Object REXX for Linux

REXX TCP/IP Socket Library Functions (RxSock)

Version 1.2

Note!
Before using this information and the product it supports, be sure to read the general information under "Appendix. Notices" on page 33.

Second Edition, March 1999

This edition applies to Version 1.2 of IBM® Object REXX for Linux, and to all subsequent releases and modifications until otherwise indicated in new editions or technical newsletters.

Contents

Chapter 1. What is RxSock?

RxSock is a REXX function package providing access to the TCP/IP socket APIs available to the C programming environment. Most of the functions described in this reference are similar to the corresponding C functions available in the TCP/IP socket library.

It is assumed that you are familiar with the basic socket APIs and can reference those specific to the system. For more information, refer to the book *Internetworking with TCP/IP, Volume I: Principles, Protocols and Architecture* by Douglas Comer (Prentice Hall PTR).

The RxSock package requires TCP/IP support to be active on your system.

Chapter 2. Installation and Removal

The RxSock package is contained in the file **rxsock.dll**. This file must be placed in a directory listed in your LIBPATH. To get access to the functions in the RxSock package, execute the following REXX code:

```
If RxFuncQuery('SockDropFuncs') then
do
   rc = RxFuncAdd("SockLoadFuncs","rxsock","SockLoadFuncs")
   rc = SockLoadFuncs()
end
```

To unload the DLL, call the SockDropFuncs() function and then exit all CMD.EXE shells. After exiting all command shells, the DLL is dropped by the system and can be deleted or replaced.

Chapter 3. Parameters and Return Values

Unless otherwise stated, the return values are the same as for the corresponding C functions. The following standard parameter types are referred to throughout this reference:

socket
is a socket value, which is an integral number.

domain
is a domain value. Currently, only the domain ″AF_INET″ is supported.

address
is the stem of a stem variable with the following values:

address.family
must always be ″AF_INET″.

address.port
is a port number.

address.addr
is a dotted decimal address or ″INADDR_ANY″, where appropriate.

When this parameter is needed, set it the name of a stem variable for the function to set (or that the function will read from). For example, if you pass the string ″xxx.!″ as a parameter, the following variables are set or queried by the function:

```
"xxx.!family"
"xxx.!port"
"xxx.!addr"
```

A null address is an address with the family field being ″AF_INET″, the port field being 0, and the addr field being ″0.0.0.0″.

dotAddress
is the standard dotted decimal address. For example, the string ″9.23.19.63″ is a valid address.

host
is the stem of a stem variable with the following values:

host.name
is the standard name of the host.

host.alias.0
is the number of aliases for this host.

host.alias.1
is the first alias for this host.

host.alias.n
is the nth alias for this host.

host.addrtype
must always be ″AF_INET″.

host.addr
is a dotted decimal address (default address).

host.addr.0
is the number of addresses for this host.

host.addr.1
is the first address for this host.

host.addr.n
is the nth address for this host.

When this parameter is needed, set it the name of a stem variable for the function to set (or that the function will read from). For example, if you pass the string ″xxx.!″ as a parameter, the following variables are set or queried by the function:

```
"xxx.!name"
"xxx.!alias.0", "xxx.!alias.1" ... "xxx.!alias.n"
"xxx.!addrtype"
"xxx.!addr"
"xxx.!addr.0", "xxx.!addr.1" ... "xxx.!addr.n"
```

Stem Variables

The address and host type of a parameter are stems of a stem variable. Normally, when you pass a string like ″addr.″ as a parameter, you expect the variables addr.family, addr.port, and addr.addr to be set by the function. In the previous examples, however, the stem contained an exclamation mark. This exclamation mark helps prevent the value that follows from getting misused as a normal variable. Example:

```
port = 923
sNew =  SockAccept(sOld,"addr.")
say addr.port
```

In this example, you might expect the say statement to write the port number of the accepted socket. Instead, it writes the value of the variable, namely addr.923, because the port variable is set to this value.

Because exclamation marks are rarely used in variables, it is unlikely that the variable ″!port″ is used in your program.

Note: Do not use the characters _, 0, and 1 to prefix tail values. 0 and 1 are difficult to distinguish from O, I, and l.

Chapter 4. Special Variables

The following variables are maintained by the system: *errno* and *h_errno*.

