue several SENDs followed by a CLOSE and to expect all the data to be sent to the destination. The user can close the connection at any time. Because closing a connection requires communication with the foreign socket, connections can stay in the closing state for a short time.

A CLOSE no longer requires a WAIT condition to occur, and so none is generated. Because CLOSE always generates a good return code, no ERROR field is required. While you may code those parameters, they are ignored and are not listed in the table below.

You must not generate your own WAIT within your application or your program will remain in a wait state. The SYSID value is established by the OPEN command, so if it is coded, it is ignored as well. The syntax is as follows:

```
CLOSE     DESCRIPTOR(fullword)
          RESULTAREA(field name)
```

The operands are described in the following table:

Parameter	**Meaning**
DESCRIPTOR (*fullword*)	Identifies the thread of operation. This field must point to a fullword in storage where the descriptor value from a previously issued OPEN call was stored.
RESULTAREA (*field name*)	Specifies a 56-byte structured area in your program to contain the results of various EXEC TCP operations. The structure is shown earlier in this chapter.

For return code information, see "Error Checking" on page 81.

SEND Verb

The SEND verb transmits data. When you issue this command, the data contained in the indicated user buffer is sent. The ECB of the RESULTAREA is posted when the data buffer is accepted by the TCP/IP partition. Because the SYSID value is established at OPEN time and IPNETPRE controls the "CICS(*value*)" process, those parameters, while accepted, are ignored and are not listed in the table below. The syntax is as follows:

```
SEND      FROM(string-address)
          LENGTH(fullword)
          RESULTAREA(field name)
          DESCRIPTOR(fullword)
         [WAIT(YES|NO)]
          ERROR(label)
```

The operands are described in the following table:

Operand	**Description**
FROM (*string-address*)	Specifies the address of a buffer containing the data to be transmitted. Use the LENGTH operand to indicate the amount of data in the buffer.

Operand	**Description**
LENGTH (*fullword*)	Tells TCP/IP FOR VSE how much data to send along the connection. This is a required parameter. There is no default.
RESULTAREA (*address*)	Specifies a 56-byte structured area in your program to contain the results of various EXEC TCP operations. The structure is shown earlier in this chapter.
DESCRIPTOR (*fullword*)	Identifies the thread of operation. This field must point to a fullword in storage where the descriptor value from a previously issued OPEN call was stored.
WAIT([YES|NO])	Indicates whether control is to be returned to your program before the requested operation is complete. Specify YES to return control only when the operation is complete. Specify NO to return control as soon as the operation is scheduled. If you specify NO, you must wait on the XOAECB yourself.
ERROR(*label*)	Specifies a branch location. If an error occurs, an immediate branch is taken to the specified label.

For return code information, see “Error Checking” on page 81.

RECEIVE Verb

The RECEIVE verb receives data over an open connection to a remote host. Depending on your buffer’s length, you may receive a complete or partial transmission. You may need to issue multiple consecutive RECEIVEs to obtain a complete transmission. The syntax is as follows:

```
RECEIVE    TO(field name)
           LENGTH(fullword)
           RESULTAREA(address)
           DESCRIPTOR(fullword)
           TIMEOUT(fullword)
          [WAIT(YES|NO)]
           ERROR(label)
```

The operands are described in the table below.

Operand	**Description**
TO(*field name*)	Specifies the buffer to receive data sent from the remote host. Use the LENGTH operand to indicate the length of the area. When a RECEIVE operation completes, the length of data actually received is contained in the RESULTAREA. This is a required operand. There is no default.
LENGTH (*fullword*)	Tells TCP/IP FOR VSE how much data you want to receive in this request. When a RECEIVE operation completes, the length of the received data is contained in the RESULTAREA. If you do not have enough data queued to your application to fill the buffer, TCP/IP FOR VSE returns whatever data is available and adjusts the length in the RESULTAREA. **Caution**: If your LENGTH is bigger than your buffer, you run the risk of overlaying storage that you won't destroy in your program.
DESCRIPTOR (*fullword*)	Identifies the thread of operation. This field must point to a fullword in storage where the descriptor value can be stored, and this descriptor must be passed to all subsequent socket calls for the open connection. In other words, once a connection has been established, any number of independent socket calls can be made to the connection, but all of them must use the descriptor field to reference the open connection.
TIMEOUT (*fullword*)	Specifies the time allowed for a RECEIVE operation to complete. Once this time is exceeded, control is returned to your program. The value is the number of 300th-second units. 18000, for example, specifies one minute. To determine the value for a 2-minute timeout, use this calculation: 2 minutes x 60 x 300 = 36000 The default is to wait forever for the RECEIVE to complete.
WAIT([YES|NO])	Indicates whether control is to be returned to your program before the requested operation is complete. Specify YES to return control only when the operation is complete. Specify NO to return control as soon as the operation is scheduled. If you specify NO, you must wait on the XOAECB yourself.
ERROR(*label*)	Specifies a branch location. If an error occurs, an immediate branch is taken to the specified label.

Operand	Description
RESULTAREA (*address*)	Specifies a 56-byte structured area in your program to contain the results of various EXEC TCP operations.

ABORT Verb

The ABORT verb closes an existing connection. Unlike the CLOSE verb, however, it is similar to taking an axe to a communication line. There may be a delay in the elimination of the SOCKET, and the session, from the point of view of the other side, was interrupted and may go into some sort of recovery. When you cannot terminate a connection gracefully, then this is the way to do it. If you issue an ABORT, then you cannot issue a CLOSE because the connection no longer exists. An internal monitor tries to clean up the leftovers that result from killing the session. The syntax is as follows:

```
ABORT      DESCRIPTOR(fullword)
           RESULTAREA(field name)
          [WAIT(YES|NO)]
           ERROR(label)
```

The operands are described in the following table:

Parameter	Meaning
DESCRIPTOR (*fullword*)	Identifies the thread of operation. This field must point to a fullword in storage where the descriptor value from a previously issued OPEN call was stored.
RESULTAREA (*field name*)	Specifies a 56-byte structured area in your program to contain the results of various EXEC TCP operations. The structure is shown earlier in this chapter.
WAIT([YES|NO])	Indicates whether control is to be returned to your program before the requested operation is complete. Specify YES to return control only when the operation is complete. Specify NO to return control as soon as the operation is scheduled. If you specify NO, you must wait on the XOAECB yourself.
ERROR(*label*)	Specifies a branch location. If an error occurs, an immediate branch is taken to the specified label.

For return code information, see “Error Checking” on page 81.

STATUS Verb

The STATUS verb directs the API to issue a SOCKET STATUS call against an existing socket. It returns the CCBLOK into the buffer indicated for the application program to review. This is not done to check whether a SOCKET call is complete or not; that can be done by checking the ECB. Rather, it is used to gather control block details that would be useful, such as after issuing an OPEN with a local port number of zero, meaning that a port number will be assigned to it by the stack.

This allows you to obtain the assigned port number before there is any connection. Most programs rarely need this. But if your program does, remember that the information returned is a raw control block. You must map it back to the layout defined in the STATBLOK.A macro that comes with the product. You must produce this layout if it is needed.

The syntax of STATUS is as follows:

```
STATUS      TO(field name)
            LENGTH(halfword)
            RESULTAREA(address)
            DESCRIPTOR(fullword)
            WAIT(YES)
            ERROR(label)
```

The operands are described in the following table:

Operand	**Description**
TO(*field name*)	Specifies the buffer to receive data sent from the remote host. Use the LENGTH operand to indicate the area’s length. When a STATUS operation completes, the length of the received data is contained in the RESULTAREA. This is a required operand.
LENGTH (*halfword*)	Tells TCP/IP FOR VSE how much data you want to receive in this request. When a STATUS operation completes, the length of data actually received is contained in the RESULTAREA. If you do not have enough data queued to your application to fill the buffer, TCP/IP FOR VSE returns whatever data is available and adjusts the length in the RESULTAREA. **Caution:** If your LENGTH is bigger than your buffer, you risk overlaying storage in the program that you don’t intend to destroy.

Operand	**Description**
DESCRIPTOR (*fullword*)	Identifies the thread of operation. This field must point to a fullword in storage where the descriptor value can be stored, and this descriptor must be passed to all subsequent socket calls for the open connection. In other words, once a connection has been established, any number of independent socket calls can be made to the connection, but all of them must use the descriptor field to reference the open connection.
WAIT(YES)	Indicates whether control is to be returned to your program before the requested operation is complete. Specifying YES returns control only when the operation is complete. Because this is just an in-memory move, always code YES.
ERROR(*label*)	Specifies a branch location. If an error occurs, an immediate branch is taken to the specified label. The only type of error would be if the CCBLOK does not exist, which would mean the SOCKET does not exist.
RESULTAREA (*address*)	Specifies a 56-byte structured area in your program to contain the results of various EXEC TCP operations. The structure is shown earlier in this chapter.

For return code information, see "Error Checking" on page 81.

Line Termination

Generated COBOL and PL/1 code lines may require termination with the appropriate punctuation mark. This terminator is referred to as a full stop or an implicit scope terminator. It ends on-going conditions. The COBOL terminator is a period (.); the PL/1 terminator is a semicolon (;).

Specifying a terminator after the END-EXEC (only) in pre-processor statements causes that punctuation to be added to the generated statements. For example, if the END-EXEC is followed with a period, then all the generated statements, with the exception of a continued statement, will end with a period. If a valid terminator (a period or a semicolon) is missing, then no punctuation is generated.

The example below shows the effect of using “END-EXEC.” (with a period) on generated COBOL statements.

```
MOVE 'Y' TO XOWAIT.
MOVE 'N' TO CICS.
...other code
IF XOCODE > 0 THEN
   GO TO ERROR-AREA.
```

Note 1: If END-EXEC=NO in the IPNETPRE parameter list, then the pre-processor checks each operand in the “EXEC *protocol command*” statement for a terminator. If the last operand ends with a terminator, then that ending punctuation is added to the generated statements. The terminator must be used only on the last operand. If it is used on another operand, then the pre-processor ends the EXEC statement at that point. The following example shows how a period is specified as a terminator in a pre-processor statement when END-EXEC=NO.

```
EXEC TCP OPEN FOREIGNPORT(0000)
              LOCALPORT(LOCAL_PORT)
              RESULTAREA(RESULTS)
              SYSID('00')
              DESCRIPTOR(MYDESC)
              PASSIVE(YES)
              WAIT(YES).
...other code
```

Note 2: If an “EXEC TCP” (for example) is made part of a conditional (IF) statement, adding an ending punctuation would terminate the IF and possibly change the logic of the statement. In the following pre-processor example, a terminator is not specified after the END-EXEC to avoid adding punctuation to the generated code.

```
IF SOMETHING = "Y" THEN
      EXEC TCP
              operand1
              operand2
      END-EXEC
ELSE
      CALL SOMETHING-ELSE.
```

Error Checking

XOBLOK Control Block

While we do not want any errors to occur in our TCP/IP processing, there may, of course, come a time when your program has a problem. Perhaps the TCP/IP stack is down, the DNS server is down, or some other problem occurs. It is important to have a centralized ERROR-reporting routine in your program that reports the problem and provides you with useful information before recovering or terminating.

In the case of a PL/1 or COBOL program, an XOBLOK control block is inserted into your code automatically when you run it through IPNETPRE. Assembler programs need to have an XOBLOK macro somewhere in the program to cause this control block to be generated. Using DSECT=YES makes this value a DSECT you can map back to a storage area, while DSECT=NO codes this control block within program storage.

The XOBLOK is initialized at OPEN time, and each TCP/IP verb call (for example, OPEN or SEND) causes small parts of this control block to be set before the block is passed back to the API. The API, in turn, sets various fields in XOBLOK before returning control back to the application program. If an error occurs, the values in certain XOBLOK fields are important. These fields are described in the table below.

XOBLOK Field	Description
XOAPIV1	An 8-byte character field containing the version/release of IPNETPRE that generated the code.
XOAPIV2	An 8-byte character field containing the version/release of the API that tried to service the code.
XOCALLID	A numeric halfword indicating which EXEC within your program caused the problem. For example, if you had an EXEC TCP CONTROL followed by an EXEC TCP OPEN, and the problem was in the EXEC TCP SEND (the third EXEC), then XOCALLID would contain X'0003'.
XOREG1	Operation field. This value identifies where the failure occurred so that the codes can be interpreted properly. Even-numbered values are assigned if an operation failed to schedule. Odd-numbered values are assigned if the operation completed with errors.

XOBLOK Field	Description
XOCODE / XORCODE	Return code field. For convenience, two fields are provided: XOCODE is a halfword field, and XORCODE is a one-byte field. For even XOREG1 values (2–12), XOCODE’s value means the same as the SOCKET macro return code in register 15. (This value is not in register 15.) For odd XOREG1 values, XOCODE’s value is in SRCODE.
XOREG0	Reason code field. This value may clarify the meaning of the XOCODE value. For some XOCODE values, XOREG0 contains the IBM return code if the failure occurred during an IBM call, such as GETMAIN. For even XOREG1 values, the XOCODE, XOREG0 code combinations have the same meanings as the register 15, register 0 codes. (See the next section for information on how to look up these codes.)

The contents of these fields should be made available to the user so that problems can be debugged. The first three fields may be needed by CSI’s technical support group if you are unable to solve the problem. The XOREG1, XOCODE, XOREG0 fields can be used to identify the error.

API calls set the following XOREG1 codes.

XOREG1	Operation
2	OPEN SOCKET scheduled
3	OPEN SOCKET completed
4	SEND SOCKET scheduled
5	SEND SOCKET completed
6	RECEIVE SOCKET scheduled
7	RECEIVE SOCKET completed
8	STATUS SOCKET scheduled
9	STATUS SOCKET completed
10	ABORT SOCKET scheduled
11	ABORT SOCKET completed
12	CLOSE SOCKET scheduled
13	CLOSE SOCKET completed
14	XOBLOK validation failure

Error Determination

Use the XOREG1 code and the following table to interpret the codes in XOCODE and XOREG0. Look up the values of those fields in the indicated table to determine the error. The listed tables are in chapter 1, "SOCKET Assembler API."

If XOREG1 contains...	then XOCODE contains...	and XOREG0 contains...	Look up the error in this table:
2, 4, 6, 8, 10, or 12	a Register 15 value	a Register 0 value	Reg15,Reg0 Codes (page 9)
3, 5, 7, 9, 11, or 13	an SRCODE value		SRCODE Field (page 11)
14, see table below.			

If XOREG1 = 14, use the table below to interpret the XOCODE value.

XOCODE	Meaning
3	"EXEC CICS WAIT" initiated a timeout.
16	The XODESC field is null for any operation except an OPEN, or it is NOT null when an OPEN call is made.
20	XOBLOK is NOT null for an OPEN call. This may indicate that the XOBLOK field is already being used for another connection.
24	The contents of XOCOMMND are invalid. XOCOMMND is used to pass commands such as OPEN and CLOSE to the API.
28	The contents of XOACCESS are invalid. XOACCESS is used to pass the specified protocol (TCP or UDP) to the API.

Connections and Data Transmission

You must open a connection before you can communicate. Connections can be either active or passive. An *active connection* seeks out the specified partner and actively negotiates the connection. A *passive connection* takes no action on its own but rather waits to receive a negotiation request from the remote end.

Active Connection Example

In the following example, TCP/IP FOR VSE is asked to establish a connection with a foreign system whose IP address is held in fullword IPADDRESS and whose foreign port number is 2000. The fullword SOCKDESC must be passed to all subsequent EXEC TCP calls for this connection once it has been established. This is the only call necessary to establish a complete connection with the foreign system. Once the RECB (contained in the result area) is posted, the connection is ready for send and receive activity.

An active connection request must be complete within a timeout period. Because the TIMEOUT= specification is omitted, the timeout value defaults to two minutes. If the connection is not complete within two minutes, the RECB is posted, and an error condition is set in the RCODE field.

```
           05  IPADDRESS.
               10 IPAD1    PICTURE X.
               10 IPAD2    PICTURE X.
               10 IPAD3    PICTURE X.
               10 IPAD4    PICTURE X.
           05  HALFWORD    PICTURE 9(4) COMP.
           05  HALFWORD-X  REDEFINES HALFWORD.
               10 BYTE1    PICTURE X.
               10 BYTE2    PICTURE X.
           05  RESULTS.
               10 RECB     PICTURE X(4).
               10 RLOPORT  PICTURE 9(4) COMP.
               10 RFOPORT  PICTURE 9(4) COMP.
               10 RFOIP    PICTURE X(4).
               10 RCOUNT   PICTURE 9(4) COMP.
               10 RFLAGS   PICTURE X.
               10 RCODE    PICTURE X.
               10 RTERMTY  PICTURE X(40).
           05  SOCKDESC    PICTURE X(4).
*
*    Setup IPADDRESS to hold 172.20.10.10 in binary
           MOVE 172 TO HALFWORD.
           MOVE BYTE2 TO IPAD1.
           MOVE 20  TO HALFWORD.
           MOVE BYTE2 TO IPAD2.
           MOVE 10  TO HALFWORD.
           MOVE BYTE2 TO IPAD3.
           MOVE 10  TO HALFWORD.
           MOVE BYTE2 TO IPAD4.
```

(continued next page)

(continued)

```
*
*    Attempt to open a connection at 172.20.10.10 port 2000
       EXEC TCP OPEN FOREIGNPORT(2000)
                     FOREIGNIP(IPADDRESS)
                     LOCALPORT(0)
                     RESULTAREA(RESULTS)
                     DESCRIPTOR(SOCKDESC)
                     ACTIVE
                     WAIT(YES)
                     ERROR(BAD-OPEN)
           END-EXEC.
```

Passive Connection Example

In the following example, the application program requested that TCP/IP FOR VSE listen for the arrival of a connection request at port 2500. When a connection request arrives, the connection is completed and this request is posted as complete. Notice the foreign port and IP address are not specified. This allows any remote user’s connection request to be accepted.

After the connection is established, further connection requests for this local port number are rejected unless there are other server programs waiting for this same local port number. Unlike an active connection, there is no timeout period while waiting for a connection request. Once a request arrives, however, the timeout values are observed for completing the connection.

```
*    Attempt to open a passive connection
*
       EXEC TCP OPEN FOREIGNPORT(0)
                     FOREIGNIP(0)
                     LOCALPORT(2500)
                     RESULTAREA(RESULTS)
                     DESCRIPTOR(SOCKDESC)
                     PASSIVE
                     ERROR(BAD-OPEN)
            END-EXEC.
```

Receiving Data

After a connection is established, the next step is to receive or send data. This section shows how to use EXEC TCP RECEIVE to receive data from the foreign host.

In the example below, the EXEC TCP RECEIVE queues a request to receive information from the foreign host. You may have many receive requests queued on the same connection. Each request must provide its own RESULTAREA but refer to the same DESCRIPTOR. Each request is processed in the order it is queued, and data is passed to the application as it arrives. The maximum information that can be received in one request is 65,535 bytes. Because it is unusual for 64K of data to arrive at one time, such a specification wastes memory.

The received information generally arrives in pieces no larger than the link’s MTU size. The amount of data received is returned in the RCOUNT field.

It is important to remember that TCP is a stream-oriented protocol, and iteration is required when receiving a data stream. This is very different from most I/O operations and requests that are record or block oriented. A disk or tape VSE I/O operation issues a read request, waits for an ECB to signal completion, checks for errors, and then has the entire record or block available for processing. A TCP stream application may send 4096 bytes of data to a receiving application all in one receive, or it may receive 2000 bytes, then 1000 bytes, and then 1096 bytes in three separate receive requests. TCP guarantees to deliver the bytes in sequence, but it does not guarantee to deliver them in the grouping in which they were sent.

To handle a TCP stream, the receiving application must loop on the input stream until the agreed-upon data structure is received. An example of an agreed-upon data structure is the telnet or FTP protocol, in which a carriage return/line feed in the data stream indicates the end of a command or record.

The receiving application is then coded to loop, receiving the data stream into a buffer until the carriage return/line feed characters are detected in the stream. At that point, the receiving application knows it has received a complete command or record that can then be processed.

The following example shows how to receive data from a foreign host:

```
        05  RESULTS.
            10 RECB     PICTURE X(4).
            10 RLOPORT  PICTURE 9(4) COMP.
            10 RFOPORT  PICTURE 9(4) COMP.
            10 RFOIP    PICTURE X(4).
            10 RCOUNT   PICTURE 9(4) COMP.
            10 RFLAGS   PICTURE X.
            10 RCODE    PICTURE X.
            10 RTERMTY  PICTURE X(40).
        05  SOCKDESC    PICTURE X(4).
        05  BUFFER.
            10  WORKAREA    PICTURE X(512).
*
*    Receive a piece of data
*
       EXEC TCP RECEIVE
                     TO(BUFFER)
                     LENGTH(512)
                     RESULTAREA(RESULTS)
                     DESCRIPTOR(SOCKDESC)
                     WAIT(YES)
                     ERROR(BAD-RECEIVE)
            END-EXEC.
```

Sending Data

The example below shows how to transmit data across a connection. In this example, 512 bytes of data are sent across the connection to the foreign system. The RECB in the RESULTAREA is posted when the data is accepted by the TCP/IP partition, but it does not mean the data buffer has arrived at the desired location. The send request must refer to the DESCRIPTOR created during the EXEC TCP OPEN processing. Send requests are processed in the order they are issued. The maximum buffer size is 65,535 bytes. Regardless of the buffer size used, when each send request is accepted by the TCP/IP partition, it is broken into different-sized pieces for actual transmission.

```
        05  RESULTS.
            10 RECB     PICTURE X(4).
            10 RLOPORT  PICTURE 9(4) COMP.
            10 RFOPORT  PICTURE 9(4) COMP.
            10 RFOIP    PICTURE X(4).
            10 RCOUNT   PICTURE 9(4) COMP.
            10 RFLAGS   PICTURE X.
            10 RCODE    PICTURE X.
            10 RTERMTY  PICTURE X(40).
        05  SOCKDESC    PICTURE X(4).
        05  BUFFER.
            10  WORKAREA    PICTURE X(512).
*
*    Sends a piece of data
       EXEC TCP SEND
                     FROM(BUFFER)
                     LENGTH(512)
                     RESULTAREA(RESULTS)
                     DESCRIPTOR(SOCKDESC)
                     WAIT(YES)
                     ERROR(BAD-SEND)
            END-EXEC.
```

Closing a Connection

After you are done with the connection you must close it. The following example shows how to do this. This is a simple operation that requires only the DESCRIPTOR and the RESULTAREA. Although the CLOSE operation is queued behind any outstanding SEND operations, it is good practice to allow previously queued SEND and RECEIVE requests to complete before issuing a CLOSE.

```
        05  RESULTS.
            10 RECB     PICTURE X(4).
            10 RLOPORT  PICTURE 9(4) COMP.
            10 RFOPORT  PICTURE 9(4) COMP.
            10 RFOIP    PICTURE X(4).
            10 RCOUNT   PICTURE 9(4) COMP.
            10 RFLAGS   PICTURE X.
            10 RCODE    PICTURE X.
            10 RTERMTY  PICTURE X(40).
        05  SOCKDESC    PICTURE X(4).
*
*    Close the connection
       EXEC TCP CLOSE
                     RESULTAREA(RESULTS)
                     DESCRIPTOR(SOCKDESC)
                     ERROR(BAD-CLOSE)
            END-EXEC.
```

Using WAIT(NO)

In some cases, you may want to use WAIT(NO) as one of the parameters in your TCP/IP FOR VSE application to manually wait within the program. When your application calls the API, the API sets a value in XOAECB that points to a field containing a pointer. In a non-CICS application, you would load the address in XOAECB into a register and invoke the WAIT macro from IBM. For a CICS application, the XOAECB field can be used as a parameter in the normal CICS WAIT EXTERNAL request.

The three examples below show how this works for each program type. The XOAECB field contains the pointer the statement requires. “NEV” is a fullword field that contains a ‘1’. You can insert a ‘2’ into NEV if you choose to wait on a second ECB. In that case, you must update the area to which XOAECB points. The area already contains one ECB pointer in the first fullword, and there is room to include a second ECB pointer.

The following examples will work for batch or CICS with the exception of the issuing of the WAIT. COBOL and PL/1 cannot issue a WAIT call because of how they are designed to function, but must instead call an assembler routine written by the user to do that WAIT for them. The assembler example *could* run in a batch partition by replacing the single CICS call to issuing the IBM “WAIT” macro, as noted above.

PL/1 Example

The following example is coded in PL/1:

```
EXEC TCP OPEN FOREIGNPORT(0000)
              LOCALPORT(LOCAL_PORT)
              RESULTAREA(RESULTS)
              SYSID('00')
              DESCRIPTOR(MYDESC)
              PASSIVE(YES)
              WAIT(NO);
/*                               */
IF XORCODE ^= '00000000'B THEN GOTO RETPROG;
EXEC CICS WAIT EXTERNAL ECBLIST(XOAECB) NUMEVENTS(NEV);
```

COBOL Example

The following example is coded in COBOL:

```
EXEC TCP OPEN FOREIGNPORT(0000)
              LOCALPORT(LOCAL_PORT)
              RESULTAREA(RESULTS)
              SYSID(00)
              DESCRIPTOR(MYDESC)
              PASSIVE(YES)
              WAIT(NO).
IF XORCODE > ZERO THEN GOTO RETPROG.
EXEC CICS WAIT EXTERNAL ECBLIST(XOAECB) NUMEVENTS(NEV).
```

Note: Do not use quotes around the two-byte SYSID value.