Variable errno

The variable *errno* is set after each RxSock call. It can have one of the following values or any other numeric value:

- "EWOULDBLOCK"
- "EINPROGRESS"
- "EALREADY"
- "ENOTSOCK"
- "EDESTADDRREQ"
- "EMSGSIZE"
- "EPROTOTYPE"
- "ENOPROTOOPT"
- "EPROTONOSUPPORT"
- "ESOCKTNOSUPPORT"
- "EOPNOTSUPP"
- "EPFNOSUPPORT"
- "EAFNOSUPPORT"
- "EADDRINUSE"
- "EADDRNOTAVAIL"
- "ENETDOWN"
- "ENETUNREACH"
- "ENETRESET"
- "ECONNABORTED"
- "ECONNRESET"
- "ENOBUFS"
- "EISCONN"
- "ENOTCONN"
- "ESHUTDOWN"
- "ETOOMANYREFS"
- "ETIMEDOUT"
- "ECONNREFUSED"
- "ELOOP"
- "ENAMETOOLONG"
- "EHOSTDOWN"
- "EHOSTUNREACH"
- "ENOTEMPTY"

Note: The value is set even if the function called does not set the variable, in which case the value has no meaning. A value of 0 indicates that no error occurred.

Variable h_errno

The variable *h_errno* is set after each RxSock call. It can have one of the following values or any other numeric value:

- HOST_NOT_FOUND
- TRY_AGAIN
- NO_RECOVERY
- NO_ADDRESS

Note: The value is set even if the function called does not set the variable, in which case the value has no meaning. A value of 0 indicates that no error occurred.

Chapter 5. Function Reference

The following sections describe how the individual functions contained in RxSock are invoked from the REXX programming environment:

- SockLoadFuncs
- SockDropFuncs
- SockVersion
- SockAccept
- SockBind
- SockClose
- SockConnect
- SockGetHostByAddr
- SockGetHostByName
- SockGetHostId
- SockGetPeerName
- SockGetSockName
- SockGetSockOpt
- SockInit
- SockIoctl
- SockListen
- SockPSock_Errno
- SockRecv
- SockRecvFrom
- SockSelect
- SockSend
- SockSendTo
- SockSetSockOpt
- SockShutDown
- SockSock_Errno
- SockSocket
- SockSoClose

SockLoadFuncs

The SockLoadFuncs() call loads all RxSock functions.

Syntax:

```
SockLoadFuncs([parm])
```

All parameters that you supply are only used to bypass copyright information.

SockDropFuncs

The SockDropFuncs call drops all RxSock functions.

Syntax:

```
SockDropFuncs()
```

To unload the dynamic load library (DLL), first call SockDropFuncs() and then exit all CMD.EXE shells. After exiting all command shells, the DLL is dropped by the system and can be deleted or replaced.

SockVersion

The SockVersion() call provides the version of RxSock.

Syntax:

```
vers = SockVersion()
```

Return Values:

The returned value is in the form *version.subversion*, for example 2.1.

Prior to Version 1.2, this function did not exist. To check if a former version of Rxsock is installed, use the following code after loading the function package with SockLoadFuncs():

```
/* oldVersion is 1 if a version of RxSock < 1.2 is loaded */
oldVersion = (1 = RxFuncQuery("SockVersion"))
```

SockAccept

The SockAccept() call accepts a connection request from a remote host.

Syntax:

```
csocket = SockAccept(socket[, address])
```

where:

socket
: is the socket descriptor created with the SockSocket() call. It is bound to an address using the SockBind() call and must be enabled to accept connections using theSockListen() call.

address
: is a stem variable that contains the socket address of the connection client when the SockAccept() call returns. This parameter is optional.

SockAccept() is used by a server in a connection-oriented mode to accept a connection request from a client. The call accepts the first connection on its queue of pending connection requests. It creates a new socket descriptor with the same properties as *socket* and returns it to the caller. This new socket descriptor cannot be used to accept new connections. Only the original *socket* can accept more connection requests.

If the queue has no pending connection requests, SockAccept() blocks the caller unless the socket is in nonblocking mode. If no connection requests are queued

and the socket is in nonblocking mode, SockAccept() returns a value of -1 and sets the return code to the value EWOULDBLOCK.

You cannot get information on requesters without calling SockAccept(). The application cannot tell the system from which requesters it will accept connections. The caller can close a connection immediately after identifying the requester.

The SockSelect() call can be used to check the socket for incoming connection requests.

Return values:

A positive value indicates successful execution of the call. The value -1 indicates an error. You can get the specific error code by calling SockSock_Errno() or SockPSock_Errno(). Possible values:

ENOTSOCK
socket is not a valid socket descriptor.

EINTR
Interrupted system call.

EINVAL
SockListen() was not called for *socket*.

EOPNOTSUPP
socket is not connection-oriented.

EWOULDBLOCK
socket is in nonblocking mode and there are no connection requests queued.

ECONNABORTED
The software caused a connection close.

Note: SockAccept() interfaces with the C function accept().

SockBind

The SockBind() call binds a local name to the socket.

Syntax:

```
rc = SockBind(socket, address)
```

where:

socket
is the socket descriptor returned by a previous call to SockSocket().

address
is a stem variable containing the address that is to be bound to *socket*.

SockBind() binds the unique local name *address* to the socket with descriptor *socket*. After calling SockSocket(), a descriptor does not have a name. However, it belongs to a particular address family that you specified when calling SockSocket().

Because *socket* was created in the "AF_INET" domain, the fields of the stem *address* are as follows:

The *family* field must be set to "AF_INET". The *port* field is set to the port to which the application must bind. If *port* is set to 0, the caller allows the system to assign

an available port. The application can call SockGetSockName() to discover the port number assigned. The *addr* field is set to the Internet address. On hosts with more than one network interface (called multihomed hosts), a caller can select the interface with which it is to bind.

Only UDP packets and TCP connection requests from this interface that match the bound name are routed to the application. This is important when a server offers a service to several networks. If *addr* is set to "INADDR_ANY", the caller requests *socket* be bound to all network interfaces on the host. If you do not specify an address, the server can accept all UDP packets and TCP connection requests made to its port, regardless of the network interface on which the requests arrived.

Return values:

The value 0 indicates successful execution of the call. The value -1 indicates an error. You can get the specific error code by calling SockSock_Errno() or SockPSock_Errno(). Possible values:

EADDRINUSE
address is already in use. See the SO_REUSEADDR option described under SockGetSockOpt() and the SO_REUSEADDR option described under SockSetSockOpt().

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

EAFNOSUPPORT
The address family is not supported.

ENOTSOCK
socket is not a valid socket descriptor.

EINVAL
socket is already bound to an address.

ENOBUFS
No buffer space available.

Note: SockBind() interfaces with the C function bind().

SockClose

The SockClose() call shuts down a socket and frees resources allocated to the socket.

Syntax

```
rc = SockClose(socket)
```

where:

socket
is the descriptor of the socket to be closed.

If the SO_LINGER option of SockSetSockOpt() is enabled, any queued data is sent. If this option is disabled, any queued data is flushed.

Return values:

The value 0 indicates successful execution of the call. The value -1 indicates an error. You can get the specific error code by calling SockSock_Errno() or SockPSock_Errno(). Possible values are:

ENOTSOCK
socket is not a valid socket descriptor.

EALREADY
The socket *socket* is in nonblocking mode. A previous connection attempt has not completed.

SockClose() is exactly the same as SockSoClose().

Note: SockClose() interfaces with the C function soclose() or, in the Windows environments, with closesocket().

SockConnect

The SockConnect() socket call requests a connection to a remote host.

Syntax:

```
rc = SockConnect(socket, address)
```

where:

socket
is the socket descriptor used to issue the connection request.

address
is a stem variable containing the address of the socket to which a connection is to be established.

The SockConnect() call performs the following tasks when called for a **stream socket**:

1. It completes the binding for a socket, if necessary.
2. It attempts to create a connection between two sockets.

This call is used by the client side of socket-based applications to establish a connection with a server. The remote server must have a passive open pending, which means it must successfully call SockBind() and SockListen(). Otherwise, SockConnect() returns the value -1 and the error value is set to ECONNREFUSED.

In the Internet communication domain, a timeout occurs if a connection to the remote host is not established within 75 seconds.

If the socket is in blocking mode, the SockConnect() call blocks the caller until the connection is established or an error is received. If the socket is in nonblocking mode, SockConnect() returns the value -1 and sets the error value to EINPROGRESS if the connection was successfully initiated. The caller can test the completion of the connection by calling:

- SockSelect(), to test for the ability to write to the socket
- SockGetsockOpt(), with option SO_ERROR, to test if the connection was established

Stream sockets can call SockConnect() only once.

Datagram or raw sockets normally transfer data without being connected to the sender or receiver. However, an application can connect to such a socket by calling SockConnect(). SockConnect() specifies and stores the destination peer address for the socket. The system then knows to which address to send data and the destination peer address does not have to be specified for each datagram sent. The address is kept until the next SockConnect() call. This permits the use of the SockRecv() and SockSend() calls, which are usually reserved for connection-oriented sockets. However, data is still not necessarily delivered, which means the normal features of sockets using connectionless data transfer are maintained. The application can therefore still use the SockSendTo()and SockRecvFrom() calls.

Datagram and raw sockets can call SockConnect() several times. The application can change their destination address by specifying a new address on the SockConnect() call. In addition, the socket can be returned to a connectionless mode by calling SockConnect() with a null destination address. The null address is created by setting the stem variable *address* as follows: the *family* field to "AF_INET", the *port* field to 0, and the *addr* field to "0.0.0.0".

The call to SockConnect returns the value -1, indicating that the connection to the null address cannot be established. Calling SockSock_Errno() returns the value EADDRNOTAVAIL.

Return values:

The value 0 indicates successful execution of the call. The value -1 indicates an error. You can get the specific error code by calling SockSock_Errno() or SockPSock_Errno(). Possible values are:

EADDRNOTAVAIL
The calling host cannot reach the specified destination.