Assembler Example

The following example is coded in Assembler:

```
EXEC TCP OPEN FOREIGNPORT(0000)
              LOCALPORT(LOCAL_PORT)
              RESULTAREA(RESULTS)
              SYSID('00')
              DESCRIPTOR(MYDESC)
              PASSIVE(YES)
              WAIT(NO)
ICM  R15,15,XORCODE
BNZ  RETPROG
EXEC CICS WAIT EXTERNAL ECBLIST(XOAECB) NUMEVENTS(NEV)
```

The API gives the same result when WAIT(YES) is used. So, unless you need to use a second ECB in your ECBLIST, coding WAIT(YES) is much simpler.

Note: Never code WAIT(YES) on a CLOSE. The stack controls the waiting.

Sample Programs

The following sample programs show the same functions in different programming languages. In each case, note the special techniques used to manipulate the data.

Note: The EXEC TCP command is shown in the following examples. If non-binary data is transferred and EDCDIC-ASCII translation is needed in both directions, you can use the EXEC TELNET command unless you want to perform your own in-program translation. For information on command parameters, refer to the appropriate COBOL reference manual.

COBOL EXEC TCP Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID.       SAMPLE2.
AUTHOR.           JOHN R.
INSTALLATION.     WORTHINGTON OHIO.
DATE-WRITTEN.     AUGUST 2, xxxx.
DATE-COMPILED.

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

DATA DIVISION.
EXEC TCP CONTROL DOUBLE(NO)
     END-EXEC.
WORKING-STORAGE SECTION.
01  WORK-AREA-ONE.
    05  PART1       PICTURE 9(4) COMP.
    05  PART2       PICTURE 9(4) COMP.
    05  PART3       PICTURE 9(4) COMP.
    05  PART4       PICTURE 9(4) COMP.
    05  IPADDRESS.
        10 IPAD1    PICTURE X.
        10 IPAD2    PICTURE X.
        10 IPAD3    PICTURE X.
        10 IPAD4    PICTURE X.
    05  HALFWORD    PICTURE 9(4) COMP.
    05  HALFWORD-X  REDEFINES HALFWORD.
        10 BYTE1    PICTURE X.
        10 BYTE2    PICTURE X.
    05  RESULTS.
        10 RECB     PICTURE X(4).
        10 RLOPORT  PICTURE 9(4) COMP.
        10 RFOPORT  PICTURE 9(4) COMP.
        10 RFOIP    PICTURE X(4).
        10 RCOUNT   PICTURE 9(4) COMP.
        10 RFLAGS   PICTURE X.
        10 RCODE    PICTURE X.
        10 RTERMTY  PICTURE X(40).
    05  MYDESC      PICTURE X(4).
01  LOCAL-PORT      PICTURE 9(4) COMP.
01  BUFFER.
    05  WORKAREA    PICTURE X(512).
```

```
  PROCEDURE DIVISION.
  BEGIN.
 *---------------------------------------*
 *                                       *
 *            First Test                 *
 *                                       *
 *---------------------------------------*
 *
 *    Setup IPADDRESS to hold 172.20.10.10 in binary
 *
      MOVE 172 TO HALFWORD.
      MOVE BYTE2 TO IPAD1.
      MOVE 20  TO HALFWORD.
      MOVE BYTE2 TO IPAD2.
      MOVE 10  TO HALFWORD.
      MOVE BYTE2 TO IPAD3.
      MOVE 10  TO HALFWORD.
      MOVE BYTE2 TO IPAD4.
*
* Attempt to open a connection at 172.20.10.10 port 2000
*
  EXEC TCP OPEN FOREIGNPORT(2000)
                FOREIGNIP(IPADDRESS)
                LOCALPORT(0)
                RESULTAREA(RESULTS)
                DESCRIPTOR(MYDESC)
                ACTIVE
                WAIT(YES)
                ERROR(SECOND-TEST)
      END-EXEC.
      DISPLAY 'Open has completed'.
 *
 *    Receive a piece of data
 *
  EXEC TCP RECEIVE
                TO(BUFFER)
                LENGTH(512)
                RESULTAREA(RESULTS)
                DESCRIPTOR(MYDESC)
                WAIT(YES)
                ERROR(SECOND-TEST)
      END-EXEC.
      DISPLAY 'Receive has completed'.
 *
 *    Close the connection
 *
  EXEC TCP CLOSE
                RESULTAREA(RESULTS)
                DESCRIPTOR(MYDESC)
                ERROR(SECOND-TEST)
      END-EXEC.
      DISPLAY 'Close has completed'.
 *---------------------------------------*
 *                                       *
 *            Second Test                *
 *                                       *
 *---------------------------------------*
```

```
       SECOND-TEST.
      *
      *    Attempt to open a connection
      *
           MOVE 2000 TO LOCAL-PORT.
       EXEC TCP OPEN FOREIGNPORT(0)
                     FOREIGNIP(0)
                     LOCALPORT(LOCAL-PORT)
                     RESULTAREA(RESULTS)
                     DESCRIPTOR(MYDESC)
                     PASSIVE
                     WAIT(YES)
                     ERROR(ERROR-SPOT)
           END-EXEC.
           DISPLAY 'Second Open has completed'.
      *
      *    Display the foreign IP address
      *
           MOVE RFOIP TO IPADDRESS.
           MOVE IPAD1 TO BYTE2.
           MOVE HALFWORD TO PART1.
           MOVE IPAD2 TO BYTE2.
           MOVE HALFWORD TO PART2.
           MOVE IPAD3 TO BYTE2.
           MOVE HALFWORD TO PART3.
           MOVE IPAD4 TO BYTE2.
           MOVE HALFWORD TO PART4.
           DISPLAY PART1 '.' PART2 '.' PART3 '.' PART4
      *
      *    Receive a piece of data
      *
       EXEC TCP SEND
                     FROM(BUFFER)
                     LENGTH(512)
                     RESULTAREA(RESULTS)
                     DESCRIPTOR(MYDESC)
                     WAIT(YES)
                     ERROR(ERROR-SPOT)
           END-EXEC.
           DISPLAY 'Second Receive has completed'.
      *
      *    Close the connection
      *
       EXEC TCP CLOSE
                     RESULTAREA(RESULTS)
                     DESCRIPTOR(MYDESC)
                     ERROR(ERROR-SPOT)
           END-EXEC.
           DISPLAY 'Second Close has completed'.

           STOP RUN.

       ERROR-SPOT.

           STOP RUN.
```

COBOL EXEC FTP Example

```
IDENTIFICATION DIVISION.
      PROGRAM-ID.   CICSFTP.
      AUTHOR.       EBASS.
      DATE-COMPILED.
      ENVIRONMENT DIVISION.
      DATA DIVISION.
       EXEC TCP CONTROL DOUBLE(NO)
                        TRACE(YES)
       END-EXEC.
      WORKING-STORAGE SECTION.
      01  MESSAGES.
          05  WTO         PIC X(60).
      01  SEND-AREA-ONE.
          05  SUSERID     PIC X(32)   VALUE 'CSI'.
          05  SPASSWRD    PIC X(32)   VALUE 'CSI'.
          05  SCMD1       PIC X(8)    VALUE 'QUIT'.
      01  RECV-AREA-ONE.
          05  RUSERID     PIC X(21)
                   VALUE 'Enter Foreign User ID'.
          05  HUSERID     PIC X(21)   VALUE SPACES.
          05  RPASSWRD    PIC X(22)
                   VALUE 'Enter Foreign Password'.
          05  HPASSWRD    PIC X(22)   VALUE SPACES.
          05  RCMD1       PIC X(30)
                   VALUE '220 Service ready for new user'.
          05  HCMD1       PIC X(30)   VALUE SPACES.
          05  IPADDRESS.
              10  IPAD1   PICTURE X.
              10  IPAD2   PICTURE X.
              10  IPAD3   PICTURE X.
              10  IPAD4   PICTURE X.
          05  HALFWORD    PICTURE 9(4) COMP.
          05  HALFWORD-X  REDEFINES HALFWORD.
              10  IPBYTE1   PICTURE X.
              10  IPBYTE2   PICTURE X.
          05  RESULTS.
              10  RECB    PICTURE X(4).
              10  RLOPORT PICTURE 9(4) COMP.
              10  RFOPORT PICTURE 9(4) COMP.
              10  RFOIP   PICTURE X(4).
              10  RCOUNT  PICTURE 9(4) COMP.
              10  RFLAGS  PICTURE X.
              10  RCODE   PICTURE X.
              10  RTERMTY PICTURE X(40).
           05 MYDESC      PICTURE X(4).
       01  LOCAL-PORT     PICTURE 9(4) COMP.
       01  IBUFFER.
           05  IP-WORKI   PICTURE X(32)       VALUE SPACES.
       01  OBUFFER.
           05  IP-WORKA   PICTURE X(512)      VALUE SPACES.
       01  TCP-ECB2       PICTURE X(4).
       PROCEDURE DIVISION.
           MOVE 192 TO HALFWORD.
           MOVE IPBYTE2 TO IPAD1.
           MOVE 168 TO HALFWORD.
           MOVE IPBYTE2 TO IPAD2.
           MOVE 141 TO HALFWORD.
           MOVE IPBYTE2 TO IPAD3.
           MOVE 18  TO HALFWORD.
           MOVE IPBYTE2 TO IPAD4.
```

```
OPEN-FTP.
    MOVE 'API1 - FTP OPEN 1     ' TO WTO.
    EXEC  CICS WRITE OPERATOR TEXT(WTO)
          END-EXEC.
    MOVE 6200 TO LOCAL-PORT.
    EXEC FTP OPEN
                  FOREIGNPORT(21)
                  FOREIGNIP(IPADDRESS)
                  LOCALPORT(LOCAL-PORT)
                  RESULTAREA(RESULTS)
                  DESCRIPTOR(MYDESC)
                  ACTIVE
                  WAIT(YES)
                  ERROR(ERROR-SPOT2)
         END-EXEC.
    PERFORM RECEIVE-IT THRU RECEIVE-IT-EXIT
            UNTIL RUSERID EQUAL HUSERID.
    PERFORM SEND-USER THRU SEND-USER-EXIT.
    PERFORM RECEIVE-IT THRU RECEIVE-IT-EXIT
            UNTIL RPASSWRD EQUAL HPASSWRD.
    PERFORM SEND-SPASSWRD THRU SEND-SPASSWRD-EXIT.
    PERFORM RECEIVE-IT THRU RECEIVE-IT-EXIT
            UNTIL RCMD1 EQUAL HCMD1.
    PERFORM SEND-SCMD1 THRU SEND-SCMD1-EXIT.
    PERFORM CLOSE-FTP.
SEND-USER.
    MOVE 'API1 - FTP SEND SUSER ' TO WTO.
    EXEC  CICS WRITE OPERATOR TEXT(WTO)
          END-EXEC.
    MOVE SUSERID TO IBUFFER.
    PERFORM SEND-IT THRU SEND-IT-EXIT.
SEND-USER-EXIT.
SEND-SPASSWRD.
    MOVE 'API1 - FTP SEND SPASSWRD ' TO WTO.
    EXEC  CICS WRITE OPERATOR TEXT(WTO)
          END-EXEC.
    MOVE SPASSWRD TO IBUFFER.
    PERFORM SEND-IT THRU SEND-IT-EXIT.
SEND-SPASSWRD-EXIT.
SEND-SCMD1.
    MOVE 'API1 - FTP SEND SCMD1' TO WTO.
    EXEC  CICS WRITE OPERATOR TEXT(WTO)
          END-EXEC.
    MOVE SCMD1 TO IBUFFER.
    PERFORM SEND-IT THRU SEND-IT-EXIT.
SEND-SCMD1-EXIT.
SEND-IT.
EXEC FTP SEND
              FROM(IBUFFER)
              LENGTH(32)
              RESULTAREA(RESULTS)
              DESCRIPTOR(MYDESC)
              WAIT(YES)
              ERROR(ERROR-SPOT2)
     END-EXEC.
SEND-IT-EXIT.
```

```
 RECEIVE-IT.
*
* I AM ONLY GUARANTEED 1 BYTE ON THIS RECEIVE
* AND MAY HAVE TO DO MULTIPLE RECEIVES
* IF FIXED AND WAIT(YES) IS USED, THEN I MUST WAIT
* TIL THE ENTIRE BUFFER IS FILLED BEFORE BEING POSTED
*
 MOVE 'API1 - FTP RECEIVE IT' TO WTO.
 EXEC CICS WRITE OPERATOR TEXT(WTO)
      END-EXEC.
 EXEC FTP RECEIVE
                   TO(OBUFFER)
                   LENGTH(512)
                   RESULTAREA(RESULTS)
                   DESCRIPTOR(MYDESC)
                   WAIT(YES)
                   ERROR(ERROR-SPOT2)
      END-EXEC.
 MOVE OBUFFER TO WTO, HUSERID, HPASSWRD, HCMD1.
 EXEC CICS WRITE OPERATOR TEXT(WTO)
      END-EXEC.
 RECEIVE-IT-EXIT.
 CLOSE-FTP.
 MOVE 'API1 - FTP CLOSE 1   ' TO WTO.
 EXEC CICS WRITE OPERATOR TEXT(WTO)
      END-EXEC.
 EXEC FTP CLOSE
                   RESULTAREA(RESULTS)
                   DESCRIPTOR(MYDESC)
                   ERROR(ERROR-SPOT2)
      END-EXEC.
 STOP RUN.
 ERROR-SPOT2.
 MOVE 'API1 - FTP ERROR 1   ' TO WTO.
 EXEC CICS WRITE OPERATOR TEXT(WTO)
      END-EXEC.
 STOP RUN.
```

COBOL EXEC CLIENT LPR Example

```
IDENTIFICATION DIVISION.
       PROGRAM-ID.   CICSLPR.
       AUTHOR.       EBASS.
       DATE-COMPILED.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
        EXEC TCP CONTROL DOUBLE(NO)
                         TRACE(YES)
           END-EXEC.
       WORKING-STORAGE SECTION.
       01  MESSAGES.
           05  WTO         PIC X(60).
       01  SEND-AREA-ONE.
           05  SCMD1 PIC X(32)   VALUE 'LPR'.
           05  SCMD2 PIC X(32)   VALUE 'SET HOST=VM'.
           05  SCMD3 PIC X(32)   VALUE 'SET PRINTER=LOCAL'.
           05  SCMD4 PIC X(32)   VALUE 'CD POWER.LST.L'.
           05  SCMD5 PIC X(32)   VALUE 'PRINT LPRBATCH'.
       01  RECV-AREA-ONE.
           05  RCMD1       PIC X(25)
                    VALUE 'Client manager connection'.
           05  HCMD1       PIC X(25)   VALUE SPACES.
           05  RCMD2       PIC X(10)
                    VALUE 'LPR Ready:'.
           05  HCMD2       PIC X(10)   VALUE SPACES.
           05  RCMD3       PIC X(10)
                    VALUE 'LPR Ready:'.
           05  HCMD3       PIC X(10)   VALUE SPACES.
           05  RCMD4       PIC X(10)
                    VALUE 'LPR Ready:'.
           05  HCMD4       PIC X(10)   VALUE SPACES.
           05  RCMD5       PIC X(10)
                    VALUE 'LPR Ready:'.
           05  HCMD5       PIC X(10)   VALUE SPACES.
           05  IPADDRESS.
               10  IPAD1   PICTURE X.
               10  IPAD2   PICTURE X.
               10  IPAD3   PICTURE X.
               10  IPAD4   PICTURE X.
           05  HALFWORD    PICTURE 9(4) COMP.
           05  HALFWORD-X  REDEFINES HALFWORD.
               10  IPBYTE1   PICTURE X.
               10  IPBYTE2   PICTURE X.
           05  RESULTS.
               10  RECB    PICTURE X(4).
               10  RLOPORT PICTURE 9(4) COMP.
               10  RFOPORT PICTURE 9(4) COMP.
               10  RFOIP   PICTURE X(4).
               10  RCOUNT  PICTURE 9(4) COMP.
               10  RFLAGS  PICTURE X.
               10  RCODE   PICTURE X.
               10  RTERMTY PICTURE X(40).
           05  MYDESC      PICTURE X(4).
       01  LOCAL-PORT      PICTURE 9(4) COMP.
       01  IBUFFER.
           05  IP-WORKI    PICTURE X(32)       VALUE SPACES.
       01  OBUFFER.
           05  IP-WORKA    PICTURE X(80)       VALUE SPACES.
       01  TCP-ECB2        PIC X(4).
       PROCEDURE DIVISION.
           MOVE 'API1 - START CICSLPR ' TO WTO.
           EXEC  CICS WRITE OPERATOR TEXT(WTO)
                 END-EXEC.
           MOVE 192 TO HALFWORD.
           MOVE IPBYTE2 TO IPAD1.
           MOVE 168 TO HALFWORD.
           MOVE IPBYTE2 TO IPAD2.
           MOVE 0   TO HALFWORD.
           MOVE IPBYTE2 TO IPAD3.
           MOVE 7   TO HALFWORD.
           MOVE IPBYTE2 TO IPAD4.

       OPEN-FTP.
           MOVE 'API1 - LPR CLIENT OPEN ' TO WTO.
           EXEC  CICS WRITE OPERATOR TEXT(WTO)
                 END-EXEC.
           MOVE 0 TO LOCAL-PORT.
       EXEC CLIENT OPEN
                         FOREIGNPORT(0)
                         FOREIGNIP(IPADDRESS)
                         LOCALPORT(LOCAL-PORT)
                         RESULTAREA(RESULTS)
                         DESCRIPTOR(MYDESC)
                         ACTIVE
                         WAIT(YES)
                         ERROR(ERROR-SPOT2)
           END-EXEC.
           PERFORM RECEIVE-IT THRU RECEIVE-IT-EXIT
                   UNTIL RCMD1 EQUAL HCMD1.
           PERFORM SEND-SCMD1 THRU SEND-SCMD1-EXIT.
           MOVE SPACES TO WTO, HCMD2, HCMD3, HCMD4, HCMD5.
           PERFORM RECEIVE-IT THRU RECEIVE-IT-EXIT
                   UNTIL RCMD2 EQUAL HCMD2.
           PERFORM SEND-SCMD2 THRU SEND-SCMD2-EXIT.
           MOVE SPACES TO WTO, HCMD3, HCMD4, HCMD5.
           PERFORM RECEIVE-IT THRU RECEIVE-IT-EXIT
                   UNTIL RCMD3 EQUAL HCMD3.
           PERFORM SEND-SCMD3 THRU SEND-SCMD3-EXIT.
           MOVE SPACES TO WTO, HCMD4, HCMD5.
           PERFORM RECEIVE-IT THRU RECEIVE-IT-EXIT
                   UNTIL RCMD4 EQUAL HCMD4.
           PERFORM SEND-SCMD4 THRU SEND-SCMD4-EXIT.
           MOVE SPACES TO WTO, HCMD5.
           PERFORM RECEIVE-IT THRU RECEIVE-IT-EXIT
                   UNTIL RCMD5 EQUAL HCMD5.
           PERFORM SEND-SCMD5 THRU SEND-SCMD5-EXIT.
           PERFORM CLOSE-CLIENT.
       SEND-SCMD1.
           MOVE 'API1 - FTP SEND SCMD1 ' TO WTO.
           EXEC  CICS WRITE OPERATOR TEXT(WTO)
                 END-EXEC.
           MOVE SCMD1 TO IBUFFER.
           PERFORM SEND-IT THRU SEND-IT-EXIT.
       SEND-SCMD1-EXIT.
       SEND-SCMD2.
           MOVE 'API1 - FTP SEND SCMD2 ' TO WTO.
           EXEC  CICS WRITE OPERATOR TEXT(WTO)
                 END-EXEC.
           MOVE SCMD2  TO IBUFFER.
           PERFORM SEND-IT THRU SEND-IT-EXIT.
       SEND-SCMD2-EXIT.
       SEND-SCMD3.
           MOVE 'API1 - FTP SEND SCMD3' TO WTO.
           EXEC  CICS WRITE OPERATOR TEXT(WTO)
                 END-EXEC.
```

```
     MOVE SCMD3 TO IBUFFER.
     PERFORM SEND-IT THRU SEND-IT-EXIT.
 SEND-SCMD3-EXIT.
 SEND-SCMD4.
     MOVE 'API1 - FTP SEND SCMD4 ' TO WTO.
     EXEC  CICS WRITE OPERATOR TEXT(WTO)
           END-EXEC.
     MOVE SCMD4  TO IBUFFER.
 PERFORM SEND-IT THRU SEND-IT-EXIT.
 SEND-SCMD4-EXIT.
 SEND-SCMD5.
     MOVE 'API1 - FTP SEND SCMD5' TO WTO.
     EXEC  CICS WRITE OPERATOR TEXT(WTO)
           END-EXEC.
     MOVE SCMD5 TO IBUFFER.
     PERFORM SEND-IT THRU SEND-IT-EXIT.
 SEND-SCMD5-EXIT.
 SEND-IT.
 EXEC CLIENT SEND
                   FROM(IBUFFER)
                   LENGTH(32)
                   RESULTAREA(RESULTS)
                   DESCRIPTOR(MYDESC)
                   WAIT(YES)
                   ERROR(ERROR-SPOT2)
      END-EXEC.
 SEND-IT-EXIT.
 RECEIVE-IT.
* I AM ONLY GUARANTEED 1 BYTE ON THIS RECEIVE
* AND MAY HAVE TO DO MULTIPLE RECEIVES
* IF FIXED AND WAIT(YES) IS USED, THEN I MUST WAIT
* TIL THE ENTIRE BUFFER IS FILLED BEFORE BEING POSTED
 MOVE 'API1 - CLIENT RECEIVE IT' TO WTO.
     EXEC  CICS WRITE OPERATOR TEXT(WTO)
           END-EXEC.
 EXEC CLIENT RECEIVE
                   TO(OBUFFER)
                   LENGTH(80)
                   RESULTAREA(RESULTS)
                   DESCRIPTOR(MYDESC)
                   WAIT(YES)
                   ERROR(ERROR-SPOT2)
     END-EXEC.
 MOVE OBUFFER TO WTO, HCMD1,HCMD2,HCMD3,HCMD4,HCMD5.
     EXEC  CICS WRITE OPERATOR TEXT(WTO)
           END-EXEC.
 RECEIVE-IT-EXIT.
 CLOSE-CLIENT.
     MOVE 'API1 - CLIENT CLOSE    ' TO WTO.
     EXEC  CICS WRITE OPERATOR TEXT(WTO)
           END-EXEC.
 EXEC CLIENT CLOSE
                   RESULTAREA(RESULTS)
                   DESCRIPTOR(MYDESC)
                   ERROR(ERROR-SPOT2)
     END-EXEC.
     STOP RUN.
 ERROR-SPOT2.
     MOVE 'API1 - CLIENT ERROR    ' TO WTO.
     EXEC  CICS WRITE OPERATOR TEXT(WTO)
           END-EXEC.
     STOP RUN.
```

PL/1 EXEC TCP Example

```
SAMPLE4: PROCEDURE OPTIONS(MAIN);

 DCL IPADDRESS     BINARY FIXED(31,0);
 DCL MYDESC        CHAR(4);
 DCL 1  RESULTS,
        2 RECB     CHAR(4),
        2 RLOPORT  BINARY FIXED(15,0),
        2 RFOPORT  BINARY FIXED(15,0),
        2 RFOIP    CHAR(4),
        2 RCOUNT   BINARY FIXED(15,0),
        2 RFLAGS   CHAR(1),
        2 RCODE    BIT(8),
        2 RTERMTY  CHAR(40);
 DCL MYDESC        CHAR(4);
 DCL LOCAL_PORT    BINARY FIXED(15,0);
 DCL BUFFER        CHAR(512);

/*---------------------------------------*
 *                                       *
 *            First Test                 *
 *                                       *
 *---------------------------------------*/
/*
 * Attempt to open a connection at 172.20.10.10 port 2000
 */
       EXEC TCP OPEN FOREIGNPORT(2000)
                     FOREIGNIP(IPADDRESS)
                     LOCALPORT(0)
                     RESULTAREA(RESULTS)
                     DESCRIPTOR(MYDESC)
                     ACTIVE
                     WAIT(YES)
                     ERROR(SECOND_TEST)
            END-EXEC;
/*
 *    Receive a piece of data
 */
       EXEC TCP RECEIVE
                     TO(BUFFER)
                     LENGTH(512)
                     RESULTAREA(RESULTS)
                     DESCRIPTOR(MYDESC)
                     WAIT(YES)
                     ERROR(SECOND_TEST)
            END-EXEC;
/*
 *    Close the connection
 */
       EXEC TCP CLOSE
                     RESULTAREA(RESULTS)
                     DESCRIPTOR(MYDESC)
                     ERROR(SECOND_TEST)
            END-EXEC;
```