EAFNOSUPPORT
The address family is not supported.

EALREADY
The socket *socket* is in nonblocking mode. A previous connection attempt has not completed.

ENOTSOCK
The socket *socket* is not a valid socket descriptor.

ECONNREFUSED
The destination host rejected the connection request.

EINPROGRESS
socket is in nonblocking mode, and the connection cannot be completed immediately. EINPROGRESS does not indicate an error.

EINTR
Interrupted system call.

EISCONN
socket is already connected.

ENETUNREACH
The network cannot be reached from this host.

ETIMEDOUT
Establishing the connection timed out.

ENOBUFS
There is no buffer space available.

EOPNOTSUPP
The operation is not supported on *socket*.

Note: SockConnect interfaces with the C function connect().

SockGetHostByAddr

The SockGetHostByAddr() call retrieves information about a specific host using its address.

Syntax:

```
rc = SockGetHostByAddr(dotAddress, host[, domain])
```

where:

dotAddress
is the standard dotted decimal address of the host.

host
is a stem variable that is to receive the information on the host.

domain
is the domain ″AF_INET″. This parameter is optional.

Return values:

The value 1 indicates successful execution of the call. The value 0 indicates an error.

Note: SockGetHostByAdress() interfaces with the C function gethostbyaddr().

SockGetHostByName

The SockGetHostByName() call retrieves host information on a specific host using its name or any alias.

Syntax:

```
rc = SockGetHostByName(nameAddress, host)
```

where:

nameAddress
is the name of a host, for example www.ibm.com.

host
is the name of a stem variable to receive the information on the host.

Return values:

The value 1 indicates successful execution of the call. The value 0 indicates an error.

Note: SockGetHostByName() interfaces with the C function gethostbyname().

SockGetHostId

The SockGetHostId() call retrieves the dotAddress of the local host.

Syntax:

```
dotAddress = SockGetHostId()
```

The return value is the dotAddress of the local host.

Note: SockGetHostId() interfaces with the C function gethostid().

SockGetPeerName

The SockGetPeerName() call gets the name of the peer connected to a socket.

Syntax:

```
rc = SockGetPeerName(socket, address)
```

where:

socket
is the socket descriptor.

address
is a stem variable containing the address of the peer connected to *socket*.

Return values:

The value 0 indicates successful execution of the call. The value -1 indicates an error. You can get the specific error code by calling SockSock_Errno() or SockPSock_Errno(). Possible values are:

ENOTSOCK
socket is not a valid socket descriptor.

ENOTCONN
socket is not connected.

ENOBUFS
There is no buffer space available.

Note: SockGetPeerName() interfaces with the C function getpeername().

SockGetSockName

The SockGetSockName() call gets the local socket name.

Syntax:

```
rc = SockGetSockName(socket, address)
```

where:

socket
is the socket descriptor.

address
is a stem variable that is to receive the address of the socket returned.

SockGetSockName() returns the address for socket *socket* in the stem variable address. If the socket is not bound to an address, the call returns a null address.

The returned null address is a stem variable with the *family* field set to "AF_INET", the *port* field set to 0, and the *addr* field set to "0.0.0.0".

All sockets are explicitly assigned an address after a successful call to SockBind(). Stream sockets are implicitly assigned an address after a successful call to SockConnect() or SockAccept() if SockBind() was not called.

The SockGetSockName() call is often used to identify the port assigned to a socket after the socket has been implicitly bound to a port. For example, an application can call SockConnect() without previously calling SockBind(). In this case, the SockConnect() call completes the binding necessary by assigning a port to the socket.

Return values:

The value 0 indicates successful execution of the call. The value -1 indicates an error. You can get the specific error code by calling SockSock_Errno() or SockPSock_Errno(). Possible values are:

ENOTSOCK
socket is not a valid socket descriptor.

ENOBUFS
There is no buffer space available.

Note: SockGetSockName() interfaces with the C function getsockname().

SockGetSockOpt

The SockGetSockOpt() call gets the socket options associated with a socket.

Syntax:

```
rc = SockGetSockOpt(socket, level, optName, optVal)
```

where:

socket
is the socket descriptor.

level
specifies which option level is queried for the specified *optname*. The only supported level is SOL_SOCKET.

optname
is the name of the specified socket option. Only one option can be specified with a call.

optval
is the variable to receive the option values requested. For socket options that are Boolean the option is enabled if *optval* is nonzero and disabled if *optval* is 0.

SockGetSockOpt() returns the value of a socket option at the socket level. It can be requested for sockets of all domain types. Some options are supported only for specific socket types.

The following options are recognized for SOL_SOCKET:

SO_BROADCAST
returns the information whether datagram sockets are able to broadcast messages. If this option is enabled, the application can send broadcast messages using datagram socket *socket*, if the interface specified in the destination supports broadcasting of packets.

SO_DEBUG
returns the information whether debug information can be recorded for a socket.

SO_DONTROUTE
returns the information whether the socket is able to bypass the routing of outgoing messages. If this option is enabled, outgoing messages are directed to the network interface specified in the network portion of the destination address. When enabled, packets can only be sent to directly connected networks.

SO_ERROR
returns any error pending at the socket and clears the error status. It can be used to check for asynchronous errors at connected datagram sockets or for asynchronous errors that are not explicitly returned by one of the socket calls.

SO_KEEPALIVE
returns the information whether stream sockets are able to send keepalive packets. TCP uses a timer called the keepalive timer. This timer monitors idle connections that might have been disconnected because of a peer crash or timeout. If this option is enabled, a keepalive packet is periodically sent to the peer.

This option is mainly used to enable servers to close connections that are no longer active as a result of clients ending connections without properly closing them.

SO_LINGER
returns the information whether stream sockets are able to linger on close if data is present. If this option is enabled and there is data still to be sent when SockSoClose() is called, the calling application is blocked during the SockSoClose() call until the data is transmitted or the connection has timed out. If this option is disabled, the SockSoClose() call returns without blocking the caller while TCP is trying to send the data. Although the data transfer is usually successful, it cannot be guaranteed because TCP tries to send the data only for a specific amount of time.

SO_OOBINLINE
returns the information whether stream sockets are able to receive out-of-band data. If this option is enabled, out-of-band data is placed in the normal data input queue as it is received. It is then made available to SockRecv() and SockRecvFrom() without the MSG_OOB flag being specified in those calls. If this option is disabled, out-of-band data is placed in the priority data input queue as it is received. It can then only be made available to SockRecv() and SockRecvFrom() by specifying the MSG_OOB flag in those calls.

SO_RCVBUF
returns the buffer size for input.

SO_RCVLOWAT
returns the receive low-water mark.

SO_RCVTIMEO
returns the timeout value for a receive operation.

SO_REUSEADDR
returns the information whether stream and datagram sockets are able to reuse local addresses. If this option is enabled, the local addresses that are already in use can then be bound. This alters the normal algorithm used in the SockBind() call. At connection time, the system checks whether the local addresses and ports differ from foreign addresses and ports. If not, the error value EADDRINUSE is returned.

SO_SNDBUF
returns the size of the send buffer.

SO_SNDLOWAT
returns the send low-water mark. This mark is ignored for nonblocking calls and not used in the Internet domain.

SO_SNDTIMEO
returns the timeout value for a send operation.

SO_TYPE
returns the socket type. The integer pointed to by *optval* is then set to one of the following: ″STREAM″, ″DGRAM″, ″RAW″, or ″UNKNOWN″.

SO_USELOOPBACK
bypasses hardware where possible.

All option values are integral except for SO_LINGER, which contains the following blank-delimited integers:

- The l_onoff value. It is set to 0 if the SO_LINGER option is disabled.
- The l_linger value. It specifies the amount of time, in seconds, to be lingered on close. A value of 0 causes SockSoClose() to wait until disconnection completes.

Return values:

The value 0 indicates successful execution of the call. The value -1 indicates an error. You can get the specific error code by calling SockSock_Errno() or SockPSock_Errno(). Possible values are:

EADDRINUSE
The address is already in use.

ENOTSOCK
socket is not a valid socket descriptor.

ENOPROTOOPT
optname or *level* is not recognized.

Note: SockGetSockOpt() interfaces with the C function getsockopt().

SockInit

The SockInit() call initializes the socket data structures and checks whether the TCP/IP network is active.

Syntax:

```
rc = SockInit()
```

SockInit() can be called at the beginning of each program that uses SockSocket(). However, it is not obligatory because each RxSock function is automatically initialized. For this reason, explicit initialization is not available in all system environments.

Return values:

The value 0 indicates successful execution of the call. The value 1 indicates an error.

Note: SockInit() interfaces with the C function sock_init().

SockIoctl

The SockIoctl() call performs special operations on the socket.

Syntax:

```
rc = SockIoctl(socket, ioctlCmd, ioctlData)
```

where:

socket
is the socket descriptor.

ioctlCmd
is the ioctl command to be performed.

ioctlData
is a variable containing data associated with the particular command. Its format depends on the command requested. Valid commands are:

FIONBIO
sets or clears nonblocking input or output for a socket. This command is an integer. If the integer is 0, nonblocking input or output on the socket is cleared. If the integer is a number other than 0, input or output calls do not block until the call is completed.

FIONREAD
gets the number of immediately readable bytes for the socket. This command is an integer.