```
SECOND_TEST:
/*---------------------------------------*
 *             Second Test               *
 *---------------------------------------*
 *    Attempt to open a connection
 */
           LOCAL_PORT = 2000;
       EXEC TCP OPEN FOREIGNPORT(0)
                     FOREIGNIP(0)
                     LOCALPORT(LOCAL_PORT)
                     RESULTAREA(RESULTS)
                     DESCRIPTOR(MYDESC)
                     PASSIVE
                     WAIT(YES)
                     ERROR(ERROR_SPOT)
            END-EXEC;
/*
 *    Display the foreign IP address
 */

/* Need code here..... */
/*
 *    Receive a piece of data
 */
       EXEC TCP SEND
                     FROM(BUFFER)
                     LENGTH(512)
                     RESULTAREA(RESULTS)
                     DESCRIPTOR(MYDESC)
                     WAIT(YES)
                     ERROR(ERROR_SPOT)
            END-EXEC;
/*
 *    Close the connection
 */
       EXEC TCP CLOSE
                     RESULTAREA(RESULTS)
                     DESCRIPTOR(MYDESC)
                     ERROR(ERROR_SPOT)
            END-EXEC;
RETURN;
 END SAMPLE4;
```

PL/1 Notes

To use the TCP/IP interface with PL/1 for VSE/ESA for a program originally written for DOS/VS PL/1, you must note an important difference between DOS/VS PL/1 and PL/1 for VSE/ESA. The working storage used by DOS/VSE PL/1 generally was pre-initialized, but the working storage for PL/1 for VSE/ESA is not. To ensure that programs originally written for DOS/VS PL/1 using the TCP/IP FOR VSE preprocessor API continue to work in PL/1 for VSE/ESA, you must use the following options on LE/VSE for these programs:

```
BATCH: STORAGE(00,NONE,00,64K)
CICS:  STORAGE(00,NONE,00,0K)
```

For more information on this topic and these options, see the *PL/I VSE Migration Guide*, IBM Manual SC26-8056-01.

4

REXX Sockets API

Overview

This chapter describes the REXX Sockets application programming interface (API). This interface is similar to the other APIs in the following ways:

- It enables you to write programs that interface with TCP/IP FOR VSE.
- It enables you to use TCP/IP FOR VSE services to communicate with other TCP applications on the TCP/IP network.

REXX Sockets is an excellent prototyping tool for coding TCP/IP applications.

Note: When a SOCKET OPEN executes, the API outputs two lines of release information on SYSLOG. If you want to suppress these lines, enable UPSI-1 in that job stream. This is true only if you are not running the SOCKET API under the TCP/IP stack as part of a REXX-CGI session.

REXX Calls

The calls you can make to REXX Sockets closely mirror the calls you can make with the higher language interfaces discussed in the previous chapters. REXX Sockets is enhanced with some REXX extensions, including those that read and set variables.

The REXX calls perform the following tasks. Each call's syntax is described later in this chapter

Call	Description
ABORT	Immediately terminates the connection with the foreign host.
CLOSE	Terminates the connection between the REXX program and the remote application.
OPEN	Establishes a socket connection with TCP/IP FOR VSE. If a client issues the call, it connects to a server. If a server issues the call, it waits for a connection from a client.
RECEIVE	Accepts data from the remote application and makes it available to your REXX program.
SEND	Passes data from the REXX program to the remote application.
STATUS	Allows you to check the status of a specific TCP/IP connection. This is useful for programs that function as servers.

Variables

REXX calls set the REXX variables that are described in the following table:

Variable	Description
handle	An output field containing a special pointer that is returned from an OPEN call. This value is used as an input field in subsequent calls. You can use *handle* as the input name or you can use a different name that contains the handle value. You may choose to copy this value when you issue multiple socket OPENs in the same program.
buffer	An output field containing data returned from a RECEIVE call.

Variable	Description
errmsg	An output field containing an error message. If an error occurs, REXX Sockets puts the message in this variable instead of sending it to SYSLOG or SYSLST. It is your responsibility as a programmer to issue the message, say errmsg, if there is a problem. If an error occurs while errmsg is being set, the message is routed to SYSLOG.
loip	An output field containing the local IP address that is returned from a STATUS call. If you need to know the local IP address before a connection is established, you can use a control connection. Note that in a multi-homing environment, the local IP address that is obtained from STATUS represents the IP address that is used for the connection. If you want the local IP address as defined by the SET IPADDR command in the TCP/IP FOR VSE initialization deck, you must use the control connection to obtain it.
foip	An output field containing the foreign IP address. It is created or updated during a synchronous OPEN. If you are using an asynchronous OPEN and a connection has been established, then subsequent STATUS calls update *foip*.
loport	An output field containing the port number used by the local TCP/IP system. Note that this may or may not be the local port number that your application requested.
foport	An output field containing the port number used by the foreign TCP/IP system. It is created during a synchronous OPEN. If you use an asynchronous OPEN, it is created or updated by STATUS.
connstate	An output field containing the current connection status. It is returned from a STATUS call. The values contained in *connstate* have the following meanings: • If *connstate* is 1, the connection represented by *handle* is a server application that is in a listening state. In other words, it is waiting for someone to connect to it. • If *connstate* is 4, a connection has been established. The variables *loip, foip, loport,* and *foport* contain valid data so that your program knows who connected to it and on which port. • If *connstate* is anything other than 1 or 4, then the state is considered transitory, meaning that the socket is no longer waiting for a connection and it is no longer connected. If your program is to continue, it must close and reopen the socket.

The following table shows how the REXX variables are used by the REXX calls. Note the following indicators in the table:

- *input* means that the variable is used as input to the call.
- *output* means that the variable is used as output from the call.
- *not used* means that the variable is not used by the call.

Variable	OPEN	CLOSE	SEND	RECEIVE	ABORT	STATUS
handle	output	input	input	input	input	input
buffer	not used	not used	not used	output	not used	not used
errmsg	output	output	output	output	output	output
loip	not used	not used	not used	not used	not used	output
foip	output	not used	not used	not used	not used	output
loport	not used	not used	not used	not used	not used	output
foport	output	not used	not used	not used	not used	output
connstate	not used	not used	not used	not used	not used	output

Return Codes

The following return codes apply to the OPEN, CLOSE, SEND, RECEIVE, and ABORT REXX calls.

Return Code	Description
0	Call was successful.
4	Timeout occurred.
8	Unspecified error. The variable *errmsg* contains the error text.
12	Foreign IP address is unavailable.
16	Local TCP/IP system is down.
20	Unregistered version.

The STATUS REXX call uses the return codes in the following table:

Return Code	Description
0	Call was successful. The variable *connstate* contains additional information. If *connstate* is 4 (meaning that you have a valid connection), then the variables *loip, foip, loport*, and *foport* contain valid information.
4	Call failed. This probably means that the variable *handle* is not valid.

Timeout Function

Each time you use a socket OPEN to start a new connection, you can set a timeout value for that connection. The timeout value remains in effect for the life of the connection. If you do not specifically set a timeout value, the default value is used. You can have multiple connections open at one time, and each connection uses the timeout value that was set during its open.

Socket Types

REXX Sockets allows you to code the socket types described in the following table. You specify the socket type on the OPEN. The Auto field in this table indicates whether a socket uses automatic translation.

Socket Type	**Description**	**Auto**
CLIENT	A connection to the generic TCP/IP FOR VSE client manager. The generic manager supports the Ping, LPR, TRACERT, REXEC, EMAIL, and DISCOVER clients. For more information, see "Obtaining Network Information" on page 113. For an example of a REXX Sockets application that uses a client connection, see member REXXPING.Z in the TCP/IP distribution library.	No
CONTROL	A connection to a TCP/IP partition. You use the connection to obtain information from the partition. For more information, see Obtaining Network Information. For an example of a REXX Sockets application that uses a control connection, see member REXXCONT.Z in the TCP/IP distribution library.	No
FTP	A connection to a TCP/IP FOR VSE FTP client manager. The client manager monitors an FTP session under control of your program. Your program uses SEND to issue commands to the client manager in the same way that you issue commands to a TCP/IP FOR VSE FTP client. Your program uses RECEIVE to obtain responses, which you can analyze by looking at information returned in variable *buffer*. Your program must follow standard FTP protocol, which means that you must know when data is available and when to issue a RECEIVE. Failure to follow FTP protocol causes unpredictable and usually negative results. For a list of FTP client commands, see the *TCP/IP FOR VSE User Guide*. For an example of a REXX Sockets application that uses an FTP socket, see member REXXFTP.Z in the TCP/IP distribution library.	Yes
TCP	A connection to a TCP application. Your program and the TCP application must use the same protocol. This means that your socket application and the partner socket application must agree on communication specifications. For example, your application must understand when it has received an entire transmission by looking at the data it receives.	No

Socket Type	Description	Auto
TELNET	A connection to a TCP/IP FOR VSE telnet client manager. The client manager monitors a telnet session under control of your program. This socket type expects you to connect to a telnet daemon at the remote end. After you connect, you issue telnet commands and interrogate the responses. TCP/IP FOR VSE always negotiates a terminal type of Network Virtual Terminal (NVT). For all SEND calls, X'15' is automatically appended to the end of the data unless the data already ends with X'15'. For an example of a REXX Sockets application that uses a telnet socket, see member REXXTEL.Z in the TCP/IP distribution library.	Yes
UDP	A connection to a UDP application. Your program and the UDP application must use the same protocol. This means that your socket application and the partner socket application must agree on communication specifications. For example, your application must understand when it has received an entire transmission by looking at the data it receives.	No

Coding REXX Calls

In this section we explain how to issue each call. We also show the syntax of each call and describe the parameters you can use. The parameters shown for each call are positional. This means that you must use a comma to hold the space if you omit a parameter.

OPEN

The OPEN call establishes a socket connection with TCP/IP FOR VSE. The syntax is as follows:

```
rc=SOCKET(type,'OPEN',loport,foip,foport,sysid,timeout,
          async,mode)
```

The variables are described in the following table:

Variable	Description
type	Socket type. Specify a type from the "Socket Types" section on page 106.
loport	Local port number. For a server function, *loport* specifies the port number you plan to listen to. This is ignored for CLIENT and CONTROL sessions.
foip	Foreign IP address. For a client function, *foip* specifies the IP address you plan to connect to. For a server function, *foip* specifies a mask. TCP/IP FOR VSE uses the mask to validate the IP address when it connects to your server. The validation rules are the same rules that apply to parameter IPADDR= on the DEFINE FTPD and DEFINE TELNETD commands. See the *TCP/IP FOR VSE Installation Guide* for more information.
foport	Foreign port address. If *foport* is coded, it specifies the port number you plan to connect to at the remote end. If *foport* is omitted and you are coding a client function, the default is *loport*. For a server function, omit *foport*. If you omit *foip*, you must also omit *foport*.
sysid	Two-digit ID of the TCP/IP FOR VSE partition. The default is 00.
timeout	Time allowed for the operation to complete. The value is the number of 300th-second intervals. If a connection is not established within this period, a timeout occurs. The default is 36000, or 2 minutes.

Variable	Description
async	Asynchronous indicator. Specify “N,” which is the default, for a synchronous OPEN. Specify “N” if you are coding a server function and you want your REXX program to wait until a client actually connects to the socket. Specify “N” if you are coding a client function. Specify “Y” if you are coding a server function and you want control immediately after TCP/IP FOR VSE acknowledges your socket.
mode	Type of OPEN. Specify CLIENT to issue an active open and to attempt to connect to the server specified by *foip* and *foport*. Specify SERVER to issue a passive open and to wait (synchronously or asynchronously) for a client to connect to you.

Your REXX program can open multiple sockets. The variable *handle* uniquely identifies each socket to TCP/IP FOR VSE. When you open multiple sockets, you must save *handle* after each OPEN. You must use the correct *handle* for each subsequent function, such as SEND or RECEIVE. In the examples that follow, we use the standard variable name *handle*. You can, however, use a different variable name after you copy the data to that name. If you are not opening multiple sockets, then let REXX Sockets set *handle* and use it as shown in the examples.

You can specify servers only for UDP, TCP, and TELNET sessions.

CLOSE

The CLOSE call terminates a socket connection with TCP/IP FOR VSE. The syntax is as follows:

```
rc=SOCKET(handle,'CLOSE',timeout)
```

The variables are described in the following table:

Variable	Description
handle	Identifying variable returned from a successful OPEN.
timeout	Time allowed for the operation to complete. The value is the number of 300th-second intervals. If this period elapses and the connection has not terminated, a timeout occurs. The default is the value used in the OPEN call.

SEND

The SEND call passes data from the REXX program to the remote application. The remote application can be a partner TCP or UDP application, or it can be a control, client, FTP or telnet connection manager. The syntax is as follows:

```
rc=SOCKET(handle,'SEND',data,timeout)
```

The variables are described in the following table:

Variable	Description
handle	Identifying variable returned from a successful OPEN.
data	The data you want to send.
timeout	Time allowed for the operation to complete. The value is the number of 300th-second intervals. If this period elapses and the connection has not terminated, a timeout occurs. The default is the value used in the OPEN call.

If SEND times out, issue a STATUS call to verify that the connection is still valid. If SEND completes normally, it does not necessarily indicate that the data was sent successfully. It simply means the data is queued and will be sent by the TCP/IP FOR VSE partition.

REXX Sockets automatically appends X'15' to the end of your data if both of the following are true:

1. The data does not already end with X'15' or a X'00', and
2. The connection type is CONTROL, CLIENT, FTP, or TELNET.

The X'15' is required because the listed components expect the commands to end in an EBCDIC CR/LF.

The *timeout* value is only useful for TCP, UDP, or TELNET connections.

RECEIVE

The RECEIVE call accepts data from the remote application and makes it available to your REXX program. The syntax is as follows:

```
rc=SOCKET(handle,'RECEIVE',timeout)
```

The variables are described in the following table:

Variable	Description
handle	Identifying variable returned from a successful OPEN.
timeout	Time allowed for the operation to complete. The value is the number of 300th-second intervals. If this period elapses and the connection has not terminated, a timeout occurs. The default is the value used in the OPEN call.

The output variable *buffer* contains the data that is received. If RECEIVE times out, *buffer* is saved with the null value. The null value is a length of zero.

The *timeout* value is only useful for TCP or TELNET connections.

ABORT

The ABORT call immediately terminates the connection with the foreign host. The syntax is as follows:

```
rc=SOCKET(handle,'ABORT',timeout)
```

The variables are described in the following table:

Variable	Description
handle	Identifying variable returned from a successful OPEN.
timeout	Time allowed for the operation to complete. The value is the number of 300th-second intervals. If this period elapses and the connection has not terminated, a timeout occurs. The default is the value used in the OPEN call.

Note the differences between CLOSE and ABORT:

- When you issue a CLOSE call, TCP/IP FOR VSE closes the connection gracefully. To do this, it announces that it is about to close the connection. The announcement enables the application on the other end to respond appropriately, which could mean that it shuts down or that it performs cleanup processing. TCP/IP FOR VSE then closes the connection.

- When you issue an ABORT call, TCP/IP FOR VSE terminates the connection immediately and then informs the application on the other end that the connection is closed.

Use ABORT only for failing TCP or TELNET sessions and for no other, and only during conditions where a CLOSE is ineffective.

STATUS

The STATUS call allows you to check the status of a specific TCP/IP connection. You can also use it to verify that the last socket operation your program performed is complete. The syntax is as follows:

```
rc=SOCKET(handle,'STATUS')
```

The variable is described in the following table.

Variable	**Description**
handle	Identifying variable returned from a successful OPEN.

Obtaining Network Information

You can make three types of calls to TCP/IP FOR VSE to obtain network information:

Call	Description
GETHOSTNAME	Returns with the name of the TCP/IP FOR VSE partition. No input parameters are required.
GETHOSTBYNAME	Translates a name into an IP address. The source of the IP address is the TCP/IP FOR VSE Domain Name Client (if one is specified with the SET DNS1 command) or the Internal Domain Name Table (if one is specified with the DEFINE NAME command). GETHOSTBYNAME takes one parameter—the symbolic name to be translated.
GETHOSTID	Returns with the IP address of the TCP/IP FOR VSE partition. This is the IP address as defined by the SET IPADDR= initialization command, and it does not take multi-homing considerations into account. If you want the local IP address that is currently used for a specific connection in a multi-homing environment, you must issue the SOCKET STATUS call after the connection is established.

Before you can use these calls, you must set up a REXX Sockets control session with the TCP/IP FOR VSE partition.

Starting a Control Connection

The following program shows how to start a control connection.

```
/* Open a control connection to the TCP/IP for VSE partition*/
rc = SOCKET('CONTROL','OPEN') if rc \= 0 then say 'rc='rc 'errmsg=' errmsg

/* Send the command string. Note that for control connections, TCP/IP for VSE
automatically includes X'15' at the end of the string */
rc = SOCKET(handle,'SEND','GETHOSTID')
if rc \= 0 then say 'rc='rc 'errmsg=' errmsg

/* Receive the response from TCP/IP for VSE */
rc = SOCKET(handle,'RECEIVE') if rc \= 0 then say 'rc='rc 'errmsg=' errmsg

/* Close the control connection */
rc = SOCKET(handle,'CLOSE')
if rc \= 0 then say 'rc='rc 'errmsg=' errmsg
```

Starting a Client Connection

You can also open a connection to the TCP/IP FOR VSE client manager. The client manager currently supports the Ping, LPR, TRACERT, DISCOVER, REXEC, and EMAIL clients. To use these functions, you must understand the command sequences the client manager expects. The example below shows how a REXX program uses Ping.

```
/* Open a client connection to the TCP/IP for VSE partition */
rc = SOCKET('CLIENT','OPEN') if rc \= 0 then say 'rc='rc 'errmsg=' errmsg

/* The first interaction with the client connection must be the name of the
protocol you want to use, which is either PING or LPR */
rc = SOCKET(handle,'SEND','PING')
if rc \= 0 then say 'rc='rc 'errmsg=' errmsg

/* Receive the response from TCP/IP for VSE. You might receive multiple lines
of response so you must loop until you receive the message PING Ready:. Send
each line to SYSLST in the meantime. */
Do forever
   rc = SOCKET(handle,'RECEIVE')    if rc \= 0 then do ;  say 'rc='rc 'errmsg='
errmsg ; exit
   if pos('PING Ready:',buffer) \= 0 then leave
   say buffer
end
/* Send a command to the Ping client */
rc = SOCKET(handle,'SEND','SET HOST=192.168.0.7')

/* Receive the results. The responses to the SET HOST command are a symbolic
representation of the IP address followed by the PING Ready prompt  */
Do forever
   rc = SOCKET(handle,'RECEIVE')    if rc \= 0 then do ;  say 'rc='rc 'errmsg='
errmsg ; exit
   if pos('PING Ready:',buffer) \= 0 then leave
   say buffer
end
/* Send the Ping command to make TCP/IP for VSE really do a Ping  */
rc = SOCKET(handle,'SEND','PING')

/* Analyze the response. You always receive 5 lines. Each line says that the
Ping was successful or that it failed. If any Ping fails, we report it. Else,
we avg Ping response times and report that */
Totms = 0
Do I = 1 to 5
   rc = SOCKET(handle,'RECEIVE')
   If pos('timeout',buffer) \=0 then,
   Do
      Say 'A ping request has timed out'
      Say buffer
      Exit
   End
   Parse var buffer 'Milliseconds:'ms
   Totms = totms + ms
End
Say 'Average Ping Response time was' totms/5

/* Close the client connection */
rc = SOCKET(handle,'CLOSE')
if rc \= 0 then say 'rc='rc 'errmsg=' errmsg
```

You can code LPR jobs in the same way, but you must ensure that the responses you code for are ones you receive. As with other automated processes in your data center, responses can change with routine maintenance.

5

Common Gateway Interfaces

Overview

This chapter explains how to program a Common Gateway Interface, or CGI. A CGI program is invoked by the TCP/IP FOR VSE HTTP daemon in response to a request from a web browser. It is responsible for returning the next webpage to the web browser. For information on defining the HTTP daemon, see the *TCP/IP FOR VSE Installation Guide*.

The browser passes an assortment of parameters through the HTTP daemon (HTTPD) and into the CGI. The CGI performs the requested activity and returns a series of Hypertext Markup Language (HTML) statements. The HTML statements enable the web browser to display an attractive screen of information that can include graphics, sounds, animation, and other elements.

Remember that webpages use HTML, but not all webpages use CGIs. A simple webpage that displays static information doesn't require a CGI. Static information is information that does not change very often, such as a list of employees within your company. In a case like this, the information is stored one time. To access the information, the client clicks on a person's name. This action passes a GET request to the web server, which tells it to load another webpage that provides even more information. The web server obtains the webpage and passes it back to the web browser for display.

A CGI program is required when you need to display dynamic information. Dynamic information is information that changes every time a webpage is displayed. For example, many webpages display a counter when they are accessed. You've probably seen a webpage that said something like, "The number of people that have accessed this page since January 1 is 3,123." How does this work? There is usually a small file on the server that contains a number and a date. When you access the webpage, it invokes a CGI that reads the file, increments the number, saves the new value, and includes this information on the webpage display.

Using CGIs with VSE

TCP/IP FOR VSE implements a VSE type of approach to the CGI process. For example, CGIs may be written in VSE languages such as assembler and REXX. These program types are covered separately in this chapter. The CGI facility takes advantage of VSE concepts such as multiprocessing and can process requests quickly.

Assembler and REXX CGI programs run under the TCP/IP FOR VSE stack, in the same partition. If you write a program that can disrupt the stack, for example, one that uses an MWAIT macro or performs heavy I/O functions, you should run the CGI external to the stack using the CGILOAD utility. See “Using the CGILOAD Utility” below.

If a CGI runs under the stack, do not issue macros such as SETIME. If you need to issue a “WAIT,” then the program should be written in assembler and defined with a “TYPE=CGI-BAL.” The control block passed to the assembler routine contains the address to the stack’s own WAIT routine, which you should use instead of invoking the IBM WAIT macro. As part of the call, a pointer to the internal WAIT routine within the stack is passed for CGI-BAL to use.

Defining a CGI to VSE

Before you use a CGI, you need to define it to VSE using the DEFINE CGI command. The syntax is as follows:

```
DEFINE CGI,PUBLIC='name'[,TYPE=cgitype]
```

The parameters are described in the following table:

Parameter	Description
'*name*'	The phase name for programs, or the PROC name for REXX
cgitype	Specify CGI-BAL for an assembler program (the FILEIOHD macro is not used), CGI-REXX for a REXX program, or CGI for assembler CGIs that use the TCP/IP FOR VSE file I/O driver design requirements.

The program is not loaded into the CGI partition until a web browser calls it.

Using the CGILOAD Utility

The CGILOAD utility enables CGIs to run in an external partition. It loads the CGI module into storage and opens a SOCKET to HTTPD. When HTTPD needs to call a CGI, it first checks to see whether the CGI is external to the stack. An external CGI takes precedence, so a CGI loaded in the external partition is the one used. This allows you to test and debug a CGI before letting it run under the stack.

The syntax is as follows:

```
// EXEC CGILOAD,PARM='[SYSID=00|xx][,PORT=portnum]'
```

The parameters are described in the following table:

Parameter	**Description**
xx	An alternate stack ID. The default is 00.
portnum	A specific port number. This can be set if there is a conflict with other servers. By default, a port is selected automatically.

CGILOAD stores a port number in the specified partition. When HTTPD scans partitions, it checks for a running CGILOAD. If one is found, HTTPD checks the port that was opened, connects to it, and tells CGILOAD to load and run the CGI there. All CGIs run in single-thread mode. If multiple CGILOADs are running and you call a REXX CGI, HTTPD forces it to a single thread. It does not do this for assembler CGIs.

Deleting a CGI

When you delete a CGI, you are deleting the module from storage. You are not deleting the CGI definition. The next time the program is invoked it is loaded into storage once again. This is useful when you want to refresh a CGI. The syntax is as follows:

```
DELETE CGI,PUBLIC='name'
```

The parameter is described in the following table:

Parameter	**Description**
'*name*'	The phase name for assembler programs, or the PROC name for REXX.

Assembler CGIs

If you plan to code a 31-bit assembler CGI, we recommend that you use the parameter list documented in this section. It reduces the coding effort significantly. The parameter list is mapped with the CGIDATA macro, which is contained in PRD2.TCPIP. The CGIDATA DSECT is shown below.