Return values:

The value 0 indicates successful execution of the call. The value -1 indicates an error. You can get the specific error code SockSock_Errno() or SockPSock_Errno(). Possible values are:

ENOTSOCK
socket is not a valid socket descriptor.

EINVAL
The request is not valid or not supported.

EOPNOTSUPP
The operation is not supported on the socket.

Note: SockIoctl() interfaces with the C function ioctl() or, in the Windows environments, with ioctlsocket().

SockListen

The SockListen() call completes the binding necessary for a socket to accept connections and creates a connection request queue for incoming requests.

Syntax:

```
rc = SockListen(socket, backlog)
```

where:

socket
is the socket descriptor.

backlog
controls the maximum queue length for pending connections.

SockListen() performs the following tasks:

1. It completes the binding necessary for socket *socket*, if SockBind() has not been called for the socket.
2. It creates a connection request queue with a length of *backlog* to queue incoming connection requests.

When the queue is full, additional connection requests are ignored.

SockListen() can only be called for connection-oriented sockets.

SockListen() is called after allocating a socket with SockSocket() and after binding a name to *socket* with SockBind(). It must be called before SockAccept().

SockListen() indicates when it is ready to accept client connection requests. It transforms an active socket to a passive socket. After it is called, *socket* cannot be used as an active socket to initiate connection requests.

If *backlog* is smaller than 0, SockListen() interprets the backlog to be 0. If it is greater than the maximum value defined by the network system, SockListen() interprets the backlog to be this maximum value.

Return values:

The value 0 indicates successful execution of the call. The value -1 indicates an error. You can get the specific error code SockSock_Errno() or SockPSock_Errno(). Possible values are:

ENOTSOCK
socket is not a valid socket descriptor.

EOPNOTSUPP
socket is not a socket descriptor that supports the SockListen() call.

Note: SockListen() interfaces with the C function listen().

SockPSock_Errno

The SockPSock_Errno() call writes a short error message to the standard error device. It describes the last error encountered during a call to a socket library function.

Syntax:

```
SockPSock_Errno([error_string])
```

where:

error_string
is the error string written to the standard error device describing the last error encountered. The string printed is followed by a colon, a space, and then the message. If it is omitted or empty, only the message is printed. The string is optional.

The error code is acquired by calling SockSock_Errno(). It is set when errors occur. Subsequent socket calls do not clear the error code.

Note: SockPSock_Errno() interfaces with the C function psock_errno().

SockRecv

The SockRecv() call receives data on a connected socket.

Syntax:

```
rc = SockRecv(socket, var, len[, flags])
```

where:

socket
is the socket descriptor.

var
is the name of a REXX variable to receive the data.

len
is the maximum amount of data to be read.

flags
is a blank-delimited list of options:

MSG_OOB
reads any out-of-band data on the socket.

MSG_PEEK
peeks at the data on the socket. The data is returned but not removed, so the subsequent receive operation sees the same data.

SockRecv()l receives data on a socket with descriptor *socket* and stores it in the REXX variable *var*. It applies only to connected sockets. For information on how to use SockRecv() with datagram and raw sockets, see Datagram or raw sockets.

SockRecv() returns the length of the incoming data. If a datagram is too long to fit the buffer, the excessive data is discarded. No data is discarded for stream sockets. If data is not available at *socket*, the SockRecv() call waits for a message and blocks the caller unless the socket is in nonblocking mode. See SockIoctl() for a description of how to set the nonblocking mode.

Return values:

If successful, the length of the data in bytes is returned. The value 0 indicates that the connection is closed. The value -1 indicates an error. You can get the specific error code SockSock_Errno() or SockPSock_Errno(). Possible values are:

ENOTSOCK
socket is not a valid socket descriptor.

EINTR
Interrupted system call.

EINVAL
Invalid argument.

EWOULDBLOCK
socket is in nonblocking mode and no data is available, or the SO_RCVTIMEO option has been set for *socket* and the timeout expired before any data arrived.

Note: SockRecv() interfaces to the C function recv().

SockRecvFrom

The SockRecvFrom() call receives data on a socket.

Syntax:

```
rc = SockRecvFrom(socket, var, len[, flags], address)
```

where:

socket
is the socket descriptor.

var
is the name of a REXX variable to receive the data.

len is the maximum amount of data to be read.

flags
is a blank delimited list of options:

MSG_OOB
reads any out-of-band data on the socket.

MSG_PEEK
peeks at the data present on the socket. The data is returned but not consumed. The subsequent receive operation thus sees the same data.

address
is a stem variable specifying the address of the sender from which the data is received, unless it is a null address.

SockRecvFrom() receives data on a socket with descriptor *socket* and stores it in a REXX variable named *var*. It applies to any socket type, whether connected or not.

SockRecvFrom() returns the length of the incoming message or data. If a datagram is too long to fit the supplied buffer, the excessive data is discarded. No data is discarded for stream sockets. If data is not available at *socket*, the SockRecvFrom() call waits for a message to arrive and blocks the caller, unless the socket is in nonblocking mode. See SockIoctl() for a description of how to set the nonblocking mode.

Return values:

If successful, the length of the data in bytes is returned. The value -1 indicates an error. You can get the specific error code SockSock_Errno() or SockPSock_Errno(). Possible values are:

ENOTSOCK
socket is not a valid socket descriptor.

EINVAL
Invalid argument.

EWOULDBLOCK
socket is in nonblocking mode, no data is available, or the SO_RCVTIMEO option has been set for *socket* and the timeout expired before data arrived.

Note: SockRecvFrom() interfaces with the C function recvfrom().

SockSelect

The SockSelect() call monitors the activity on a socket with regard to readability, readiness for writing, and pending exceptional conditions.

Syntax:

```
rc = SockSelect(reads, writes, excepts[, timeout])
```

where:

reads
is the number of sockets to be checked for readability.

writes
is the number of sockets to be checked for readiness for writing.

excepts
is the number of sockets to be checked for pending exceptional conditions. For Network Services sockets, the only pending exceptional condition is out-of-band data in the receive buffer.

timeout
is the maximum number of seconds the system waits for the selection to complete. Set the timeout parameter to 0 for a blocking operation. If the socket is ready, the return will be immediate.

Each parameter specifying a number of sockets is qualified by a stem variable which is queried and set by this function. The stem variable has the following format: stem.0 contains the number of sockets, stem.1 the first socket, and so on. Upon return, the stem variables are reset to the sockets that are ready. If any of the stem variables are empty (""), or no parameter is passed, no sockets for that type are checked.

The timeout value must be integral (no fractional values). Nonnumeric and negative numbers are considered to be 0. If no timeout value is passed, an empty string ("") is assumed.

If the timeout value is 0, SockSelect() does not wait before returning. If the timeout value is an empty string (""), SockSelect() does not time out, but returns when a socket becomes ready. If the timeout value is in seconds, SockSelect() waits for the

specified interval before returning. It checks all indicated sockets at the same time and returns as soon as one of them is ready.

Return values:

The number of ready sockets is returned. The value 0 indicates an expired time limit. In this case, the stem variables are not modified. The value -1 indicates an error. You can get the specific error code SockSock_Errno() or SockPSock_Errno(). Possible values are:

ENOTSOCK
socket is not a valid socket descriptor.

EFAULT
The address is not valid.

EINVAL
Invalid argument.

EINTR
Interrupted system call.

Examples:

```
r.0 = 2                  /* specify 2 sockets for read in stem r. */
r.1 = 101
r.2 = 102
                         /* specify 1 socket for write in stem w. */
w.0 = 1
w.1 = 103
                         /* no sockets for exceptions in stem e.  */
e.0 = 0
rc = SockSelect("r.","w.","e.")

do i = 1 to r.0          /* display sockets ready for read        */
   say "socket" r.i "is ready for reading."
end
```

That SockSelect() call can be invoked as:

```
rc = SockSelect("r.","w.","")
```

or

```
rc = SockSelect("r.","w.",)
```

The function call SockSelect(, , , x) results in the program pausing for x seconds.

Note: SockSelect() interfaces with the C function select().

SockSend

The SockSend() call sends data to a connected socket.

Syntax:

```
rc = SockSend(socket, data[, flags])
```

where:

socket
is the socket descriptor.

data
is the name of a REXX variable containing the data to be transmitted.

flags
is a blank delimited list of options:

MSG_OOB
sends out-of-band data to sockets that support SOCK_STREAM communication.

MSG_DONTROUTE
turns on the SO_DONTROUTE option for the duration of the send operation. This option is usually only used by diagnostic or routing programs.

SockSend() sends data to a connected socket with descriptor *socket*. For information on how to use SockSend() with datagram and raw sockets, see Datagram or raw sockets.

If the socket does not have enough buffer space to hold the data to be sent, the SockSend() call blocks unless the socket is placed in nonblocking mode. See SockIoctl() for a description of how to set the nonblocking mode. Use the SockSelect() call to determine when it is possible to send more data.

Return values:

If successful, the number of bytes of the socket with descriptor *socket* that is added to the send buffer is returned. Successful completion does not imply that the data has already been delivered to the receiver.

The return value -1 indicates that an error was detected on the sending side of the connection. You can get the specific error code SockSock_Errno() or SockPSock_Errno(). Possible values are:

ENOTSOCK
socket is not a valid socket descriptor.

EINTR
Interrupted system call.

EINVAL
Invalid argument.

ENOBUFS
There is no buffer space available to send the message.

EWOULDBLOCK
socket is in nonblocking mode, the data cannot be sent without blocking, or the SO_SNDTIMEO option has been set for *socket* and the timeout expired before any data was sent.