```
CGIDATA DSECT ,
CGIID    DC   CL6'CGBLOK'       <--- Eyecatcher
CGIFPO   DC   H'0'              <--- FPORT value
CGINAME  DC   CL16' '           <--- Incoming user ID
CGIPASS  DC   CL16' '           <--- Incoming password
CGIILEN  DC   F'0'              <--- Input data length
CGIOLEN  DC   F'0'              <--- Output data length
CGIIPT   DS   AL4               <--- Input area pointer
CGIOPT   DS   AL4               <--- Output area pointer
CGIACT   DC   F'0'              <--- Actual data area size
CGIFIP   DC   F'0'              <--- FOIP value
CGILIP   EQU  *                 <--- LOIP value
CGIDLENG EQU  *-CGINAME         <--- Length of the area
```

Example

The CGI program below is coded in 31-bit assembler language.

```
* --------------------------------------------------------- *
* Program: CGIIPLD2                                         *
*                                                           *
* Purpose: Example of an assembler CGI program.             *
*                                                           *
* Input: R2 = Pointer to control block pointer              *
*        Parms: See the CGIDATA control block layout        *
*                                                           *
* Output: CGIOLEN indicates the returned length             *
*         CGIOPT points to the returned data                *
*                                                           *
* CGILOAD needs to be active for this CGI to work.          *
* --------------------------------------------------------- *
         PUNCH ' PHASE CGIIPLD2,*'     Name of the CGI
CGIIPLD2 CSECT ,                       Start of the subroutine
         USING *,RF                    Addressability
         SAVE (14,12)                  Save incoming registers
         LR   RC,RD                    Copy the savearea
         L    RD,=A(SAVEAREA)          Point to new area
         ST   RD,8(RC)                 Save backward pointer
         ST   RC,4(RD)                 Save forward pointer
         LR   RC,RF                    Get the base
         DROP RF                       No longer needed
         USING CGIIPLD2,RC             New base
         USING CGIPARM,R2              And map to the layout
         L    R2,0(R2)                 Point to the data
*
* If this is the first time in, display some info and setup the value.
*
         CLI   BODYJ,C' '              First time in ?
         BNZ   GETLEN                  No...proceed
         GETFLD FIELD=IPLTIME          Get the IPLTIME STCK value       00032000
         GETIME CLOCK=YES              Convert to a usable format       00033000
         STCM  RE,12,MMDDYY            Get the month                    00034000
         STCM  RE,3,MMDDYY+3           And the day                      00035000
         STCM  RF,12,MMDDYY+6          And the year                     00036000
         ST    R1,TIME                 Save the time                    00037000
         ED    HHMMSS(L'HHMMSS),TIME   Unpack/format the time           00038000
         MVC   HMS(8),HHMMSS+1         Put it in the message            00039000
         MVC   BODYJ(40),MSG           Insert the message here...       00040000
         MVC   BODYNJ(40),MSG          And here too                     00041000
*                                                                       00042000
* Now we'll check the data passed                                       00043000
*                                                                       00044000
GETLEN  CLC   CGIILEN(4),=F'0'          Is there data ?                 00045000
        BZ    DODEF                     No...do the default             00046000
        L     R5,CGIIPT                 Point to the data               00047000
        CLI   5(5),C'Y'                 Java request ?                  00048000
        BZ    DOJAVA                    Yes...proceed                   00049000
        CLI   5(5),C'N'                 No Java request ?               00050000
        BZ    DOTEXT                    Yes...proceed                   00051000
DODEF   MVC   CGIOLEN(4),DEFLEN         Get the length                  00052000
        MVC   CGIOPT(4),=AL4(DEFPAGE)   Point to the data               00053000
        B     EXIT                      Return to HTTPD                 00054000
*                                                                       00055000
* If JAVA=YES, then display info in Java format                         00056000
*                                                                       00057000
DOJAVA  MVC   CGIOLEN(4),JAVALEN        Get the length                  00058000
        MVC   CGIOPT(4),=AL4(JAVA)      Point to the data               00059000
        B     EXIT                      Return to HTTPD                 00060000
*                                                                       00061000
* If JAVA=NO, then display info in text format                          00062000
*                                                                       00063000
DOTEXT  MVC   CGIOLEN(4),TEXTLEN        Get the length                  00064000
        MVC   CGIOPT(4),=AL4(NOJAVA)    Point to the data               00065000
*                                                                       00066000
EXIT    L    RD,4(RD)                   Regain savearea                 00067000
        LM   RE,RC,12(RD)               Regain the regs                 00068000
        XR   RF,RF                      Clear the return code           00069000
        BR   RE                         Return to caller                00070000
* ---------------- Data Area -------------------- *                     00071000
         LTORG  ,                                                       00072000
SAVEAREA DS    9D                                                       00073000
TIME     DS    F                         HHMMSSF RETURNED HERE          00074000
HHMMSS   DC    XL9'2120207A20207A2020'                                  00075000
*                                                                       00076000
MSG      DC    CL20'System was IPLed on '                               00077000
MMDDYY   DC    CL8'MM/DD/YY'                                            00078000
         DC    CL4' at'                                                 00079000
HMS      DC    CL8'HH:MM:SS'                                            00080000
*                                                                       00081000
DEFPAGE  DC   C'<HTML><HEAD><TITLE>CGIIPLD2 Test Page</TITLE>'          00082000
         DC   C'</HEAD><BODY><FORM ACTION="CGIIPLD2" METHOD=GET">'      00083000
         DC   C'<CENTER><H1>CGIIPLD2 Test Page</H1><H2><I>'             00084000
         DC   C'Perform one of the following tests:<HR></I>'            00085000
         DC   C'Cursor Selection Test<BR><BR>Pass the'                  00086000
         DC   C'<A HREF="CGIIPLD2?JAVA=NO"><I>NOSCRIPT</I></A>'         00087000
         DC   C'or the <A HREF="CGIIPLD2?JAVA=YES"><I>SCRIPT'           00088000
         DC   C'</I></A>parameter to CGIIPLD2<HR></I><H2>'              00089000
         DC   C'Form SUBMIT Test<BR><I><BR>Select an option and'        00090000
         DC   C'press the Test button that follows:<BR></I></H2>'       00091000
         DC   C'<B><INPUT TYPE="radio" NAME="JAVA" VALUE="NO">'        00092000
         DC   C'NOSCRIPT<INPUT TYPE="radio" NAME="JAVA" '              00093000
         DC   C'VALUE="YES"> SCRIPT</B>'                               00094000
         DC   C'<INPUT TYPE="submit" name="SUBMIT" value="Test">'      00095000
         DC   C'<HR></FORM></BODY></HTML>'                             00096000
DEFLEN   DC   AL4(*-DEFPAGE)                                           00097000
*                                                                      00098000
JAVA     DC   C'<HTML><HEAD><TITLE>CGIIPLD2 Test Page</TITLE>'         00099000
         DC   C'</HEAD><BODY>'                                         00100000
         DC   C'<SCRIPT LANGUAGE="JavaScript">'                        00101000
         DC   C'<!-- Hide from non-JAVA browsers',X'0D25'              00102000
         DC   C'alert("'                                               00103000
BODYJ    DC   CL40' '                                                  00104000
         DC   C'");',X'0D25'                                           00105000
         DC   C'// Stop Hiding -->',X'0D25'                            00106000
         DC   C'</SCRIPT>'                                             00107000
         DC   C'<A HREF="../CGIIPLD2">Return to menu'                  00108000
         DC   C'</A></BODY></HTML>'                                    00109000
JAVALEN  DC   AL4(*-JAVA)                                              00110000
*                                                                      00111000
NOJAVA   DC   C'<HTML><HEAD><TITLE>CGIIPLD2 Test Page</TITLE>'         00112000
         DC   C'</HEAD><BODY><H1><HR>'                                 00113000
BODYNJ   DC   CL40' '                                                  00114000
         DC   C'<HR></H1><A HREF="../CGIIPLD2">Return to menu'         00115000
         DC   C'</A></BODY></HTML>'                                    00116000
TEXTLEN  DC   AL4(*-NOJAVA)                                            00117000
* ---------------- Dummy Sections --------------------- *              00118000
         CGIDATA DSECT=YES             Define the parameter            00119000
         IPW$EQU                                                       00120000
         END   CGIIPLD2                End of program                  00121000
```

Note:

Keep in mind that while the first fullword of R2 contains the parameter block, there is also a second fullword that contains the address of the TCP/IP FOR VSE internal WAIT routine. If you need to issue a WAIT, load this second fullword to R15, point to your ECB in R1, and BASSM into it. Again, this is only if you need to issue a WAIT.

Be aware that this only applies to the CGI-BAL type. The other assembler format, the TYPE=CGI, differs in that its design requirements are identical to a TCP/IP FOR VSE file I/O driver.

REXX CGIs

REXX is an excellent language for prototyping CGI applications. In this section, we explain how to code a REXX program and show an example.

Programming

Your REXX program receives the following parameters:

- UserID
- Password
- Data
- FOIP
- FOPORT
- LOIP
- LOPORT

If security is off, the user ID and password are both set to ANONYMOUS. After the REXX program receives the data, it processes it and then returns the webpage. To do this, you invoke the HTML( ) function.

To illustrate the power and simplicity of this interface, consider the following sample program. This program allows the user to enter VSE console commands and receive the response from the web browser. As explained above, you receive the user ID, password, data, and other fields. A standard header for your program might appear as follows:

```
/* Get the passed parameters  */
userid=arg(1)
password=arg(2)
data=arg(3)
foip=arg(4)
/*                            */
```

In this example, we are not interested in the LOIP, and so there is no need to reference it. Data passed to the CGI is formatted by the web browser. The web browser passes the parameters in the order they are received. This includes the parameter name, preceded by an ampersand (&) and followed by an equal sign (=), and then the data. This sequence repeats until there is no more data. For example, assume you send three fields named ONE, TWO, and THREE, and the fields contain, respectively, the data strings 111, 222, and 333. The data your program receives is in the following format:

```
&ONE=111&TWO=222&THREE=333
```

Once you have the information and have processed it, you need to send data back to the HTTP daemon.

To do this, you use the REXX HTML function. For example, to send back a simple “Thank you” response, you could use the following code:

```
rc=HTML('<HTML><TITLE>Response</TITLE><BODY>')
rc=HTML('<H2><B><I>Thank you</B></I></H2>')
rc=HTML('</BODY></HTML>')
```

That is all there is to writing a simple REXX CGI. The amount of data can be large and contained in a single HTML( ) call, or it can be small and use several HTML( ) calls. Fewer calls are faster, but not significantly so.

Execution Requirements

As with an assembler program, you must take the following actions before you can run your CGI:

1. Catalog your REXX program in a VSE library that is in the LIBDEF search chain for your TCP/IP FOR VSE CGILOAD partition. This is a program, not a document, so it does not need to be in the sublibrary that contains your HTML documents. If you do put it in the HTML sublibrary, make sure the HTML sublibrary is part of the LIBDEF search chain, as shown in the following example:

```
// LIBDEF *,SEARCH=(PRD1.BASE,USR.HTML)
```

2. Define an HTTPD. If you keep all of your webpages for HTTP daemon HTTP1 in PRD2.HTML and you use a REXX program named VSECOM, you could use the control statements in the following example:

```
DEFINE CGI,PUBLIC='VSECOM',TYPE=CGI-REXX
DEFINE HTTPD,ID=HTTP1,ROOT='PRD2.HTML',CONFINE=NO
DEFINE FILE,PUBLIC='PRD2',DLBL=PRD2,TYPE=LIBRARY
```

Example

The following REXX program functions as a VSE console in a web browser.

```
CATALOG VSECOM.PROC
/*Program: VSECOM
  Purpose: Demonstrate how to code a sample CGI: TCP/IP for VSE.
           This CGI will:
           1) Send a dynamic screen to the user
           2) Read a passed VSE command entered by the user
           3) Return data back to the Web browser
  Description: This program is an example of how to code a REXX-CGI.
*/
userid=arg(1)              /* Get the passed user name  */
password=arg(2)            /* Get the passed password   */
data=arg(3)                /* Get the passed command    */
foip=arg(4)                /* 15-byte IP-address string */
inlen=length(data)         /* Get the length passed     */

if inlen=0 then do         /* A null length returns this screen*/
   x=HTML('text/html;')    /* Required to force MIME type HTML */
   x=HTML('<HTML><HEAD><TITLE>'
   x=HTML('VSE Console Command Processor</TITLE></HEAD>')
   x=HTML('<BODY TEXT="#993300" BGCOLOR="#66FF99"><CENTER>')
   x=HTML('<H2><B><I><FONT COLOR="#000000">')
   x=HTML('VSE Console Command Processor')
   x=HTML('</FONT></I></B></H2></CENTER><P><HR>')
   x=HTML('<FORM METHOD=GET  ACTION="VSECOM">')
   x=HTML('Input:<INPUT TYPE="text" NAME="COMMAND" SIZE=25>')
   x=HTML('<BR><HR></BODY></HTML>')
   exit
   end

parse upper var data request 9 command  /* Data was passed     */
ADDRESS CONSOLE                         /* Activate interface  */
'ACTIVATE NAME CGICONR PROFILE REXNORC'
'CART USCHI'
command                                 /* Pass command to VSE */
rc = GETMSG(msg.,'RESP','USCHI',,5)     /* Get the response    */

x=HTML('text/html;')       /* Required to force MIME type HTML */
x=HTML('<HTML><HEAD><TITLE>')           /* Pass back headings  */
x=HTML('VSE Console Command Processor</TITLE></HEAD>')
x=HTML('<BODY TEXT="#993300" BGCOLOR="#66FF99"><CENTER>')
x=HTML('<H2><B><I><FONT COLOR="#000000">')
x=HTML('VSE Console Command Processor')
x=HTML('</FONT></I></B></H2></CENTER><P><HR>')
x=HTML('<FORM METHOD=GET  ACTION="VSECOM">')
x=HTML('Input:<INPUT TYPE="text" NAME="COMMAND" SIZE=25>')
x=HTML('<BR><HR>')
x=HTML('<FONT COLOR="#000066"><PRE>')

i = 1                                   /* Insert the response */
do while i <= msg.0
   x=HTML(msg.i)
   i=i+1
   end

x=HTML('</BODY></HTML>')                /* And the HTML footer */

ADDRESS CONSOLE 'DEACTIVATE CGICONR'    /* Deactivate interface*/
exit
```

6

SSL/TLS for VSE APIs

Overview

The SSL/TLS for VSE feature provides three APIs for developing cryptographically secure TCP/IP socket applications:

- Secure Socket Layer API (page 125)

  This API allows you to develop SSL/TLS-enabled applications on the VSE platform using functions that can be called within Assembler, C, and other languages that use standard call/save linkage conventions.

  For information on TLS 1.2 protocol support, see “Appendix C: TLS 1.2 Enhancement” on page 208.

- CryptoVSE API (page 148)

  This API allows you to implement cryptographic algorithms in a VSE application.

- Common Encryption Cipher Interface (page 182)

  This API allows you to control the encryption algorithm and key values used in an application without changing the application itself. Functions in this API can be called from within Assembler, COBOL, or other high level languages using standard call/save linkage conventions.

The SSL/TLS for VSE feature is provided with TCP/IP FOR VSE, but it must be activated with a product key. See the “SSL/TLS for VSE” chapter in the *TCP/IP FOR VSE Optional Features Guide* for more information. That chapter also contains an introduction to the SSL/TLS protocol.

Secure Socket Layer (SSL) and Transport Layer Security (TLS) API

You can use the SSL/TLS for VSE API to develop SSL/TLS-enabled applications on the VSE platform. The traditional SSL/TLS handshake between an SSL/TLS-enabled client and an SSL/TLS-enabled server follows this process:

1. The server allocates a socket, binds to a port, performs a listen, and issues an accept.
2. The client connects to the server (for example, the VSE application program) and sends a hello message with the highest release level of the protocol that it supports and an ordered list of preferred cipher suites.
3. The server, running on VSE, passes control to the secure socket initialization routine, which performs the actual SSL/TLS handshake. During the handshake, the server responds to the client’s hello message by choosing the release level of the protocol and the cipher suite that will be used during the session. The server also sends its X.509v3 PKI certificate. Next, a very secure and complex function generates the key material that is used for encryption, decryption, and message authentication.
4. After the handshake successfully completes, the connection is ready to securely exchange data using the secure socket read and write functions of the SSL/TLS for VSE API.

The SSL/TLS for VSE API is closely modeled after the C functions of the *OS/390 SSL Programming Guide and Reference* (IBM manual number SC24-5877). Applications written to conform to these specifications are easily ported into VSE. The functions described on the following pages can be called from the Assembler and the C programming languages.

Hardware Assist Options Settings

You can set several options that affect the exploitation of hardware assists that are used in many of the SSL/TLS and cryptographic functions. These options, described in the table below, are set using keyword values in the $SOCKOPT options phase. See “Appendix A: $SOCKOPT Options Phase,” page 196, for details on setting options in a custom options phase.

Option Setting	**Description**
SSLFLG2=$OPTSNHC	Do not use hardware cryptography for RSA operations. Setting this option suppresses the use of hardware cryptography for RSA cipher algorithm requests. See also Notes below.

Option Setting	Description
SSLFLG2=$OPTSNZA	Never issue CP Assist Cryptographic Function (CPACF) z/architecture hardware instructions. See also Notes below.
SSLFLG2=$OPTSFZA	Always issue CPACF z/architecture hardware instructions. See also Notes below. Use caution when choosing this option. If the hardware instructions are not available, an operation exception program-check abend will occur in the application.

Notes:

1. When the TLS 1.2 version of the protocol is negotiated, the hardware assists for both RSA and CPACF must be available, and the $OPTSNHC, $OPTSNZA, and $OPTSFZA settings are ignored.

2. Normally, z/ architecture CPACF hardware instructions are detected automatically and used when available. The $OPTSNZA and $OPTSFZA options allow you to suppress or force, respectively, the use of CPACF hardware cryptographic instructions.

Error Codes

The SSLVSE.A member contains terse explanations of most of the codes returned by SSL functions when an error occurs. This member is in the TCP/IP FOR VSE library. For further analysis, see the section “Debugging Problems” on page 194.

Functions

The SSL/TLS for VSE API’s functions are described on the following pages of this section.

gsk_free_memory( )

This function is maintained for portability of OS/390 applications. It is not used by the SSL/TLS for VSE API. The syntax is as follows.

```
#include <sslvse.h>
void gsk_free_memory(void * address,
                     void * reserved);
```

The parameters are described in the following table:

Parameter	Description
address	Pointer to the memory that is to be freed. The address was passed to the application by a previous call to an SSL function.
reserved	Reserved for future use. Code a null.

The return codes are as follows:

Return Code	Description
Zero	Successful completion.
Non-zero	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- This function is not currently used.
- For Assembler, use macro SSLVSE to generate the required data areas and call the IPCRFMEM entry point contained in the IPCRYPTS object deck. See the SSLSERVR and SSLCLINT sample programs for detailed Assembler interface specifications.
- For information on TLS 1.2 protocol support, see "Appendix C: TLS 1.2 Enhancement" on page 208.

gsk_get_cipher_info( )

This function requests cipher-related information for SSL/TLS for VSE. The information determines the encryption level that the system can support and returns a list of cipher specifications that SSL can use. This allows an application to determine, at run time, the level of SSL encryption that the installed application can request.

The syntax is as follows.

```
#include <sslvse.h>
int gsk_get_cipher_info(int level,
        gsk_sec_level * sec_level, void * reserved);
```

The parameters are described in the following table:

Parameter	**Description**
level	Determines the type of cipher information to be returned. The types are as follows: • `GSK_LOW_SECURITY`. Causes only exportable cipher information to be returned. This value is useful when setting up SSL communications with systems that may be located outside of the U.S. and Canada and in any area where strong cryptographic functions are not available. • `GSK_HIGH_SECURITY`. Causes exportable and domestic cipher information to be returned. See the `v3cipher_specs` and `cipher_specs` fields in the gsk_soc_init_data structure, in the function gsk_secure_soc_init( ), for a list of valid cipher values.
sec_level	Pointer to the gsk_sec_level data area.
reserved	Reserved for future use. Code a null.

The gsk_sec_level structure specifies information about the level of cryptography that is available on the system. The application must allocate the memory necessary for this structure. On successful return, the contents of the structure are set.

The gsk_sec_level data area has the following structure:

```
typedef struct gsk_sec_level {
    int  version;            /* Output: System SSL version            */
    char v3cipher_specs[64]; /* Output: The sslv3 cipher specs allowed */
    char v2cipher_specs[32]; /* Output: The sslv2 cipher specs allowed */
    int  security_level;     /* Output: Initially one of   */
                             /*   GSK_SEC_LEVEL_US,        */
                             /*   GSK_SEC_LEVEL_EXPORT,    */
                             /*   GSK_SEC_LEVEL_EXPORT_FR  */
 } gsk_sec_level;
```

The fields in the gsk_sec_level structure are described in the table below.

Field	**Description**
version	Specifies the version of System SSL that is being used. This returns the value `GSK_VERSION3` as defined in <sslvse.h>.
v3cipher_specs[64]	Specifies the SSL/TLS cipher specs that are acceptable on the system. This data is contained in the v3cipher_specs field in the gsk_soc_init_data structure that is passed on the gsk_secure_soc_init( ) call.
v2cipher_specs[32]	The SSL/TLS for VSE implementation does not support SSLv2, and the v2cipher_specs contain null information.

security_level	Specifies the level of encryption that is acceptable on the system. One of the following values, as defined in <sslvse.h>, is returned in this field:
	`GSK_SEC_LEVEL_US`
	`GSK_SEC_LEVEL_EXPORT`

The return codes are as follows:

Return Code	**Description**
Zero	Successful completion.
Non-zero	An error occurred. See the section “Debugging Problems” on page 194.

Usage Notes:

- You can use gsk_get_cipher_info( ) to determine the valid values that are specified in the cipher_specs of the gsk_soc_init_data area used by gsk_secure_soc_init( ).
- For Assembler, use macro SSLVSE to generate the required data areas and call the IPCRGCIN entry point contained in the IPCRYPTS object deck. See the SSLSERVR and SSLCLINT sample programs for detailed Assembler interface specifications.
- For information on TLS 1.2 protocol support, see “Appendix C: TLS 1.2 Enhancement” on page 208.

gsk_get_dn_by_label( )

This optional function allows you to identify the member name containing the private key and certificates. The syntax is as follows:

```
#include <sslvse.h>
char * gsk_get_dn_by_label(char * label);
```

The parameter is described in the following table:

Parameter	Description
label	Points to a null-terminated character string of the library member contained in the key ring lib.sublib. The member name must be eight characters or less in length and be terminated with a null character (`x'00'`).

The return codes are as follows:

Return Code	Description
Positive value	Successful completion. The value is a pointer to a character string that identifies the library member name.
Zero or a negative value	An error occurred. See the section “Debugging Problems” on page 194.

Usage Notes:

- For Assembler, use the macro SSLVSE to generate the required data areas and call the IPCRGDBL entry point contained in the IPCRYPTS object deck. See the SSLSERVR and SSLCLINT sample programs for detailed Assembler interface specifications.
- For information on TLS 1.2 protocol support, see “Appendix C: TLS 1.2 Enhancement” on page 208.

gsk_initialize( )

This function sets the overall SSL/TLS for VSE environment for the current partition. After the function completes successfully, the application is ready to call SSL/TLS for VSE interfaces and create and use secure socket connections.

The syntax is as follows:

```
#include <sslvse.h>
int gsk_initialize(gsk_init_data * init_data);
```

The parameter is described in the following table:

Parameter	Description
init_data	Pointer to the gsk_init_data area

The gsk_init_data area has the following structure:

```
typedef struct gsk_init_data {
   char * sec_types;             /* Minimum security protocol:  */
   char * keyring;               /* Key ring file name          */
   char * keyring_pw;            /* Key ring password           */
   char * keyring_stash;         /* File name - stashed passwrd */
   long   V2_session_timeout;    /* Number of seconds for SSLV2 */
   long   V3_session_timeout;    /* Number of seconds for SSLV3 */
   char * LDAP_server;           /* Name or IP addr of X500 host*/
   int    LDAP_port;             /* Port number of X500 host    */
   char * LDAP_user;             /* User name for X500 host     */
   char * LDAP_password;         /* Password of X500 host       */
   gsk_ca_roots   LDAP_CA_roots; /* Which CA roots to use       */
   gsk_auth_type  auth_type      /* Client authentication type  */
   } gsk_init_data;
```

The fields in the gsk_init_data structure are described in the table below.

Field	Description
sec_types	Specifies a null-terminated character string that identifies the minimum acceptable security protocol version to be used. The value must be in upper case characters. Valid values are: • `SSL30` for version 3.0 of the protocol. This conforms to the SSL 3.0 standard as defined by Netscape, which has now been turned over to the IETF and renamed TLS. • `TLS10` for version 3.1 of the protocol. This conforms to TLS 1.0 protocol as defined in RFC2246. • `TLS11` for version 3.2 of the protocol. This conforms to TLS 1.1 protocol as defined in RFC4346. • `TLS12` for version 3.3 of the protocol. This conforms to TLS 1.2 protocol as defined in RFC5246. **Note:** *Clients* should set this field to the highest protocol version they can support. *Servers* should set this field to the lowest protocol version they are willing to accept.
keyring	Optionally specifies a null-terminated character string that identifies the lib.sublib in which the private key and certificates are stored. The lib name must be 7 characters or less, followed by a period, followed by the sublib name, and terminated with a null character (`x'00'`). A null pointer causes the private key and certificates to be read from the default sequential disk files.
keyring_pw	Not used.
keyring_stash	Not used.
V2_session_timeout	Not used.
V3_session_timeout	Specifies the number of seconds in which a server allows a client to reconnect without performing a full SSL handshake. The recommended setting for this field is `86400`, which is the number of seconds in 24 hours. A setting of `0` effectively disables the fast reconnect option and causes higher CPU consumption during the handshake process because it requires all clients to perform a full SSL handshake.
LDAP_server	Not used.
LDAP_port	Not used.

Field	**Description**
LDAP_user	Not used.
LDAP_password	Not used.
gsk_ca_roots	Specifies which CA roots to use for client certificate authentication. Valid values are `0` and `1`. When an application wants to allow client authentication with certificates issued by the same certificate authority as VSE, it should use a value of `1`.
gsk_auth_type	Specifies the method to use for verifying the client’s certificate. This field is used only when field gsk_ca_roots is set to `1`. Valid values are `0` for Client_auth_local `1` for Client_auth_strong_over_ssl `2` for Client_auth_strong `3` for Client_auth_passthru

The return codes are as follows:

Return Code	**Description**
Zero	Successful completion.
Non-zero	An error occurred. See the section “Debugging Problems” on page 194.

Usage Notes:

- You can make multiple calls to gsk_initialize( ) as long as you call gsk_uninitialize( ) to clean up the existing SSL/TLS for VSE environment before the next call.
- For Assembler, use macro SSLVSE to generate the required data areas and call the IPCRINIT entry point contained in the IPCRYPTS object deck. See the SSLSERVR and SSLCLINT sample programs for detailed Assembler interface specifications.
- For information on TLS 1.2 protocol support, see “Appendix C: TLS 1.2 Enhancement” on page 208.

gsk_secure_soc_close( )

This function ends a secure socket connection and frees all SSL/TLS for VSE resources for that connection. The syntax is as follows:

```
#include <sslvse.h>
void gsk_secure_soc_close(gsk_soc_data * user_socket);
```

The parameter is described in the following table:

Parameter	Description
user_socket	Points to the gsk_soc_data structure returned from the gsk_secure_soc_init( ) call.

The return codes are as follows:

Return Code	Description
Zero	Successful completion.
Non-zero	An error occurred. See the section "Debugging Problems" on page 194.

Security note:

This note describes a measure to prevent hacker truncation attacks. TCP/IP FOR VSE always sends a close_notify alert as required by the protocol specification, but other non-VSE applications may not comply with the specification. This can cause an application on VSE to hang during session termination while waiting for a close_notify alert. To avoid this problem, TCP/IP FOR VSE does not require a close_notify alert to be received to complete a close.

You can change this default behavior by setting the keyword SSLFLG1 to $OPTSRQC in the $SOCKOPT options phase. When this option is set, TCP/IP FOR VSE requires a close_notify alert to be received during a gsk_secure_soc_close. This means that a secure socket close will not complete successfully unless a close_notify alert is received from the non-VSE application.

CSI International recommends that all customers use the $OPTSRQC option to prevent truncation attacks. Each site must weigh the risk of problems caused by applications that do not comply with the protocol's specification of exchanging close_notify alerts before terminating a connection. If you do not set this option, then your applications may be susceptible to truncation attacks.

See "Appendix A: $SOCKOPT Options Phase," page 196, for details on setting options in a custom options phase.

Usage Notes:

- This function frees all storage referenced by the user_socket parameter.

- The user application must close all socket descriptors opened by any socket API. This function does not close any open socket descriptors.

- For Assembler, use macro SSLVSE to generate the required data areas and call the IPCRSCLS entry point contained in the IPCRYPTS object deck. See the SSLSERVR and SSLCLINT sample programs for detailed specifications.

- For information on TLS 1.2 protocol support, see “Appendix C: TLS 1.2 Enhancement” on page 208.

gsk_secure_soc_init( )

This function initializes the data areas necessary for SSL/TLS for VSE to initiate or accept a secure socket connection. After the function completes successfully, a handle is returned to the application. Other calls using this secure socket connection must use this handle.

The syntax is as follows:

```
#include <sslvse.h>
gsk_soc_data * gsk_secure_soc_init(
                    gsk_soc_init_data * soc_init_data);
```

The parameter is described in the following table:

Parameter	Description
soc_init_data	Pointer to the gsk_soc_init_data area

During the call, a complete SSL handshake is performed based on the input specified in the gsk_soc_init_data structure. While SSL/TLS for VSE performs the mechanics of the SSL handshake, the application must supply the routines necessary to transport the SSL data during the SSL handshake, as well as for all subsequent read/write operations. See the Usage notes below for information on disallowing fast resumes.

The gsk_soc_init_data structure specifies characteristics of the secure socket connection. In addition, SSL/TLS for VSE uses this structure to return information about the secure socket connection after it is established. The gsk_soc_init_data area has the following structure:

```
typedef struct gsk_soc_init_data {
   int fd;                      /* Socket descriptor             */
   gsk_handshake hs_type;       /* Client or server handshake    */
   char * DName;                /* Key ring entry name           */
   char * sec_type;             /* Type of security protocol used
                                   to protect this socket        */
   char * cipher_specs;         /* Cipher specs choice and order
                                   for SSLV2                     */
   char * v3cipher_specs;       /* Cipher specs choice and order
                                   for SSLV3                     */
   int (* skread)               /* User-defined READ func pointer */
       (int fd, void * buffer, int numbytes);
   int (* skwrite)              /* User-defined WRITE func pointer*/
       (int fd, void * buffer, int num_bytes);
   unsigned char cipherSelected[3];       /* V2 CipherSpec used   */
   unsigned char v3cipherSelected[2];     /* V3 CipherSpec used   */
   int failureReasonCode;                 /* Failure reason code  */
   gsk_cert_info * cert_info; /* Used during client authentication*/
   gsk_init_data * gsk_data;    /* Required for password exchange
                                   during DName negotiation       */
 } gsk_soc_init_data;
```

The fields in the gsk_soc_init_data structure are described in the table below.

Field	Description
fd	Socket descriptor for this connection. The socket descriptor address is passed to the application routines specified in fields skread and skwrite. These application-supplied routines use the socket descriptor as required to read/write SSL data. **Note 1:** No socket calls are issued by SSL/TLS for VSE using the socket descriptor. It is the address of a token that is passed to the user-provided socket read and write exit routines. You may store the address of a dynamic work area as the socket descriptor, and when your exit gets control you have easy addressability to your work area containing the your actual socket descriptor. **Note 2:** The socket must be created, opened, and connected before calling gsk_secure_soc_init( ).
hs_type	Specifies how the SSL handshake is performed. Valid values are • `GSK_AS_SERVER` to perform the SSL handshake as a server without client authentication. • `GSK_AS_SERVER_WITH_CLIENT_AUTH` to perform the SSL handshake as a server that requires client authentication. • `GSK_AS_CLIENT` to perform the SSL handshake as a client with or without client authentication. • `GSK_AS_CLIENT_NO_AUTH` to perform the SSL handshake as a client without client authentication.
DName	Points to a null-terminated character string of the library member name for the .cert, .root, and .prvk files to be read from the keyring lib.sublib. The member name must be eight characters or less in length and must terminate with a null character (`x'00'`). To use sequential disk files, specify SDFILES.

Field	Description
sec_type	After a socket is successfully connected, this contains a pointer to a null-terminated character string that identifies the security protocol version set by the server application during the handshake process: • `SSL30` for version 3.0 of the protocol. This conforms to the SSL 3.0 standard as defined by Netscape. • `TLS31` for version 3.1 of the protocol. This conforms to TLS 1.0 protocol as defined in RFC2246. • `TLS11` for version 3.2 of the protocol. This conforms to TLS 1.1 protocol as defined in RFC4346. • `TLS12` for version 3.3 of the protocol. This conforms to TLS 1.2 protocol as defined in RFC5246.
cipher_specs	Specifies a null-terminated character string that contains the list of SSL Version 2.0 ciphers in order of usage preference. This field is NOT USED by SSL/TLS for VSE.
v3cipher_specs	Specifies a null-terminated character string that contains the list of SSL/TLS ciphers in order of usage preference. Valid values are: `01` for RSA-NULL-MD5 `02` for RSA-NULL-SHA `08` for RSA-SDES040-SHA `09` for RSA-SDES056-SHA `0A` for RSA-TDES168-SHA `2F` for RSA-AES128-SHA `35` for RSA-AES256-SHA You can use any combination of these values in any order. The level of cryptography installed on the system may render some values invalid. See gsk_get_cipher_info( ) for information about determining the cipher specs supported by the system. If you specify a NULL value for cipher_specs, the default SSL/TLS cipher specs are used. The default ciphers are dependent on the SSLCIPH keyword setting in $SOCKOPT. See "Appendix A: $SOCKOPT Options Phase," page 196, for details on setting options in a custom options phase.

Field	**Description**

skread	Points to an application-provided routine that performs a read function for SSL/TLS for VSE. The parameters for this routine must be defined as specified in skread. This application-provided routine must use standard linkage conventions. SSL/TLS for VSE uses the skread routine while performing the SSL handshake during the gsk_secure_soc_init( ) call and the gsk_secure_soc_read( ) call. The skread routine contains the following code, which uses the sockets recv( ) call to read the data for the SSL connection:
	`int skread( int fd, void &data, int len ) {`
	`return( recv( fd, data, len, 0 ) ); }`
	The skread routine must return one of the following:
	• A positive value to indicate the number of bytes received
	• A negative value to indicate that an error occurred
	If the return code is set to zero, the gsk_secure_soc_init( ) function will reissue the call to skread until either a positive or a negative return code is received.

skwrite	Points to an application-provided I/O routine that performs a write function for SSL/TLS for VSE. The parameters for this routine must be defined as specified in skwrite. This application-provided I/O routine must use standard linkage conventions. SSL/TLS for VSE uses the skwrite routine while performing the SSL handshake during the gsk_secure_soc_init( ) call and the gsk_secure_soc_write( ) call. The skwrite routine contains the following code, which uses the sockets send( ) call to write the data for the SSL connection:
	`int skwrite( int fd, void &data, int len) {`
	`return( send( fd, data, len, 0 ) );       }`
	The skwrite routine must return one of the following:
	• A positive value to indicate the number of bytes sent
	• A negative value to indicate that an error occurred
	If the return code is set to zero, the SSL connection will be terminated.

Field	**Description**
cipherSelected	Specifies the architected SSL version 2.0 cipher spec value selected for this session. SSL/TLS for VSE does not support the SSL 2.0 standard.
V3cipherSelected	Specifies the architected SSL version 3.0 cipher spec value selected for this session, for example, `0X00,0X09`.
failureReasonCode	Specifies the failure reason code for gsk_secure_soc_init( ).
cert_info	Specifies the Distinguished Name components from the client's certificate. This parameter is valid only when client authentication is requested for a server using SSL. At successful completion, this field contains a pointer to the following structure, which is also contained in <sslvse.h>:

```
typedef struct gskcertinfo {
  char * cert_body;      /* Base64 certificate body        */
  int    cert_body_len;  /* Length of base64 cert body     */
  char * sessionID;      /* Session ID for this connection */
  int    newSessionID;   /* Flag to indicate if new session*/
  char * serial_num;     /* Certificate Serial number      */
  char * common_name;    /* Common Name of client          */
  char * locality;           /* Locality                   */
  char * state_or_province;  /* State or Province          */
  char * country;            /* Country                    */
  char * org;                /* Organization               */
  char * org_unit;           /* Organizational unit        */
  char * issuer_common_name; /* Issuer's common name       */
  char * issuer_locality;    /* Issuer's locality          */
  char * issuer_state_or_province; /* Issuer's sta or prov */
  char * issuer_country;    /* Issuer's country            */
  char * issuer_org;        /* Issuer's organization       */
  char * issuer_org_unit;   /* Issuer's organizational unit*/
} gsk_cert_info;
```

`gsk_data`	Points to the gsk_init_data structure. You must specify the address of the same gsk_init_data structure that was used during the gsk_initialize( ) function call.

The return codes are as follows:

Return Code	Description
Positive value	Successful completion. The value is a pointer to a structure of type gsk_soc_data. Save this pointer: the structure is used in subsequent SSL/TLS for VSE operations. The gsk_soc_data structure is defined in <sslvse.h> as follows: `typedef struct _gsk_soc_data { void * sk_SSLHandle; } gsk_soc_data;`
Zero or a negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- You can block fast resumes by setting the keyword SSLFLG1 to $OPTSNFR (Fast Resume Not Allowed) in the $SOCKOPT options phase. Setting this option causes an SSL/TLS server application to reject all fast resume requests. Fast resume can improve performance, but some sites may consider it to be a security exposure. When fast resume is disabled, a full SSL/TLS handshake is performed for all session negotiations.

  See "Appendix A: $SOCKOPT Options Phase," page 196, for details on setting options in a custom options phase.

- The socket descriptor must be open and connected before you call this routine. For socket operations, this implies that a client must perform the socket( ) and connect( ) calls before the gsk_secure_soc_init( ) call. For servers, this implies that the server must perform the socket( ), bind( ), listen( ), and accept( ) calls before the gsk_secure_soc_init( ) call.

- For Assembler, use macro SSLVSE to generate the required data areas and call the IPCRSINI entry point contained in the IPCRYPTS object deck. See the SSLSERVR and SSLCLINT sample programs for detailed Assembler interface specifications.

- For information on TLS 1.2 protocol support, see "Appendix C: TLS 1.2 Enhancement" on page 208.

gsk_secure_soc_read( )

This function receives data on a secure socket connection using the application-specified read routine.

The syntax is as follows:

```
#include <sslvse.h>
int gsk_secure_soc_read(gsk_soc_data * user_socket,
                                void * data_buffer,
                                 int   buffer_length);
```

The parameters are described in the following table:

Parameter	Description
user_socket	Pointer to gsk_soc_data
data_buffer	Pointer to user-supplied buffer in which the data is to be stored
buffer_length	Specifies the length of *data_buffer*

The return codes are as follows:

Return Code	Description
Zero or a positive value	Successful completion. The value is the number of bytes read.
Negative value	An error occurred. See the section “Debugging Problems” on page 194.

Usage Notes:

- The maximum length of the data returned cannot exceed 32K. This is because SSL is a record-level protocol in which the largest record allowed is 32K minus the necessary SSL record headers.

- Mixing calls to gsk_secure_soc_read( ) and any of the socket read function’s receive calls, while possible, is not recommended. This requires very close matching of operations between client and server programs. If any portion of an SSL record is read using a socket read function, a fatal SSL protocol error is detected when the next gsk_secure_soc_read( ) is performed.

- SSL/TLS for VSE can be mixed with socket reads and writes, but they must be performed in matched sets. If a client application writes 100 bytes of data using one or more of the socket send calls, then the server application must read exactly 100 bytes of data using one or more of the socket receive calls. This is also true for gsk_secure_soc_read( ) and gsk_secure_soc_write( ).

- Because SSL is a record-oriented protocol, it must receive an entire record before it is decrypted and any data is returned to the application. Thus, a select( ) can indicate that data is available to be read, but a subsequent gsk_secure_soc_read( ) can hang while waiting for the remainder of the SSL record to be received.

- For Assembler, use macro SSLVSE to generate the required data areas and call the IPCRSRED entry point contained in the IPCRYPTS object deck. See the SSLSERVR and SSLCLINT sample programs for detailed Assembler interface specifications.

- For information on TLS 1.2 protocol support, see "Appendix C: TLS 1.2 Enhancement" on page 208.

gsk_secure_soc_reset( )

This function refreshes the security parameters, such as encryption keys, for a session. You can also use it to resume or restart a cached session.

The syntax is as follows:

```
#include <sslvse.h>
int gsk_secure_soc_reset(gsk_soc_data * user_socket);
```

The parameter is described in the following table:

Parameter	**Description**
user_socket	Points to the gsk_soc_data structure returned from the gsk_secure_soc_init( ) call.

The return codes are as follows:

Return Code	**Description**
Zero	Successful completion.
Non-zero	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- Use gsk_secure_soc_reset( ) when a client or server needs to reset the SSL environment. Call gsk_secure_soc_reset( ) only after a successful call to gsk_secure_soc_init( ). Also, use gsk_secure_soc_reset( ) when resuming or restarting a connection for an SSL session that was cached and when resetting the keys used for that connection.
- For Assembler, use macro SSLVSE to generate the required data areas and call the IPCRSRST entry point contained in the IPCRYPTS object deck. See the SSLSERVR and SSLCLINT sample programs for detailed Assembler interface specifications.
- For information on TLS 1.2 protocol support, see "Appendix C: TLS 1.2 Enhancement" on page 208.

gsk_secure_soc_write( )

This function sends data on a secure socket connection using the application-specified write routine. The syntax is as follows:

```
#include <sslvse.h>
int gsk_secure_soc_write(gsk_soc_data * user_socket,
            void * data_buffer,  int  buffer_length);
```

The parameters are described in the following table:

Parameter	**Description**
user_socket	Pointer to gsk_soc_data.
data_buffer	Pointer to the user-supplied buffer in which the data to be written is stored.
buffer_length	Specifies the length of *data_buffer*. The maximum allowed is 65536 bytes.

The return codes are as follows:

Return Code	**Description**
Zero or a positive value	Successful completion. The value is the number of bytes written.
Negative value	Error occurred. See "Debugging Problems" page 194.

Usage Notes:

- If the application data sent to an SSL/TLS for VSE application is more than 32K, you must make multiple calls to gsk_secure_soc_read( ) in order to read the entire block of application data.

- SSL/TLS for VSE reads and writes can be mixed with socket reads and writes, but they must be performed in matched sets. If a client application writes 100 bytes of data using one or more of the socket send calls, then the server application must read exactly 100 bytes of data using one or more of the socket receive calls. This is also true for gsk_secure_soc_read( ) and gsk_secure_soc_write( ). If a write buffer is separated into multiple buffers, the remote site of the secure socket connection must perform enough gsk_secure_soc_read( ) operations to read the complete buffer.

- For Assembler, use macro SSLVSE to generate the required data areas and call the IPCRSWRT entry point contained in the IPCRYPTS object deck. See the SSLSERVR and SSLCLINT sample programs for detailed Assembler interface specifications.

- For information on TLS 1.2 protocol support, see "Appendix C: TLS 1.2 Enhancement" on page 208.

gsk_uninitialize( )

This function removes the current overall settings for the SSL environment. It removes fields such as session timeout values and SSL protocols.

The syntax is as follows:

```
#include <sslvse.h>
int gsk_uninitialize(void);
```

The return codes are as follows:

Return Code	Description
Zero	Successful completion.
Non-zero	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- Use gsk_uninitialize( ) when you need to reset the System SSL environment settings. Then, use gsk_initialize( ) to create a new set of System SSL environment settings.
- You must close all SSL sessions that were created using the current System SSL environment before you call gsk_uninitialize( ).
- For Assembler, use macro SSLVSE to generate the required data areas and call the IPCRUNIN entry point contained in the IPCRYPTS object deck. See the SSLSERVR and SSLCLINT sample programs for detailed Assembler interface specifications.
- For information on TLS 1.2 protocol support, see "Appendix C: TLS 1.2 Enhancement" on page 208.

gsk_user_set( )

This function is not used by the SSL/TLS for VSE API, but it is maintained for portability of OS/390 applications.

The syntax is as follows:

```
#include <sslvse.h>
int gsk_user_set(int    user_data_fid,
                 void * user_input_data,
                 void * reserved);
```

The parameters are described in the following table:

Parameter	**Description**
user_data_fid	Specifies the action to perform.
user_input_data	Specifies information required for the requested action.
reserved	A reserved field; specify it as null.

The return codes are as follows:

Return Code	**Description**
Zero	Successful completion.
Non-zero	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- For Assembler, use macro SSLVSE to generate the required data areas and call the IPCRUSET entry point contained in the IPCRYPTS object deck. See the SSLSERVR and SSLCLINT sample programs for detailed Assembler interface specifications.
- For information on TLS 1.2 protocol support, see "Appendix C: TLS 1.2 Enhancement" on page 208.

CryptoVSE API

Overview

SSL/TLS for VSE provides a cryptographic API that you can use to implement cryptographic algorithms in a VSE application. The algorithms provide confidentiality, authentication, and integrity of data in the applications.

There are many good websites and books that discuss cryptography, and it is beyond the scope of this document to provide a comprehensive treatment of the subject. To learn more about the algorithms implemented in this API and how they are used, see “Published Standards” and “References” in the “SSL/TLS for VSE” chapter in the *TCP/IP FOR VSE Optional Features Guide*.

This API provides the following cryptographic services:

- Data encryption:

  AES 128, 192, and 256 based on FIPS Pub 197

  DES based on FIPS Pub 46-3

  Triple DES based on ANSI X9.52 Triple DES

  RSA PKCS #1 also contained in RFC2313

- Message Digests:

  MD5 based on RFC1321

  SHA-1 based on FIPS Pub 180-1

- Message Authentication:

  HMAC based on RFC2104

- Digital Signatures:

  RSA PKCS #1 with either a SHA1 or MD5 message digest

- Miscellaneous:

  Universal printable-character encoding

  Random-number generation (RNG)

  SSL/TLS pseudo RNG as documented in RFC2246

The functions that make up the cryptographic API are described in the following pages of this section.

Note: You must issue the cry_initialize( ) call before issuing any other crypto calls. This call initializes the environment for cryptography functions.

cry_3des_cbc_encrypt( )

This function uses the Triple DES algorithm in CBC (Cipher Block Chaining) mode to encrypt the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_3des_cbc_encrypt( &input, input_length, &key,
                          key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	Description
input	Reference to the data to be encrypted.
input_length	Length of the data to be encrypted.
key	Reference to the key data for the encryption.
key_length	Length of the key data for the encryption.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	Description
Zero	Successful completion.
Negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- The data length must be a multiple of 8 bytes.
- The key length must be 32 bytes. The first 8 bytes contain the initialization vector, which is followed by the 24-byte key.
- For Assembler, call the CRYTDESE vcon with the same parameters.

cry_3des_cbc_decrypt( )

This function uses the Triple DES algorithm in CBC (Cipher Block Chaining) mode to decrypt the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_3des_cbc_decrypt(&input, input_length, &key,
                  key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	Description
input	Reference to the data to be decrypted.
input_length	Length of the data to be decrypted.
key	Reference to the key data for the decryption.
key_length	Length of the key data for the decryption.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	Description
Zero	Successful completion.
Negative value	An error occurred. See the section “Debugging Problems” on page 194.

Usage Notes:

- The data length must be a multiple of 8 bytes.
- The key length must be 32 bytes. The first 8 bytes contain the initialization vector, followed by the 24-byte key.
- For Assembler, call the CRYTDESD vcon with the same parameters.

cry_aes128_cbc_encrypt( )

This function uses the AES 128 algorithm in CBC (Cipher Block Chaining) mode to encrypt the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_aes128_cbc_encrypt(&input, input_length, &key,
                    key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	Description
input	Reference to the data to be encrypted.
input_length	Length of the data to be encrypted.
key	Reference to the key data for the encryption.
key_length	Length of the key data for the encryption.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	Description
Zero	Successful completion.
Negative value	An error occurred. See the section “Debugging Problems” on page 194.

Usage Notes:

- The data length must be a multiple of 16 bytes.
- The key length must be 32 bytes. The first 16 bytes contain the initialization vector, followed by the 16-byte key.
- For Assembler, call the CRYA12EC vcon with the same parameters.

cry_aes128_cbc_decrypt( )

This function uses the AES 128 algorithm in CBC (Cipher Block Chaining) mode to decrypt the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_aes128_cbc_decrypt(&input, input_length, &key,
                           key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	**Description**
input	Reference to the data to be decrypted.
input_length	Length of the data to be decrypted.
key	Reference to the key data for the decryption.
key_length	Length of the key data for the decryption.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	**Description**
Zero	Successful completion.
Negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- The data length must be a multiple of 16 bytes.
- The key length must be 32 bytes. The first 16 bytes contain the initialization vector, followed by the 16-byte key.
- For Assembler, call the CRYA12DC vcon with the same parameters.

cry_aes128_ecb_encrypt( )

This function uses the AES 128 algorithm in ECB (Electronic Feedback) mode to encrypt the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_aes128_ecb_encrypt(&input, input_length, &key,
                    key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	**Description**
input	Reference to the data to be encrypted.
input_length	Length of the data to be encrypted.
key	Reference to the key data for the encryption.
key_length	Length of the key data for the encryption.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	**Description**
Zero	Successful completion.
Negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- The data length must be a multiple of 16 bytes.
- The key length must be 16 bytes.
- For Assembler, call the CRYA12EE vcon with the same parameters.

cry_aes128_ecb_decrypt( )

This function uses the AES 128 algorithm in ECB (Electronic Feedback) mode to decrypt the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_eas128_ecb_decrypt(&input, input_length, &key,
                   key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	Description
input	Reference to the data to be decrypted.
input_length	Length of the data to be decrypted.
key	Reference to the key data for the decryption.
key_length	Length of the key data for the decryption.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	Description
Zero	Successful completion.
Negative value	An error occurred. See the section “Debugging Problems” on page 194.

Usage Notes:

- The data length must be a multiple of 16 bytes.
- The key length must be 16 bytes.
- For Assembler, call the CRYA12DE vcon with the same parameters.

cry_aes192_cbc_encrypt( )

This function uses the AES 192 algorithm in CBC (Cipher Block Chaining) mode to encrypt the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_aes192_cbc_encrypt(&input, input_length, &key,
                           key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	Description
input	Reference to the data to be encrypted.
input_length	Length of the data to be encrypted.
key	Reference to the key data for the encryption.
key_length	Length of the key data for the encryption.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	Description
Zero	Successful completion.
Negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- The data length must be a multiple of 16 bytes.
- The key length must be 40 bytes. The first 16 bytes contain the initialization vector, followed by the 24-byte key.
- For Assembler, call the CRYA19EC vcon with the same parameters.

cry_aes192_cbc_decrypt( )

This function uses the AES 192 algorithm in CBC (Cipher Block Chaining) mode to decrypt the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_aes192_cbc_decrypt(&input, input_length, &key,
                   key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	Description
input	Reference to the data to be decrypted.
input_length	Length of the data to be decrypted.
key	Reference to the key data for the decryption.
key_length	Length of the key data for the decryption.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	Description
Zero	Successful completion.
Negative value	An error occurred. See the section “Debugging Problems” on page 194.

Usage Notes:

- The data length must be a multiple of 16 bytes.
- The key length must be 40 bytes. The first 16 bytes contain the initialization vector, followed by the 24-byte key.
- For Assembler, call the CRYA19DC vcon with the same parameters.

cry_aes192_ecb_encrypt( )

This function uses the AES 192 algorithm in ECB (Electronic Feedback) mode to encrypt the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_aes192_ecb_encrypt(&input, input_length, &key,
                   key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	Description
input	Reference to the data to be encrypted.
input_length	Length of the data to be encrypted.
key	Reference to the key data for the encryption.
key_length	Length of the key data for the encryption.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	Description
Zero	Successful completion.
Negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- The data length must be a multiple of 16 bytes.
- The key length must be 24 bytes.
- For Assembler, call the CRYA19EE vcon with the same parameters.

cry_aes192_ecb_decrypt( )

This function uses the AES 192 algorithm in ECB (Electronic Feedback) mode to decrypt the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_aes192_ecb_decrypt(&input, input_length, &key,
                           key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	Description
input	Reference to the data to be decrypted.
input_length	Length of the data to be decrypted.
key	Reference to the key data for the decryption.
key_length	Length of the key data for the decryption.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	Description
Zero	Successful completion.
Negative value	An error occurred. See the section “Debugging Problems” on page 194.

Usage Notes:

- The data length must be a multiple of 16 bytes.
- The key length must be 24 bytes.
- For Assembler, call the CRYA19DE vcon with the same parameters.

cry_aes256_cbc_encrypt( )

This function uses the AES 256 algorithm in CBC (Cipher Block Chaining) mode to encrypt the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_aes256_cbc_encrypt(&input, input_length, &key,
                           key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	Description
input	Reference to the data to be encrypted.
input_length	Length of the data to be encrypted.
key	Reference to the key data for the encryption.
key_length	Length of the key data for the encryption.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	Description
Zero	Successful completion.
Negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- The data length must be a multiple of 16 bytes.
- The key length must be 48 bytes. The first 16 bytes contain the initialization vector, followed by the 32-byte key.
- For Assembler, call the CRYA25EC vcon with the same parameters.

cry_aes256_cbc_decrypt( )

This function uses the AES 256 algorithm in CBC (Cipher Block Chaining) mode to decrypt the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_aes256_cbc_decrypt(&input, input_length, &key,
                           key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	**Description**
input	Reference to the data to be decrypted.
input_length	Length of the data to be decrypted.
key	Reference to the key data for the decryption.
key_length	Length of the key data for the decryption.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	**Description**
Zero	Successful completion.
Negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- The data length must be a multiple of 16 bytes.
- The key length must be 48 bytes. The first 16 bytes contain the initialization vector, followed by the 32-byte key.
- For Assembler, call the CRYA25DC vcon with the same parameters.

cry_aes256_ecb_encrypt( )

This function uses the AES 256 algorithm in ECB (Electronic Feedback) mode to encrypt the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_aes256_ecb_encrypt(&input, input_length, &key,
                   key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	**Description**
input	Reference to the data to be encrypted.
input_length	Length of the data to be encrypted.
key	Reference to the key data for the encryption.
key_length	Length of the key data for the encryption.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	**Description**
Zero	Successful completion.
Negative value	An error occurred. See the section “Debugging Problems” on page 194.

Usage Notes:

- The data length must be a multiple of 16 bytes.
- The key length must be 32 bytes.
- For Assembler, call the CRYA25EE vcon with the same parameters.

cry_aes256_ecb_decrypt( )

This function uses the AES 256 algorithm in ECB (Electronic Feedback) mode to decrypt the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_aes256_ecb_decrypt(&input, input_length, &key,
                   key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	Description
input	Reference to the data to be decrypted.
input_length	Length of the data to be decrypted.
key	Reference to the key data for the decryption.
key_length	Length of the key data for the decryption.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	Description
Zero	Successful completion.
Negative value	An error occurred. See the section “Debugging Problems” on page 194.

Usage Notes:

- The data length must be a multiple of 16 bytes.
- The key length must be 32 bytes.
- For Assembler, call the CRYA25DE vcon with the same parameters.

cry_des_cbc_encrypt( )

This function uses the DES algorithm in CBC (Cipher Block Chaining) mode to encrypt the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_des_cbc_encrypt(&input, input_length, &key,
                 key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	Description
input	Reference to the data to be encrypted.
input_length	Length of the data to be encrypted.
key	Reference to the key data for the encryption.
key_length	Length of the key data for the encryption.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	Description
Zero	Successful completion.
Negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- The data length must be a multiple of 8 bytes.
- The key length must be 16 bytes. The first 8 bytes contain the initialization vector, followed by the 8-byte key.
- For Assembler, call the CRYDECBE vcon with the same parameters.

cry_des_cbc_decrypt( )

This function uses the DES algorithm in CBC (Cipher Block Chaining) mode to decrypt the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_des_cbc_decrypt(&input, input_length, &key,
                  key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	**Description**
input	Reference to the data to be decrypted.
input_length	Length of the data to be decrypted.
key	Reference to the key data for the decryption.
key_length	Length of the key data for the decryption.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	**Description**
Zero	Successful completion.
Negative value	An error occurred. See the section “Debugging Problems” on page 194.

Usage Notes:

- The data length must be a multiple of 8 bytes.
- The key length must be 16 bytes. The first 8 bytes contain the initialization vector, followed by the 8-byte key.
- For Assembler, call the CRYDECBD vcon with the same parameters.

cry_des_encrypt( )

This function uses the DES algorithm to encrypt the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_des_encrypt(&input, input_length, &key,
                    key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	Description
input	Reference to the data to be encrypted.
input_length	Length of the data to be encrypted.
key	Reference to the key data for the encryption.
key_length	Length of the key data for the encryption.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	Description
Zero	Successful completion.
Negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- The data length must be a multiple of 8 bytes, and the key length must be 8 bytes.
- For Assembler, call the CRYDESEC vcon with the same parameters.

cry_des_decryt( )

This function uses the DES algorithm to decrypt the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_des_decrypt(&input, input_length, &key,
                    key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	Description
input	Reference to the data to be decrypted.
input_length	Length of the data to be decrypted.
key	Reference to the key data for the decryption.
key_length	Length of the key data for the decryption.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	Description
Zero	Successful completion.
Negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- The data length must be a multiple of 8 bytes, and the key length must be 8 bytes.
- For Assembler, call the CRYDESDC vcon with the same parameters.

cry_gen_random( )

This function uses a crypto-coprocessor hardware card to generate a random number.

The syntax is as follows:

```
#include <sslvse.h>
int cry_gen_random(&input, input_length, &key,
                   key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	**Description**
input	Pointer to the random data area.
input_length	Length of the random data area. The maximum is 2048.
key	Must be supplied but is not used.
key_length	Must be supplied but is not used.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	**Description**
Positive value	Successful completion. The value is the actual number of random bytes generated and should be equal to the passed input length.
Negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- This function requires a crypto-coprocessor card.
- For Assembler, call the CRYGENRA vcon with the same parameters.

cry_get_cert_info( )

This function retrieves subject and issuer information from a PKI X.509v3 certificate.

The syntax is as follows:

```
#include <sslvse.h>
int cry_get_cert_info(&input, input_length, &key,
                   key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	**Description**
input	Reference to the input X.509v3 certificate to be decoded.
input_length	Length of the input certificate.
key	Reference to the area to contain the certificate information.
key_length	Length of the certificate information area. It must be greater than 64 bytes.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes.

The return codes are as follows:

Return Code	**Description**
Positive value	Successful completion. The value is the length of the certificate information returned.
Negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- The certificate information that is returned is the same as the cert_info structure that is returned from a gsk_secure_soc_init with client authentication.
- For Assembler, call the CRYGCRIN vcon with the same parameters.

cry_hmac_md5( )

This function creates an MD5 keyed hash, also known as a MAC (Message Authentication Code) of the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_hmac_md5(&input, input_length, &key,
                   key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	Description
input	Reference to the input data for the hash.
input_length	Length of the input data for the hash.
key	Reference to the key data for the MAC.
key_length	Length of the key data for the MAC.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	Description
Zero	Successful completion.
Negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- A 16-byte MD5 MAC is returned in the user-supplied work area.
- For Assembler, call the CRYHMMD5 vcon with the same parameters.

cry_hmac_sha( )

This function creates a SHA keyed hash, also known as a MAC (Message Authentication Code) of the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_hmac_sha(&input, input_length, &key,
                  key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	**Description**
input	Reference to the input data for the hash.
input_length	Length of the input data for the hash.
key	Reference to the key data for the MAC.
key_length	Length of the key data for the MAC.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	**Description**
Zero	Successful completion.
Negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- A 20-byte SHA-1 MAC is returned in the user-supplied work area.
- For Assembler, call the CRYHMSHA vcon with the same parameters.

cry_initialize( )

This function initializes the environment for cryptography functions.

The syntax is as follows:

```
#include <sslvse.h>
int cry_initialize(&input, input_length, &key,
                   key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	Description
input	Not used.
input_length	Not used.
key	Not used.
key_length	Not used.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	Description
Zero	Successful completion.
Negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- This initialization call must be made before you issue any other crypto calls.
- This call opens and reads the RSA certificate, root, and key files.
- For Assembler, call the CRYINITI vcon with the same parameters.

cry_md5_hash( )

This function uses the MD5 (RSA Message Digest 5) algorithm to create a hash of the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_md5_hash(&input, input_length, &key,
                   key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	Description
input	Reference to the input data for the hash.
input_length	Length of the input data for the hash.
key	Must be supplied but is not used.
key_length	Must be supplied but is not used.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	Description
Zero	Successful completion.
Negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- A 16-byte MD5 hash is returned in the user-supplied work area.
- For Assembler, call the CRYMD5HA vcon with the same parameters.

cry_rsa_encrypt( )

This function uses the RSA PKCS #1 version 1.5 algorithm to encrypt the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_rsa_encrypt(&input, input_length, &key,
                   key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	Description
input	Reference to the data to be encrypted.
input_length	Length of the data to be encrypted.
key	Reference to the RSA key to be used.
key_length	Length of the RSA key data.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	Description
Positive value	Successful completion. The value is the length of the RSA encrypted block. The encrypted block overlays the input data area.
Negative value	An error occurred. See the section “Debugging Problems” on page 194.

Usage Notes:

- If the key length is zero, then the RSA key is read from the PRVKFIL key file.
- If the key length is not zero, then the caller must supply the RSA key.
- The RSA key size determines the size of the encrypted block. When using a 512-bit key, the encrypted block is 64 bytes. When using a 1024-bit key, the encrypted block is 128 bytes.
- For Assembler, call the CRYRSAEC vcon with the same parameters.

cry_rsa_decrypt( )

This function uses the RSA PKCS #1 version 1.5 algorithm to decrypt the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_rsa_decrypt(&input, input_length, &key,
                   key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	**Description**
input	Reference to the data to be decrypted.
input_length	Length of the data to be decrypted.
key	Reference to the RSA key to be used.
key_length	Length of the RSA key.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	**Description**
Positive value	Successful completion. The value is the length of the decrypted data. The decrypted data overlays the input data area.
Negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- If the key length is zero, then the RSA key is read from the PRVKFIL key file.
- If the key length is not zero, then the caller must supply the RSA key.
- For Assembler, call the CRYRSADC vcon with the same parameters.

cry_rsa_genprvk( )

This function uses a crypto-coprocessor card to generate an RSA private key.

The syntax is as follows:

```
#include <sslvse.h>
int cry_rsa_genprvk(&area, area_length, &key,
                   key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	**Description**
area	Pointer to the RSA private key area.
area_length	Length of the RSA private key area.
key	Must be supplied but is not used.
key_length	Pointer to a fullword area that contains the size of the RSA private key to be generated. This must be 1024, 2048, or 4096.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	**Description**
Positive value	Successful completion. The value is the number of bytes in the generated RSA key blob. The *area* then contains the generated RSA private key blob.
Negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- This function requires a crypto-coprocessor card.
- For Assembler, call the CRYGRSAP vcon with the same parameters.

cry_rsa_signature_create( )

This function creates a digital signature based on the RSA PKCS #1 version 1.5 standard.

The syntax is as follows:

```
#include <sslvse.h>
int cry_rsa_signature_create(&input, input_length, &key,
                    key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	Description
input	Reference to the input data for the signature.
input_length	Length of the input data.
key	Reference to the RSA key to be used.
key_length	Length of the RSA key.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	Description
Positive value	Successful completion. The value is the length of the RSA signature block. The signature block overlays the input data area.
Negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- If the key length is zero, then the RSA key is read from the PRVKFIL key file.
- If the key length is not zero, then the caller must supply the RSA key.
- The RSA key size determines the size of the signature block. When using a 512-bit key, the signature block is 64 bytes. When using a 1024-bit key, the signature block is 128 bytes.
- For Assembler, call the CRYRSASC vcon with the same parameters.

cry_rsa_signature_verify( )

This function verifies a digital signature based on the RSA PKCS #1 version 1.5 standard.

The syntax is as follows:

```
#include <sslvse.h>
int cry_rsa_signature_verify(&input, input_length, &key,
                        key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	Description
input	Reference to the input data for the signature verification.
input_length	Length of the input data.
key	Reference to the RSA key to be used.
key_length	Length of the RSA key.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	Description
Positive value	Successful completion. The value is the length of the signature data. The signature block overlays the input data area.
Negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- If the key length is zero, then the RSA key is read from the PRVKFIL key file.
- If the key length is not zero, then the caller must supply the RSA key.
- For Assembler, call the CRYRSASV vcon with the same parameters.

cry_sha_hash( )

This function uses the SHA (Secure Hash Algorithm) to create a hash of the passed data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_sha_hash(&input, input_length, &key,
                  key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	**Description**
input	Reference to the input data for the hash.
input_length	Length of the input data for the hash.
key	Must be supplied but is not used.
key_length	Must be supplied but is not used.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	**Description**
Zero	Successful completion.
Negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- A 20-byte SHA-1 hash is returned in the user-supplied work area.
- For Assembler, call the CRYSHAHA vcon with the same parameters.

cry_sha2_hash( )

This function creates a SHA-256 message hash from the passed input data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_sha2_hash(&input, input_length, &key,
                   key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	**Description**
input	Pointer to the input data area.
input_length	Length of the input data area.
key	Must be supplied but is not used.
key_length	Must be supplied but is not used.
workarea	Reference to a work area.
work_length	Length of the work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	**Description**
Zero	Successful completion. A 32-byte, SHA 256-bit hash is returned in the work area.
Negative value	An error occurred. See the section “Debugging Problems” on page 194.

Usage Notes:

- This function requires hardware support for the KLMD instruction, which should be available on a z10 or higher processor. Check with your IBM hardware provider to verify that your system supports this function.
- For Assembler, call the CRYSHA2H vcon with the same parameters.

cry_universal_print_encode( )

This function converts binary data into universally printable characters.

The syntax is as follows:

```
#include <sslvse.h>
int cry_universal_print_encode(&input, input_length,
             &key, key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	Description
input	Reference to the input data for the encoding.
input_length	Length of the input data.
key	Must be specified but is not used.
key_length	Must be specified but is not used.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	Description
Zero	Successful completion.
Negative value	An error occurred. See the section “Debugging Problems” on page 194.

Usage Notes:

- The input data length must be 48 bytes.
- The 64 bytes of universally printable characters are returned in the user-supplied work area.
- For Assembler, call the CRYUPENC vcon with the same parameters.

cry_universal_print_decode( ) This function converts universally printable characters back to binary data.

The syntax is as follows:

```
#include <sslvse.h>
int cry_universal_print_decode(&input, input_length,
             &key, key_length, &workarea, work_length);
```

The parameters are described in the following table:

Parameter	**Description**
input	Reference to the input data for the decoding.
input_length	Length of the input data.
key	Must be specified but is not used.
key_length	Must be specified but is not used.
workarea	Reference to a work area.
work_length	Length of the passed work area. It must be at least 256 bytes long.

The return codes are as follows:

Return Code	**Description**
Positive value	Successful completion. The value is the length of the binary data.
Negative value	An error occurred. See the section "Debugging Problems" on page 194.

Usage Notes:

- The input data length must be 64 bytes.
- The 48 bytes of binary data are returned in the user-supplied work area.
- For Assembler, call the CRYUPDEC vcon with the same parameters.

Common Encryption Cipher Interface

The Common Encryption Cipher Interface (CECI) is an interface applications can call from Assembler, COBOL, or other high level languages using standard call/save linkage conventions. It is designed to separate the coding of the encryption algorithm and key values used in an application program from the application itself. A separate phase is created that contains the cipher algorithm and key-seed values from which the keys are generated. Encryption and decryption can then be performed in the application, and the key seed and algorithm can be changed without having to change the application. You also can add a secure hash (SHA-1 or SHA-2) to guarantee integrity of each encrypted data block. You define the phase name and specify the default cipher and key-seed number that applications can use in a CIALSEED JCL.

This interface meets specific auditing requirements of the VISA CISP standard. This standard has been incorporated into the more encompassing “Payment Card Industry (PCI) Data Security Standard” (DSS), which is published and enforced by the Payment Card Industry Security Standards Council. More information on this standard is available at www.pcisecuritystandards.org.

Calling the CIALCECI Stub

To implement the interface, your application must call the CIALCECI stub and pass the request area as the first parameter for all requests. The CIALCECI stub loads and calls the CIALCECZ phase, which performs the work. These components are distributed with the TCP/IP FOR VSE 2.2 software. The application must include the TCP/IP lib.sublib in the phase libdef search chain in the partition where the application is running.

In Assembler programs, the stub can be called as follows:

```
       LA     R1,MEMADSCT      Address of request area
       ST     R1,PARM1         Store request area addr as 1st parm
       LA     R1,PARMLIST      Set address of parmlist into R1
       L      R15,=V(CIALCECI) Load entry address of CIALCECI stub
       BASR   R14,R15          Enter CIALCECI
....
PARMLIST   DS       0F
PARM1    DS       F
PARM2    DS       F
PARM3    DS       F
PARM4    DS       F
```

In COBOL programs, the stub can be called using the CALL command, as follows:

```
CALL 'CIALCECI' USING CIALCECI-RQSTAREA
```

Request Area

All calls require a request area to be passed using standard call/save linkage. The request area fields for COBOL and Assembler are listed in the following sections. A list of the COBOL constants follows the COBOL fields list.

Note: In this chapter, request area fields are referenced by their COBOL name. The corresponding Assembler name follows in parentheses.

The field CIALCECI-EYE-CATCHER (MEMAEYEC) must be set to "MEMAEYEC" and placed at the beginning of the request area.

COBOL Fields

The COBOL request area fields are as follows:

```
01  CIALCECI-RQSTAREA.
           05  CIALCECI-EYE-CATCHER         PIC X(8).
           05  CIALCECI-RQSTAREA-LEN        PIC S9(8) COMP.
           05  CIALCECI-REQ-CODE            PIC S9(8) COMP.
      *         0  -  INITIALIZE REQUEST
      *         4  -  CREATE CIPHER KEY
      *         8  -  ENCRYPT BLOCK
      *        12  -  ENCRYPT DONE
      *        16  -  DECRYPT BLOCK
      *        20  -  ERASE KEY
      *        24  -  RELEASE TOKEN
           05  CIALCECI-RC                  PIC S9(4) COMP.
      *         0  -  REQUEST SUCCESSFULLY PROCESSED
           05  CIALCECI-EXRC                   PIC S9(4) COMP.
      *         2  -  ENCRYPT DATA BUFFERED NO DATA BLK TO WRITE
      *         4  -  ENCRYPT DATA BUFFERED WRITE DATA BLOCK
           05  CIALCECI-ERROR-DISP          PIC S9(8) COMP.
           05  CIALCECI-ERROR-REASON        PIC X(8).
           05  CIALCECI-KEY-PHASE           PIC X(8).
           05  CIALCECI-MAX-BLKSIZE         PIC S9(8) COMP.
           05  CIALCECI-TOKEN               PIC S9(8) COMP.
           05  CIALCECI-CURRENT-CIPHER      PIC S9(8) COMP.
           05  CIALCECI-CURRENT-KEY-NBR     PIC S9(8) COMP.
           05  CIALCECI-DATA-LEN            PIC S9(8) COMP.
           05  CIALCECI-DATA-ADDR           USAGE IS POINTER.
           05  CIALCECI-OUTPUT-LEN          PIC S9(8) COMP.
           05  CIALCECI-OUTPUT-ADDR         USAGE IS POINTER.
           05  CIALCECI-CIPHER-DESC         PIC X(16).
           05  FILLER                       PIC X(8).
```

COBOL Constants

The COBOL request area constants are as follows:

```
01  CIALCECI-CONSTANTS.
05  CAILCECI-REQ-INIT            PIC S9(4) COMP VALUE +0.
05  CAILCECI-REQ-GEN-KEY         PIC S9(4) COMP VALUE +4.
05  CAILCECI-REQ-ENCRYPT         PIC S9(4) COMP VALUE +8.
05  CAILCECI-REQ-ENCRYPT-DONE    PIC S9(4) COMP VALUE +12.
05  CAILCECI-REQ-DECRYPT         PIC S9(4) COMP VALUE +16.
05  CAILCECI-REQ-ERASE-KEY       PIC S9(4) COMP VALUE +20.
05  CAILCECI-REQ-REL-TOKEN       PIC S9(4) COMP VALUE +24.
05  CAILCECI-RC-HAVE-DATA        PIC S9(4) COMP VALUE +4.
```

Assembler Fields

The Assembler request area fields are listed in the following table. Equate field names are indented.

Field Name	Storage	Description
MEMADSCT DSECT	OD	
MEMAEYEC DS	CL8	Eye catcher
MEMALENG DS	F	Length of this request block
MEMARQST DS	F	Request code
MEMAINIT EQU	0	Initialize request
MEMACRKY EQU	4	Create cipher key
MENAEBLK EQU	8	Encrypt block
MEMAEDON EQU	12	Encrypt done
MEMADBLK EQU	16	Decrypt block
MEMAXKEY EQU	20	Erase key
MEMARTOK EQU	24	Release token
MEMARCOD DS	H	Return code
MEMARCOK EQU	0	Request successful
MEMAEXRC DS	H	Extended return code
MEMAE002 EQU	2	Good MEMAEBLK, no data write
MEMAE004 EQU	4	Good MEMAEBLK, data to write
MEMAEDSP DS	F	Error displacement
MEMAERSN DS	CL8	Error reason (displayable)
MEMAKYPH DS	CL8	Key phrase phase name (CIALSEED)
MEMAMAXB DS	F	Maximum block size
MEMATOKN DS	F	Token for all requests
MEMACIPH DS	F	Current cipher
MEMAKNUM DS	F	Current key number
MEMALDAT DS	F	Length of passed data
MEMA@DAT DS	A	Address of passed data
MEMALOUT DS	F	Length of output data

Field Name	Storage	Description
`MEMA@OUT DS`	`A`	Address of output data
`MEMACIDE DS`	`CL16`	Cipher description
`DS`	`D`	Reserved
`MEMARQSL EQU`	`*-MEMAEXRC`	Length of the entire request area

Return Codes

The application should check the return code after completing each call to the CIALCECI stub. For assembler programs, this is in R15. For COBOL programs, this is in the COBOL-reserved RETURN-CODE field. The CIALCECI-RC (MEMARCOD) field also contains the return code unless the request area does not start with the required MEMAEYEC eye-catcher. In both cases (in R15 and in RETURN-CODE), a value of 0 indicates that the request was processed successfully.

If a request fails with a non-zero return code, a CCZ601 error message is issued that contains the reason the request failed. The reason code is also set in the CIALCECI-ERROR-REASON (MEMAERSN) field in the request area. The displacement of the error in the CIALCECZ phase is set in the CIALCECI-ERROR-DISP (MEMAEDSP) field.

There is an exception such that if the CDLOAD for the CIALCECZ.phase fails, then the return code is set to -101. In this case, the CIALCECI-RC (MEMARCOD) field is not set.

Encryption Pseudocode

This section describes pseudocode to implement encryption in an application. You must make the following requests. The remainder of this section explains each step.

1. Initialize request
2. Create key material request
3. Encrypt-data request
4. Encrypt-done request
5. Erase-key request
6. Release-token request

1. Initialize

For the initialize request, the application must set the following fields:

- CIALCECI-REQ-CODE (MEMAEXRC) must be set to CAILCECI-REQ-INIT (MEMAINIT).
- CIALCECI-MAX-BLKSIZE (MEMAMAXB) must contain the maximum block size that is passed by the application. This value must be a multiple of 16, and it must be between 256 and 65536 (64K).

- CIALCECI-KEY-PHASE (MEMAKYPH) must contain the name of the generated phase that defines the cipher and available key seeds. See the section CIALSEED JCL for more information on generating this phase. Key seeds are explained in the next step.

After the initialize completes, the request area contains the address of a token that has been allocated. You must pass the same request area and only change the fields needed for a specific request.

2. Create Key

This request causes key seed phrase data you specify in the CIALSEED JCL to be input to a pseudo-random-function (PRF) generator. This generator creates the key material for the cipher being used.

What is a PRF generator and why do we use it? Different cipher algorithms require different amounts (number of bytes) of key material. We can use a PRF generator with any phrase to generate the right amount of key material for a selected cipher. Part of the input to the generator is a seed phrase you select that can be any alphanumeric or binary value. This approach creates a very secure key and does not leave the key's actual value exposed.

You can define multiple seed phrases in the CIALSEED JCL. Each seed phrase definition is associated with a key number. You must set a default key number in the JCL from this group of phrase definitions. You also must set a default cipher algorithm/hash.

For the create-key request, the application must set the following fields:

- CIALCECI-REQ-CODE (MEMAEXRC) must be set to CAILCECI-REQ-GEN-KEY (MEMACRKY).
- CIALCECI-CURRENT-CIPHER (MEMACIPH) and CIALCECI-CURRENT-KEY-NBR (MEMAKNUM) are set to the default cipher/hash and key number, respectively, during the previous initialize request from the values defined in the CIALSEED-generated phase.

You can override the default values in an application by setting these request area fields to other valid values. The cipher/hash combinations you can select are listed in the sample JCL in a later section. The key number you use must be defined in the CIALSEED-generated phase. See Cipher Suite Selection for more information on selecting an appropriate cipher/hash combination.

3. Encrypt Data

You can pass any number of bytes less than or equal to the CIALCECI-MAX-BLKSIZE (MEMAMAXB) value.

For the encrypt-data request, the application must set the following fields:

- CIALCECI-REQ-CODE (MEMAEXRC) must be set to CAILCECI-REQ-ENCRYPT (MEMAEBLK).

- CIALCECI-DATA-LEN (MEMALDAT) must be set to the length of the data to be encrypted. It must be greater than zero and less than or equal to the value of CIALCECI-MAX-BLKSIZE (MEMAMAXB), which was set during the previously issued initialize request.

The encrypt-data request also requires two other parameters to be passed in the parameter list:

- The second parameter is the address of the input data to be encrypted.
- The third parameter is the address of the output buffer for encrypted data.

The output buffer area should be allocated equal to the CIALCECI-MAX-BLKSIZE (MEMAMAXB) size.

If the request completes successfully, that is, CIALCECI-RC (MEMARCOD) equals ZERO, then a return code of 2 or 4 is set in CIALCECI-EXRC (MEMAEXRC). These codes have the following meanings.

Return Code	Description
2	The application's data was buffered, but a full encrypted block is not available yet for the application to write.
4	The application's data was buffered, and now a full encrypted block is available for the application to write. The length of the encrypted block is set in the CIALCECI-OUTPUT-LEN (MEMALOUT) field. The output buffer passed as the third parameter of the encrypt-data request contains the encrypted block of data for the application to write out to a disk, tape, or other device. The encrypted block may have a SHA-1 or a SHA-2 hash on the end of each block, depending on the cipher used.

4. Encrypt Done

Once the application has sent all of the data to be encrypted, it must make an encrypt-done request. No input data is passed in this request, and it is similar to a close request.

For this request, the application must set CIALCECI-REQ-CODE (MEMARQST) equal to CAILCECI-REQ-ENCRYPT-DONE (MEMAEDON).

If the request completes successfully, that is, CIALCECI-RC (MEMARCOD) equals ZERO, then an extended return code of 2 or 4 is set in CIALCECI-EXRC (MEMAEXRC). These codes have the following meanings:

Return Code	Description
2	There is no encrypted data block for the application to write.
4	There is a final encrypted data block for the appl. to write. The size of this last encrypted block is probably smaller than the value of CIALCECI-MAX-BLKSIZE (MEMAMAXB). The length of this block is set in the CIALCECI-OUTPUT-LEN (MEMALOUT) field. The output buffer passed as the third parameter of the prior encrypt-data request contains the last encrypted block of data for the application to write out to a disk, tape, or other device.

5. Erase Key

The erase-key request clears the current key to binary zeros in memory. For this request, the application must set CIALCECI-REQ-CODE (MEMARQST) equal to CAILCECI-REQ-ERASE-KEY (MEMAXKEY).

6. Release Token

The release-token request releases the allocated token storage. For this request, the application must set CIALCECI-REQ-CODE (MEMAEXRC) equal to CAILCECI-REQ-REL-TOKEN (MEMARTOK).

Encryption Flow

Data is buffered internally based on the value of CIALCECI-MAX-BLKSIZE (MEMAMAXB). CECI adds a pad of 1 to 15 bytes plus a 32-byte hash on the end of each block encrypted with a SHA-1, SHA-2, or null hash. As you pass data to be encrypted, you are notified that a full encrypted block is ready to be written.

This is also true for CIALCECI-REQ-ENCRYPT-DONE (MEMAEDON), which almost always completes with the last encrypted block for you to write. All blocks prior to the last one are equal to CIALCECI-MAX-BLKSIZE (MEMAMAXB). The last block is less than or equal to this size. Because CECI buffers the data on encrypt-data requests, there may or may not be a partial buffer of data when the encrypt-done request is issued. It may be that the last encrypt-data request returns an encrypted block and nothing is left over after the done request. In that case, a return code of 2 is set in CIALCECI-EXRC (MEMARCOD).

For example, if CIALCECI-MAX-BLKSIZE (MEMAMAXB) is set to 4096 and you send 4063 bytes during an encrypt, you would receive CIALCECI-EXRC equal to CIALCECI-RC-HAVE-DATA with the CIALCECI-OUTPUT-LEN field equal to 4096. This block, when decrypted, would contain your 4063 bytes of data, a one-byte pad with x01, and then a 32-byte area on the end with a SHA-1, SHA-2, or null hash. When decrypted, you would receive a length of 4063 (padding and hash removed). On the CIALCECI-REQ-ENCRYPT-DONE (MEMAEDON) request, you would receive CIALCECI-EXRC (MEMARCOD) equal to CIALCECI-RC-NO-DATA (MEMAE002).

Decryption Pseudocode

You must make the following requests to implement decryption in an application:

1. Initialize request
2. Create-key request
3. Decrypt-data request
4. Erase-key request
5. Release-token request

Detailed information on each step follows.

1. Initialize

For the initialize request, the application must set the following fields:

- CIALCECI-REQ-CODE (MEMAEXRC) must be set to CAILCECI-REQ-INIT (MEMAINIT).
- CIALCECI-MAX-BLKSIZE (MEMAMAXB) must contain the maximum block size that was used during the encryption process by the application. This value must be a multiple of 16, and it must be between 256 and 65536 (64K).
- CIALCECI-KEY-PHASE (MEMAKYPH) must contain the phase name of the generated cipher and key seeds.

After the initialize completes, the request area contains the address of a token that has been allocated. You must pass the same request area and only change the fields needed for a specific request.

2. Create Key

For the create-key request, the application must set the following fields:

- CIALCECI-REQ-CODE (MEMAEXRC) must be set to CAILCECI-REQ-GEN-KEY (MEMACRKY).
- CIALCECI-CURRENT-CIPHER (MEMACIPH) and CIALCECI-CURRENT-KEY-NBR (MEMAKNUM) are set to the default cipher and key number, respectively, which were set during the previous initialize request.

If these fields were set to other values in the application at encryption (in the create-key request), they must be set to the same values here.

3. Decrypt Data

You must decrypt each block separately to decrypt the previously encrypted data. For the decrypt-data request, the application must set the following fields:

- CIALCECI-REQ-CODE (MEMAEXRC) must be set to CAILCECI-REQ-DECRYPT (MEMADBLK).
- CIALCECI-DATA-LEN (MEMALDAT) must be set to the length of the block being decrypted, which should be the same as the value of CIALCECI-MAX-BLKSIZE (MEMAMAXB) used during the encryption process (except for the last block of data in the file).

The second parameter passed in the parm list is the address of the encrypted block to be decrypted. The third parameter passed is the address at which the decrypted data is to be stored upon successful completion. This third parameter can be the same as the second parameter, in which case the data is decrypted in place.

After a successful decrypt request, the actual length of the decrypted data (minus the pad and hash) is set in the CIALCECI-OUTPUT-LEN (MEMALOUT) field.

4. Erase Key

The erase-key request clears the current key to binary zeros in memory. For this request, the application must set CIALCECI-REQ-CODE (MEMAEXRC) equal to CAILCECI-REQ-ERASE-KEY (MEMAXKEY).

5. Release Token

The release-token request releases the allocated token storage. For this request, the application must set CIALCECI-REQ-CODE (MEMAEXRC) equal to CAILCECI-REQ-REL-TOKEN (MEMARTOK).

CIALSEED JCL

You must edit and run a CIALSEED JCL to generate a phase and create required definitions. These definitions are described below. Each is used in a sample JCL that appears in the next section.

Definitions

Phase Name. The CIALSEED JCL generates the phase that an application must specify in the CIALCECI-KEY-PHASE (MEMAKYPH) field in the request area. A PUNCH command is used to define the phase name.

> **WARNING:** If the phase defined in the CIALSEED JCL is deleted, any data encrypted with its ciphers and key values will be permanently lost.
>
> You must establish appropriate procedures for maintaining, recovering, and destroying this critical phase. The following scenarios show the importance of good procedures in managing this phase.

Good Example: The data center is in an unfriendly country and must be evacuated suddenly. You simply delete this phase, and all data encrypted with the CECI interface is secure and unusable by unfriendly invaders. You also have a copy of the phase at a secure, remote location that authorized persons can access.

Bad Example: You implemented the API in a production environment and a natural disaster occurs that destroys the data center. Your data is backed up and ready to be restored at a disaster recovery site, but you neglected to include the CIALSEED-generated source code or phase into your disaster recovery plans. The restored data is encrypted and cannot be deciphered. Effectively, the data is lost.

Seed Phrases and Key Numbers. The CIALSEED macro and SEED statement defines both a key seed phrase and an associated key number. The PHRASE keyword can be set to any alphanumeric value up to 255 bytes long. The XPHRAS keyword can be set to any binary value (represented in hex EBCDIC characters). Each phrase must be enclosed in single quotes. You can define multiple phrases/key numbers.

Default Key Number. The CIALSEED macro and DEFLTKY keyword sets the default key-seed number used by applications. If you change keys or start an encrypt/decrypt operation, you must issue CAILCECI-REQ-GEN-KEY (MEMACRKY) using a key number defined in the CIALSEED-generated phase.

Default Cipher. The DEFLTCI keyword sets the default cipher algorithm and hash. The cipher/hash combinations you can select are listed in comments in the sample JCL. Applications can override this value in the create-key request.

See the section “Cipher Suite Selection,” page 193, for more information on selecting a cipher/hash combination.

Example

The following JCL is an example you can examine and modify. The phase it generates is named “SEEDSAMP.”

Strings representing the cipher, hash, and cipher-mode combinations you can select are listed in the file comments. For example, the first entry in the list is ACNULSH1.

```
// JOB CIALSEED
// OPTION CATAL
// LIBDEF *,CATALOG=lib.sublib
// EXEC ASMA90,SIZE=ASMA90
         PUNCH    ' PHASE SEEDSAMP,* '
SEEDSAMP CIALSEED  BEGIN,DEFLTKY=1,DEFLTCI=ACA12SH1
*
* * Available cipher equates for DEFLTCI=
*              ACNULSH1  08 NULL-SHA1 (no encrypt)
*              ACDESNUL  12 SDES-NULL CBC mode
*              ACDESSH1  16 SDES-SHA1 CBC mode
*              ACTDENUL  20 TDES-NULL CBC mode
*              ACTDESH1  24 TDES-SHA1 CBC mode
*              ACA12NUL  28 AES128-NULL CBC mode
*              ACA12SH1  32 AES128-SHA1 CBC mode
*              ACA19NUL  36 AES192-NULL CBC mode
*              ACA19SH1  40 AES192-SHA1 CBC mode
*              ACA25NUL  44 AES256-NULL CBC mode
*              ACA25SH1  48 AES256-SHA1 CBC mode
*              ACA12SH2  52 AES128-SHA2 CBC mode
*              ACA19SH2  56 AES192-SHA2 CBC mode
*              ACA25SH2  60 AES256-SHA2 CBC mode
*              AEA12SH1  64 AES128-SHA1 ECB mode
*              AEA19SH1  68 AES192-SHA1 ECB mode
*              AEA25SH1  72 AES256-SHA1 ECB mode
*              AEA12SH2  76 AES128-SHA2 ECB mode
*              AEA19SH2  80 AES192-SHA2 ECB mode
*              AEA25SH2  84 AES256-SHA2 ECB mode
*              AEDESNUL  88 SDES-NULL ECB mode
*              AEDESSH1  92 SDES-SHA1 ECB mode
*              AETDENUL  96 TDES-NULL ECB mode
*              AETDESH1 100 TDES-SHA1 ECB mode
*
         CIALSEED  SEED,KEY=1,                                         X
               PHRASE='SEEDSAMP sample key phrase 1'
         CIALSEED  SEED,KEY=2,                                         X
               PHRASE='SEEDSAMP sample key phrase 2'
         CIALSEED  SEED,KEY=3,                                         X
               PHRASE='SEEDSAMP sample key phrase 3'
         CIALSEED  SEED,KEY=4,                                         X
               XPHRAS='04F9C16E02BA6620CB1DE0F6671348C190220AD331CB'
         CIALSEED  SEED,KEY=5,                                         X
               XPHRAS='D409712E823A2617C011E6F6371548C99112468321BB'
         CIALSEED END
      END
/*
// EXEC LNKEDT,SIZE=512K
/*
/&
```

Cipher Suite Selection

You must select a cipher suite and related values carefully:

- Choose a *cipher strength* that meets your auditing standards but does not require an excessive CPU overhead. Generally speaking, the stronger the cipher, the higher the CPU overhead.

  Selecting a cipher that is supported by a cryptographic hardware assist (CPACF) can greatly reduce the CPU overhead. The available hardware assists can be checked by issuing the following z/VSE command to the security server partition:

```
MSG FB,DATA=STATUS=CRYPTO
  BST223I CURRENT STATUS OF THE SECURITY TRANSACTION SERVER:
  ADJUNCT PROCESSOR CRYPTO SUBTASK STATUS:
    AP CRYPTO SUBTASK STARTED ......... : NO
  CPU CRYPTOGRAPHIC ASSIST FEATURE:
    CPACF AVAILABLE ................... : YES
    INSTALLED CPACF FUNCTIONS:
      DES, TDES-128, TDES-192
      AES-128
      SHA-1, SHA-256
END OF CPACF STATUS
```

  The available block ciphers are Data Encryption Standard (DES) and Advanced Encryption Standard (AES). AES has replaced DES and is more efficient and secure. For more information on cryptographic algorithms, go to http://csrc.nist.gov/groups/ST/toolkit/index.html.

- Choose the *hash type*—SHA or null—to use with the cipher. The SHA hash adds some CPU overhead, but it provides integrity for the encrypted data. The AES and DES block encryption ciphers provide secrecy and confidentiality for data, but they do not prevent the encrypted data from being modified. The secure hash (SHA) guarantees that the encrypted data has not been modified.

- Choose the *operational mode* for the encryption cipher—either Electronic Feedback (ECB) or Cipher Block Chaining (CBC) mode.

  In ECB mode, each block is independently encrypted, and any randomly accessed block can be decrypted. But blocks containing the same values always encrypt to the same exact values. DES uses an 8-byte block, and AES uses a 16-byte block. So, assuming you use the DES cipher in ECB mode, if you encrypted a 4K block of binary zeros, you would see exactly the same 8-byte encryption pattern for each 8-byte block in the 4K area.

  The CBC mode addresses the problem of repeating patterns by adding an initialization vector to the generated key material. This 8 (DES) or 16 (AES) vector is then exclusive OR'd with the first 8 (DES) or 16 (AES) block of data being encrypted. The next and subsequent blocks are then always exclusive OR'd with the prior encrypted block. This makes the encrypted data much more secure than it would be in ECB mode, but it also means you cannot randomly decrypt any block in the file. The chained blocks must be decrypted from beginning to end.

Debugging Problems

For errors generated by the SSL API, terse explanations of most error codes are contained in the member SSLVSE.A, which is in the TCP/IP FOR VSE library.

To debug errors produced by the SSL API or the CryptoVSE API, you can create a custom $SOCKDBG.PHASE to generate diagnostic information. You can then send this information to CSI technical support for a detailed analysis of the problem.

To enable diagnostics, use the following keyword settings in a custom $SOCKDBG phase. This custom phase must be placed in a test lib.sublib that is located before the default phase in the search chain.

Keyword Setting	Result
FL02=$DBGISON	Activates diagnostics for all components (for example, BSD and SSL)
MSGT=$DBGALL	Specifies that all message categories should be logged (for example, diagnostic and critical)
SSLD=$DBGSDMP	Creates a dump of SSL/TLS handshake records
CIAL=$DBGSDMP	Creates a dump of cryptographic input and output areas

See "Appendix B: $SOCKDBG Debugging Phase," page 203, for information on other keyword settings and the steps to modify this phase. This appendix also contains an example of a modified $SOCKDBG phase.

Programming Notes

The following notes apply when developing SSL/TLS-enabled applications:

- Programs must include the IPCRYPTS.OBJ deck in their link edit JCL. This is a small stub program that loads the IPCRYPTO phase to process the request. When new maintenance is available, you do not have to link edit the user application again.
- The C language linkage standard is used to pass all parameters.
- When the function is called from an Assembler program, the parameter list is always a list of addresses. The high-order bit of the last parameter in the list must be turned on to indicate the end of the parameter list.
- If a parameter is an address-type parameter, the address is stored directly in the parameter list.
- If the parameter is a value-type parameter, the address of the value is stored in the parameter list.
- All passed addresses are validated to ensure that they are within the partition storage from which the request is issued. The following files are used:

SSLVSE.H	Header file for C/C++ language programs.
SSLVSE.A	Macro file for BAL (Assembler) programs.

- Sample code is available from CSI at www.csi-international.com.

Appendix A: $SOCKOPT Options Phase

Using $SOCKOPT

The BSD and the SSL/TLS interfaces in TCP/IP FOR VSE use options set by the $SOCKOPT options phase. The $SOCKOPT.PHASE is CDLOADed into the partition using these interfaces at run time, and the phase can be customized to allow installation-specific values. The default values should suffice for most installations, and CSI technical support should be consulted before changing them. Examples in this appendix show how new values can be defined. The options for the BSD socket interface and the SSL/TLS interface are described in separate tables.

The $SOCKOPT phase is loaded based on the LIBDEF of the partition loading it. This means that if you have multiple copies of the phase, the first one found in a partition's LIBDEF chain is the one used for that partition. In general, you should install only one copy of the phase.

Default Phase

The sample job below creates an options phase with the current default settings. This sample can be modified to create a custom $SOCKOPT.PHASE.

```
// JOB $SOCKOPT
// OPTION CATAL
// LIBDEF *,SEARCH=lib.sublib
// LIBDEF *,CATALOG=lib.sublib
// EXEC ASMA90,SIZE=ASMA90,PARM='SZ(MAX-200K,ABOVE)'
         PUNCH    ' PHASE $SOCKOPT,* '
$SOCKOPT CSECT
         SOCKOPT  CSECT,                   Generate a csect            X
               SYSID=00, TCP/IP sysid when BSDCFG1 contains $OPTUSYD   X
               BSDCFG1=$OPTMECB+$OPTSNWT+$OPTNBSD, Options flag 1      X
               BSDCFG2=$OPTGTSP+$OPTCHKR,                              X
               BSDXPHS=IPNRBSDC,           Name of BSD phase           X
               CHKT=60,                    Check sockets seconds       X
               CLST=30,                    Close timeout seconds       X
               CSRT=30,                    Socket reuse seconds        X
               QUEDMAX=0,                  Max queued connects allowed X
               RCVT=00,                    Receive timeout seconds     X
               SNOWMAX=262144,             256K max for send nowait    X
               SOCFLG1=00,                 Socket flag special options X
               SSLCIPH=A,                  SSL default cipher suites   X
               SSLFLG1=00,                 SSL option flag 1           X
               SSLFLG2=00,                 SSL option flag 2           X
               SSLSTOR=80                  SSL cached sessions
         END   $SOCKOPT
/*
// EXEC LNKEDT,SIZE=512K
/*
/&
```

Example 1: Setting the Stack ID

The following procedure shows how to override both the system-default stack ID and the ID set by the // OPTION SYSPARM= statement in a program's JCL.

1. Copy and paste the sample job above into your local editor.
2. Modify the following keyword settings:

Keyword	Edit the setting to...
BSDCFG1	Include $OPTUSYD (separate options with a '+')
SYSID	Specify a TCP/IP FOR VSE stack ID

3. Modify the CATALOG parameter to contain a lib.sublib in the phase search chain of the partition in which the BSD application executes.
4. Run the modified job to create the custom $SOCKOPT.PHASE.
5. Cycle the partition in which the application (for example, CICS) is running to force a reload of the newly created $SOCKOPT.PHASE.

Example 2: Setting the Default Cipher Suite

This example shows how to change the default cipher suite used by SSL/TLS applications such as SecureFTP. The cipher suite is set during the application's gsk_secure_soc_init( ) call with the v3cipher_specs parameter. If the SSL/TLS application sets this parameter to zero, then it allows the system to use the default cipher suite.

Use the following procedure to override the default cipher suite.

1. Copy and paste the sample job into your local editor.
2. Replace the SSLCIPH setting with one of the following selections:

Cipher Selection	Description
SSLCIPH=A	Use all possible cipher suites
SSLCIPH=D[1]	Use only DES cipher suites
SSLCIPH=E[1]	Use only AES cipher suites
SSLCIPH=N[1]	Use only null cipher suites
SSLCIPH=W	Use only weak cipher suites
SSLCIPH=M	Use only medium cipher suites
SSLCIPH=S	Use only strong cipher suites
SSLCIPH=H[1]	Use only hardware-assisted suites

[1] Does not apply to TLS 1.2. See "New $SOCKOPT Settings for TLS 1.2" on page 201.

3. Modify the CATALOG parameter to contain a lib.sublib in the phase search chain of the partition in which the SSL/TLS application executes.
4. Run the modified job to create the custom $SOCKOPT.PHASE.
5. Cycle the partition in which the application (for example, CICS) is running to force a reload of the newly created $SOCKOPT.PHASE.

Keep in mind that some applications are configured to set the cipher suite to be used. If they are doing this, then the value cannot be overridden by modifying the $SOCKOPT.PHASE. Check with the application's vendor to verify how the cipher suite is set. This applies when the SSL30, the TLS 1.0, or the TLS 1.1 protocol version is used.

Note: When the TLS 1.2 protocol version is used, you can override the application's default cipher suite even when the SIN@V3CS field is not set to zero. To do this, use the SSLFLG1=$OPTCIPH option setting. See "SSL/TLS Interface Options" on page 201.

BSD Interface Options

The following table lists the option settings for the BSD interface. Options for the BSDCFG1 and BSDCFG2 keywords can be concatenated using a '+' character.

Function	Option Setting	Description

Automatic socket cleanup	BSDCFG2=$OPTCHKR
	CHKT=60
		Setting these options causes disconnected but unclosed (dead) sockets to be detected and made available for reuse. CHKT=60 specifies a nominal socket-checking interval of 60 seconds. The socket is marked closed if the connection has been terminated. It may be reused after the CSRT interval.

Bind	BSDCFG1=$OPTBNDX	Have bind check external partitions. By default, bind checks for a port already bound within just the application's partition. This option causes bind to also check all other partitions for a port already bound to a socket.
Close	BSDCFG1=$OPTCNFW	Close no FIN wait from foreign. The application's connection ends immediately upon close. The TCP/IP stack sends a close request (FIN) but discards any incoming datagrams. It does not wait for a FIN from the foreign connection.
	CSRT=30	A socket that is closed from an abort, close, or automatic cleanup becomes available for reuse after the specified number of seconds.
Givesocket/ Takesocket	BSDCFG2=$OPTGTSP	This option causes givesocket/takesocket processing to occur in the same partition as the request, not in the TCP/IP FOR VSE partition. This improves the performance of applications that use a listener transaction to pass connections to other tasks.

Function	Option Setting	Description
Listen	`QUEDMAX=`*`qmax`*	QUEDMAX=*qmax* affects the connection queuing depth for applications that use the BSD socket interface. If *qmax* = 0 (the default), the PORTQUEUE command can be used to control the number of connection requests to be accepted and queued. If *qmax* > 0, an application can specify a queuing depth count in listen( ), but if the application's value exceeds QUEDMAX, the application's value will not be used. See the listen( ) function, page 37, and "Port Queuing" in the "Performance" chapter in the *TCP/IP FOR VSE Installation Guide* for details.
	`SOCFLG1=$OPTDNSQ`	SOCFLG1=$OPTDNSQ turns off queuing of inbound connections when the server is not in a listen state. This means that both the QUEDMAX= value and the application's value will be ignored and new inbound connections will be rejected until the server re-enters the listen state.
Receive	`BSDCFG1=$OPTXNBK`	Extend non-blocking to all calls. In previous releases a receive call could block even on a non-blocking socket. This option extends the non-block to receive calls. An `ewouldblock` error number 1102 can occur during a receive with this option setting.
Selectex	`BSDCFG1=$OPTMECB`	Allow multiple user ECBs in selectex. This is the sixth parameter passed into a selectex call. Without this option, the sixth parameter is considered a single ECB address.

Send	`BSDCFG1=$OPTSNWT`
	`SNOWMAX=262144`
		Setting these options allows up to 256K of unacknowledged data to be sent before waiting for an acknowledgement. When a CLOSE is issued, all data is acknowledged before the CLOSE completes.

System ID	`BSDCFG1=$OPTUSYD`
	`SYSID={`*`id`*`}`
		Setting these options overrides the default stack ID with *id*. See "Example 1: Setting the Stack ID," page 197.

SSL/TLS Interface Options

The following table lists the option settings for the SSL/TLS interface. Settings for the SSLFLG1 and SSLFLG2 keywords can be concatenated using a '+' character.

Function	Option Setting	Description
Cipher suite **Note:** For TLS 1.2, see text below.	SSLCIPH={A|D|E|N|W|M|S|H}	Override the default cipher suite used by SSL/TLS applications. See "Example 2: Setting the Default Cipher Suite," page 198, for a description of each suite-selection letter.
	SSLFLG1=$OPTCIPH	When TLS 1.2 is used, force an override of the application's cipher suite setting and use the SSLCIPH= setting. See text below for details.
Close	SSLFLG1=$OPTSRQC	Require close_notify alert. See the gsk_secure_soc_close( ) function on page 134.
Hardware assist **Note:** These settings do not apply to TLS 1.2; see text below.	SSLFLG2=$OPTSNHC	Do not use hardware cryptography for RSA operations.
	SSLFLG2=$OPTSNZA	Never issue CP Assist Cryptographic Function (CPACF) z/architecture hardware instructions.
	SSLFLG2=$OPTSFZA	Always issue CPACF z/architecture hardware instructions. See also "Hardware Assist Options Settings" on page 125.
Initialize	SSLFLG1=$OPTSNFR	Fast resume not allowed. See "Usage Notes" under gsk_secure_soc_init( ), page 141.
	SSLFLG1=$OPTTLSX	Only allow TLS 1.2 to be negotiated during session initialization.
	SSLSTOR=80	The maximum number of cached sessions stored for an SSL/TLS server application. A value of 80 caches a maximum of 80 SSL/TLS sessions that can then be reused by foreign clients that use fast resume. Using a value of 0 is equivalent to setting SSLFLG1=$OPTSNFR. Each cached session requires about 64K of 31-bit storage.

New $SOCKOPT Settings for TLS 1.2

A custom $SOCKOPT.PHASE can be created to control the behavior of the SSL/TLS protocols. It is CDLOADed into the partition in which the application is running, and it therefore must be cataloged in a lib.sublib in the application's libdef phase search chain.

Changes to $SOCKOPT settings for the TLS 1.2 protocol are as follows.

- By default, older versions of the SSL/TLS protocol are allowed to be used.

  Using SSLFLG1 with the $OPTTLSX setting allows only TLS 1.2 to be negotiated during session initialization—older versions of the protocol are disallowed.

- By default, applications set the cipher suites they use with the SIN@V3CS pointer during the session initialization. If this field is zero, then the SSLCIPH= setting is used. The default is SSLCIPH=A, which allows all supported cipher suites to be used.

  For TLS 1.2, the SSLCIPH option allows the following settings:

  - A is the default and allows all supported cipher suites to be used.
  - S only allows strong cipher suites to be used.
  - M only allows medium cipher suites to be used.
  - W only allows weak cipher suites to be used.

  In addition, you can override the application's cipher suites by setting SSLFLG1 with $OPTCIPH. With $OPTCIPH, the SSLCIPH setting will override the application's setting even when its SIN@V3CS field is not zero.

- When the TLS 1.2 protocol version is negotiated, the hardware assists for both RSA and CPACF must be available, and the SSLFLG2 setting—$OPTSNHC, $OPTSNZA, or $OPTSFZA—is ignored.

See also "Appendix C: TLS 1.2 Enhancement."

Appendix B: $SOCKDBG Debugging Phase

Using $SOCKDBG

You can debug and diagnose problems in BSD and SSL/TLS applications by creating a custom $SOCKDBG phase. You can select the diagnostic information you want to generate by editing a job that catalogs this phase. The information includes messages that can be issued to SYSLOG or SYSLST and dumps of parameters and control blocks. These messages and dumps may be requested by CSI technical support, which you can contact if you need help interpreting the information.

Sample Job

The following sample job contains the current default settings. You can modify this job to create a custom $SOCKDBG.PHASE. The next sections explain how to edit and run this job to generate diagnostics.

```
// JOB $SOCKDBG
// OPTION CATAL
// LIBDEF *,CATALOG=lib.sublib
// EXEC ASMA90,SIZE=ASMA90
         PUNCH    ' PHASE $SOCKDBG,* '
         SOCKDBG  CSECT,        Generate BSD SSL/TLS debugging phase   X
               FL01=$DBGWLST+$DBGWLOG, Messages to syslst and syslog   X
               FL02=$DBGISON,   Debug flag is on                       X
               FL03=$DBGIOFF,   Connection diagnostics are off         X
               MSGT=$DBGVITA+$DBGCRIT+$DBGIMPO+$DBGRESP, Message Types X
               DUMP=$DBGNONE,   Diagnostic dumps $DBGNONE or $DBGSDMP  X
               LSTCCW=09,       CCW write command for syslst           X
               SSLD=$DBGNONE,   Diagnostics for SSL api parameters     X
               CIAL=$DBGNONE    Diagnostic dumps for Crypto functions/*
// EXEC LNKEDT,SIZE=512K
/*
/&
```

Keyword Settings

The following keyword settings control messages and dumps.

- **Message routing**. The FL01 keyword controls message routing:

FL01 Setting	Result
$DBGWLST	Writes messages to the assigned printer (SYSLST) for the partition in which the application executes
$DBGWLOG	Writes messages to the VSE system console
$DBGUPER	Writes messages in upper case only

Note: Use $DBGWLOG with caution. It can increase traffic to the VSE system console when MSGT=$DBGALL (see below) and slow down the associated application's performance. $DBGWLST is recommended and is more efficient for high volume applications.

- **SYSLST CCW operation code**. The LSTCCW keyword sets the CCW operation code to be used for writing messages to SYSLST. The default is 09 for write and then space one line. LSTCCW=0B can be specified to space one line and then write the message. This option can prevent print overlays in some applications.

- **Message source**. The FL02 keyword specifies the message source:

FL02 Setting	Result
$DBGISBS	Enables messages for the BSD interface only
$DBGISON	Enables messages for all components: BSD, SSL/TLS, etc.
$DBGISSL	Enables messages for the SSL/TLS components only
$DBGISCI	Enables messages for the cryptographic component only
$DBGIOFF	Disables debugging for all components

- **Message type**. The MSGT keyword specifies the message type(s):

MSGT Setting	Message Type
$DBGALL	All
$DBGCRIT	Critical
$DBGVITA	Vital
$DBGWARN	Warning
$DBGIMPO	Important

MSGT Setting	Message Type
$DBGINFO	Informational
$DBGRESP	Response
$DBGDIAG	Debug/diagnostic
$DBGSCTY	Security related

See chapter 10, “Operation,” in the *TCP/IP FOR VSE Installation Guide* for more information about these message types.

- **Diagnostics flag**. The FL03 keyword controls connection diagnostics:

FL03 Setting	Result
$DBGDTCP	Activates connection diagnostics for the connections. The additional diagnostic information is sent to TCP/IP’s SYSLST.

- **Diagnostic dumps**. The following settings create diagnostic dumps:

Keyword Setting	Result
DUMP=$DBGSDMP	Creates dumps using the SDUMP service for applications that use the BSD application programming interface
CIAL=$DBGSDMP	Creates dumps using the SDUMP service for applications that use the cryptographic application programming interface
SSLD=$DBGSDMP	Creates dumps using the SDUMP service for applications that use the SSL/TLS application programming interface
SSLD=$DBGSSLD	Create dumps of the parameters passed to the SSL/TLS application programming interface. These are the standard parm list values pointed to by R1 when requesting an SSL/TLS service.

Note: By default, the SDUMPs go to the defined dump lib.subib in the partition in which the associated application is running. Also, the SDUMP header that identifies the dump contents is lost. Therefore, always send the SDUMPs to SYSLST. To force SDUMPs directly to SYSLST and preserve the headers, add this JCL statement:

```
// OPTION NOSYSDMP
```

This statement is required if you plan to send SDUMPs to CSI Technical Support because the SDUMP headers are critically important to identifying the dumped contents.

Example 1: BSD Application Output

You have a test BSD application. You want to see the flow from all the functions used, but you do not want to send messages to the VSE system console. Messages are to go to SYSLST only.

Use the following procedure.

1. Copy and paste the sample job into your local editor.
2. Modify the following keyword settings:

Keyword	Use this setting	Result
FL01	FL01=$DBGWLST	Sends messages to SYSLST only
MSGT	MSGT=$DBGALL	Enables all messages to be issued

3. Modify the CATALOG parameter to contain a test lib.sublib in the phase search chain of the partition in which the application executes. The test lib.sublib must be located in the search chain *before* the TCP/IP FOR VSE installation's lib.sublib and PRD1.BASE, which may contain the IBM-distributed version of $SOCKDBG.
4. Run the modified job to create the custom $SOCKDBG.PHASE.
5. Cycle the partition in which the application, such as CICS, is running to force a reload of the newly created $SOCKDBG.PHASE.

Example 2: SSL/TLS Application Output

You have a vendor-supplied SSL/TLS application, and CSI technical support has requested that you send diagnostic dumps and messages to SYSLST.

Use the following procedure.

1. Copy and paste the sample job into your local editor.
2. Modify the following keyword settings:

Keyword	Use this setting	Result
FL01	FL01=$DBGWLST	Sends messages to SYSLST only
MSGT	MSGT=$DBGALL	Enables all messages to be issued
SSLD	SSLD=$DBGSDMP	Enables SSL diagnostic dumps

3. Modify the CATALOG parameter to contain a test lib.sublib in the phase search chain of the partition in which the application executes. The test lib.sublib must be located in the search chain *before* the TCP/IP FOR VSE installation lib.sublib and PRD1.BASE, which may contain the IBM-distributed version of $SOCKDBG.
4. Run the modified job to create the custom $SOCKDBG.PHASE.

5. Cycle the partition in which the application is running to force a reload of the newly created $SOCKDBG.PHASE.

Additional notes:

- The SDUMPX macro is used to create the diagnostic dumps.
- The application's JCL must include a // OPTION NOSYSDMP entry to force a dump to SYSLST instead of the default dump library.

Controlling $SOCKDBG Messages

You can use the IBM z/VSE EZAAPI console command to turn $SOCKDBG messages on and off dynamically. For CICS applications, this command allows you to enable and disable debugging messages without cycling CICS. Note that you cannot use this command to control dumps in the debug phase.

The EZAAPI trace facility generates one or more trace messages for each (IBM) EZASMI or EZASOKET socket call. It allows you to trace these calls either for all partitions in the system or for selected partitions and/or dynamic classes. You can direct trace messages to SYSLOG or SYSLST.

The EZAAPI command has the following arguments:

Argument	Description
?	Display the command syntax
TRACE	Display current trace settings
TRACE=ON	Define and start or resume starting: • (default with no trace defined yet): Define and start a trace with the defaults ALL and SYSLST • (default after EZAAPI TRACE=OFF): Resume trace • All: Define and start a trace for all partitions • PART=(part,..): Define and start a trace for selected partitions • CLASS=(class,..): Define and start a trace for selected dynamic classes
TRACE=OFF	Suspend the current trace
TRACE=END	End tracing, and clear all trace definitions
SYSLST	Send trace output to SYSLST (if SYSLST is assigned)
SYSLOG	Send trace output to SYSLOG
LOGLST	Send trace output to SYSLOG and SYSLST

See the appropriate IBM z/VSE documentation for more information.

Appendix C: TLS 1.2 Enhancement

Overview

TLS 1.2 was a major revision in the evolution of the Secure Socket Layer (SSL) and Transport Layer Security (TLS) protocols. It is documented in IETF RFC5246. Because the changes in TLS 1.2 are significant, support for TLS 1.2 is in new, separate modules. This is to provide maximum stability for applications currently using SSL 3.0, TLS 1.0, and TLS 1.1. Applications are not required to make any changes and will function without any changes.

Support for TLS 1.2 can be obtained by setting the INI@PROT of the call to IPCRINIT to PROT0303, which is the internal TLS release value for TLS 1.2. This will cause the current calls to invoke the modules for TLS 1.2.

Using TLS 1.2

SSL/TLS VSE applications that want to support only TLS 1.2 can

- Include a new stub object deck, IPCRTLSS.OBJ, to obtain support for only TLS 1.2, and
- Set an option in $SOCKOPT to only allow the usage of TLS 1.2. See "New $SOCKOPT Settings for TLS 1.2" on page 201 for details.

Supporting only TLS 1.2 may be desirable when you want to create more secure applications by not allowing the older, less secure versions of the protocol to be used.

The implementation of TLS 1.2 also requires that external clients and servers provide support for it. But many applications, like older versions of web browsers, FTP clients, FTP servers, and TN3270 clients, may not provide support for TLS 1.2 yet, and it is outside the control of the z/VSE clients and servers that require this new, higher level of security. When a business requires using TLS 1.2, outside vendors must be contacted to provide support for TLS 1.2.

Entry Points

By default, the older entry points (IPCR*) will automatically pass control to the new TLS 1.2 modules, but it can be more efficient to directly call the new TLS 1.2 entry points that are resolved in the new IPCRTLSS.OBJ module. The new TLSVSE.A macro should also be used to generate the support for applications that want to support TLS 1.2. See "TLSVSE Macro Settings," page 211, for macro details.

New entry points should be used to invoke the usage of TLS 1.2. These entry points are listed in the table below. They use the same input parameters as documented for each function in "Secure Socket Layer (SSL) and Transport Layer Security (TLS) API," page 125.

	Entry Point Name	
C Function	**New**	**Old**
gsk_initialize()	TLSRINIT	IPCRINIT
gsk_secure_soc_init()	TLSRSINI	IPCRSINI
gsk_secure_soc_read()	TLSRSRED	IPCRSRED
gsk_secure_soc_write()	TLSRSWRT	IPCRSWRT
gsk_secure_soc_close()	TLSRSCLS	IPCRSCLS
gsk_secure_soc_reset()	TLSRSRSM	IPCRSRST[1]
gsk_user_set()	TLSRUSET	IPCRUSET
gsk_free_memory()	TLSRFMEM	IPCRFMEM
gsk_get_cipher_info()	TLSRGCIN	IPCRGCIN
gsk_get_dn_by_label()	TLSRGDBL	IPCRGDBL
gsk_uninitialize()	TLSRUNIN	IPCRUNIN

[1] Resumes a previously negotiated session.

Support for Older Protocol Versions

Applications calling the new entry points will also still be able to provide support for the older versions (SSL 3.0, TLS 1.0, and TLS 1.1) since the code will automatically pass control to the older IPCRYPTO.PHASE during the SSL/TLS session negotiation. Applications using the new entry points above should also remove the IPCRYPTS.OBJ from their link edit.

The important consideration is to guarantee the integrity and security of connections, in that the older versions of the protocol can be susceptible to various forms of attack such as rollbacks and other weaknesses. For more detailed information on the security of the older protocol versions, see https://en.wikipedia.org/wiki/Transport_Layer_Security#Security . Google searches and other websites can also be referenced for more information on the security exposures in the older versions of SSL and TLS.

The question is not so much “Is the connection secure or not?”, but more of “How secure is it?” Some of the older versions and encryption algorithms such as DES are no longer considered strongly secured, but also keep in mind that currently many clients and servers have not been upgraded to support the new, more secure algorithms such as AES and TLS 1.2. But this should change over time.

Cipher Suite Values

Introduction

The SSL and TLS protocols allow a client to propose a list of cipher suites to be used during session negotiation, and the server then selects the cipher suite that will be used based on the client’s proposed list. If no cipher suite is acceptable to the server, the session will NOT be established. Most applications want to use the strongest available cipher suites, but this will in general use more overhead (CPU processing time).

In addition, older cipher suites can become deprecated due to known weaknesses being exploited. TLS 1.2 recommends no longer using the DES algorithm, and TCP/IP FOR VSE does not support it when using TLS 1.2. The older cipher suites for SSL 3.0, TLS 1.0, and TLS 1.1, as documented in the *Optional Features Guide*, can still be used when using the older versions of the protocol.

When TLS 1.2 is used, cipher suites supporting the DES algorithm have been removed. TLS 1.2 support also requires the IBM hardware CP Assist for Cryptographic Function (CPACF) feature for the SHA160, SHA-256, AES128, and AES256 algorithms.

Cipher Suites for TLS 1.2

TCP/IP FOR VSE currently supports these cipher suites for TLS 1.2:

- All cipher suites

  x2F = RSA_AES128CBC_SHA160

  x35 = RSA_AES256CBC_SHA160

  x3C = RSA_AES128CBC_SHA256

  x3D = RSA_AES256CBC_SHA256

- Strong cipher suites

  x3D = RSA_AES256CBC_SHA256

- Medium cipher suites

  x35 = RSA_AES256CBC_SHA160

  x3C = RSA_AES128CBC_SHA256

- Weak cipher suites

  x2F = RSA_AES128CBC_SHA160

Note that when a client proposes a list of cipher suites, the suites are in the client’s preferred order.

Suite Selection Example

The following cases assume that a server is set to use the “All cipher suites” above—2F, 35, 3C, 3D.

Case 1. If the client proposes (0A 3C 2F 3D), the server will choose the first match, which is 3C.

Case 2. If the client proposes (0A 2F 3C 3D), the server will choose the first match, which is 2F.

Note also that 0A is an older DES-based cipher suite that is not supported in TLS 1.2.

Pseudo Random Facility (PRF)

The TLS 1.2 protocol specifications are defined in IETF RFC5246, which has replaced the MD5/SHA-1 PRF with cipher-suite-specified PRFs.

All cipher suites in this document use P_SHA256 as defined in RFC5246, which can and should be referenced for details on this central, critical function of the protocol.

TLSVSE Macro Settings

The operands and values for the new TLSVSE macro are listed below.

- TLSVSE   GENVTLS – will generate:

```
o  IPCRTLSS DC       CL8'IPCRTLSS'
o           DC       V(IPCRTLSS)
```

Usage Note: This should be used for applications wanting to support only TLS 1.2.

- TLSVSE   GENVWTLS – will generate:

```
o  IPCRTLSS DC       CL8'IPCRTLSS'
o           DC       V(IPCRTLSS)
o           WXTRN    IPCRYPTS                 Older stub
```

Usage Note: The above adds a weak external reference (WXTRN) for the IPCRYPTS.OBJ. This can then be used to optionally allow applications to support both TLS 1.2 and the older versions of the protocol.

- TLSVSE   GENTLS12 – will generate:

```
o  SUITETAB DS    0F
o  SUIT002F DC    CL16'RSA_AES128_SH160',XL2'002F',CL2'2F'
o  SUIT0035 DC    CL16'RSA_AES256_SH160',XL2'0035',CL2'35'
o  SUIT003C DC    CL16'RSA_AES128_SH256',XL2'003C',CL2'3C'
o  SUIT003D DC    CL16'RSA_AES256_SH256',XL2'003D',CL2'3D'
o  SUITENTL EQU   20                            Length of a single tab entry
o  SUITECNT EQU   (*-SUITETAB)/SUITENTL       Number of entries in table
o  *
o  PROTTABL DS    0F
o  PROT0303 DC    XL2'0303',CL5'TLS12',XL1'00' TLS 1.2 defined in RFC5246
o  PROTENTL EQU   8                             Length of a single tab entry
o  PROTECNT EQU   (*-PROTTABL)/PROTENTL       Number of entries in table
```

Usage Note: The above are the cipher suites and protocol version for supporting only TLS 1.2.

- TLSVSE   GENVCON – will generate:

  ```
  o  IPCRYPTS DC       CL8'IPCRYPTS'
  o           DC       V(IPCRYPTS)
  ```

  Usage Note: The above can be used for applications requiring support for the older versions of the protocol (SSL 3.0, TLS 1.0, TLS 1.1).

- TLSVSE   GENSUIT – will generate:

  ```
  o  SUITETAB DS    0F
  o  SUIT0000 DC    CL16'NULL_NULL_NULL  ',XL2'0000',CL2'00'
  o  SUIT0001 DC    CL16'RSA_NULL_MD5    ',XL2'0001',CL2'01'
  o  SUIT0002 DC    CL16'RSA_NULL_SHA    ',XL2'0002',CL2'02'
  o  SUIT0008 DC    CL16'RSA_SDES040_SHA ',XL2'0008',CL2'08'
  o  SUIT0009 DC    CL16'RSA_SDES056_SHA ',XL2'0009',CL2'09'
  o  SUIT000A DC    CL16'RSA_TDES168_SHA ',XL2'000A',CL2'0A'
  o  SUIT002F DC    CL16'RSA_AES128_SHA  ',XL2'002F',CL2'2F'
  o  SUIT0035 DC    CL16'RSA_AES256_SHA  ',XL2'0035',CL2'35'
  o  SUITENTL EQU   20                     Length of a single tab entry
  o  SUITECNT EQU   (*-SUITETAB)/SUITENTL Number of entries in table
  ```

  Usage Note: The above can be used for applications requiring support for the older cipher suites (SSL 3.0, TLS 1.0, and TLS 1.1).

- TLSVSE   GENPROT – will generate:

  ```
  o  PROTTABL DS    0F
  o  PROT0300 DC  XL2'0300',CL5'SSL30',XL1'00'  SSL 3.0 created by Netscape
  o  PROT0301 DC  XL2'0301',CL5'TLS10',XL1'00'  TLS 1.0 defined in RFC2246
  o  PROT0302 DC  XL2'0302',CL5'TLS11',XL1'00'  TLS 1.1 defined in RFC4346
  o  PROT0303 DC  XL2'0303',CL5'TLS12',XL1'00'  TLS 1.2 defined in RFC5246
  o  PROTENTL EQU   8                             Length of a single tab entry
  o  PROTECNT EQU   (*-PROTTABL)/PROTENTL       Number of entries in table
  ```

  Usage Note: The above can be used for applications requiring support for the older protocol versions (SSL 3.0, TLS 1.0, and TLS 1.1) and TLS 1.2.

- TLSVSE   GENCNST – will generate the combination of GENVTLS, GENVCON, GENSUIT, and GENPROT in a single statement.

  Usage Note: This can be used for applications wanting to support all the possible protocol versions and cipher suites.

- TLSVSE   INLINE – will generate the assembler request areas in line.

- TLSVSE   DSECT – will generate the assembler request areas in dsects.