Note: SockSend() interfaces with the C function send().

SockSendTo

The SockSentTo() call sends data to a connected or unconnected socket.

Syntax:

```
rc = SockSendTo(socket, data[, flags], address)
```

where:

socket
is the socket descriptor.

data
is a string of data to be transmitted.

flags
is a blank delimited list of options:

MSG_OOB
sends out-of-band data to sockets that support SOCK_STREAM communication.

MSG_DONTROUTE
turns on the SO_DONTROUTE option for the duration of the send operation. This option is usually only used by diagnostic or routing programs.

address
is a stem variable containing the destination address.

SockSendTo() sends data to a connected or unconnected socket with descriptor *socket*. For unconnected datagram and raw sockets, it sends data to the specified destination address. For stream sockets, the destination address is ignored.

Datagram sockets are connected by calling SockConnect(). This call identifies the peer to send or receive the datagram. After a datagram socket is connected to a peer, you can still use the SockSendTo() call but you cannot include a destination address.

To change the peer address when using connected datagram sockets, issue SockConnect() with a null address. Specifying a null address removes the peer address specification. You can then issue either a SockSendTo() call and specify a different destination address or a SockConnect() call to connect to a different peer. For more information on connecting datagram sockets and specifying null addresses, see Datagram or raw sockets.

Return values:

If successful, the number of bytes sent is returned. Successful completion does not guarantee that the data is delivered to the receiver. The return value -1 indicates that an error was detected on the sending side. You can get the specific error code SockSock_Errno() or SockPSock_Errno(). Possible values are:

ENOTSOCK
socket is not a valid socket descriptor.

EMSGSIZE
The message *data* was too big to be sent as a single datagram.

ENOBUFS
There is no buffer space available to send the message.

EWOULDBLOCK
socket is in nonblocking mode, the data cannot be sent without blocking, or the SO_SNDTIMEO option has been set for *socket* and the timeout expired before any data was sent.

ENOTCONN
The socket is not connected.

EDESTADDRREQ
Destination address required.

Note: SockSendTo() interfaces with the C function sendto().

SockSetSockOpt

The SockSetSockOpt() call sets options associated with a socket.

Syntax:

```
rc = SockSetSockOpt(socket, level, optName, optVal)
```

where:

socket
is the socket descriptor.

level
specifies which option level is set. The only supported level is SOL_SOCKET.

optname
is the name of a specified socket option.

optval
is the variable containing the data needed by the set command. It is optional.

SockSetSockOpt() sets options associated with a socket with descriptor *socket* such as enabling debugging at the socket or protocol level, controlling timeouts, or permitting socket data broadcasting. Options can exist at the socket or the protocol level. They are always present at the highest socket level. When setting socket options, the option level and name must be specified.

For socket options that are toggles, the option is enabled if *optval* is nonzero and disabled if *optval* is 0.

The following options are recognized for SOL_SOCKET:

SO_BROADCAST
enables datagram sockets to broadcast messages. The application can then send broadcast messages using datagram socket *socket*, if the interface specified in the destination supports broadcasting of packets.

SO_DEBUG
enables debug information to be recorded for a socket.

SO_DONTROUTE
enables the socket to bypass the routing of outgoing messages. Outgoing messages are then directed to the network interface specified in the network portion of the destination address. When enabled, packets can only be sent to directly connected networks.

SO_KEEPALIVE
enables stream sockets to send keepalive packets, which keep the connection alive. TCP uses a timer called the keepalive timer. This timer monitors idle connections that might have been disconnected because of a peer crash or timeout. If this option is enabled, a keepalive packet is periodically sent to the peer.

This option is mainly used to enable servers to close connections that are no longer active as a result of clients ending connections without properly closing them.

SO_LINGER
enables stream sockets to linger on close if data is present. If this option is enabled and there is data still to be sent when SockSoClose() is called, the calling application is blocked during the SockSoClose() call until the data is transmitted or the connection has timed out. If this option is disabled, the SockSoClose() call returns without blocking the caller while TCP is trying to send the data. Although the data transfer is usually successful, it cannot be guaranteed because TCP tries to send the data only for a specific amount of time.

SO_OOBINLINE
enables stream sockets to receive out-of-band data, which is a logically separate data path using the same connection as the normal data path. If this option is enabled, out-of-band data is placed in the normal data input queue as it is received. It is then made available to SockRecv() and SockRecvFrom() without the MSG_OOB flag being specified in those calls. If this option is disabled, out-of-band data is placed in the priority data input queue as it is received. It can then only be made available to SockRecv() and SockRecvFrom() by specifying the MSG_OOB flag in those calls.

SO_RCVBUF
sets the buffer size for input. This option sets the size of the receive buffer to the value contained in the buffer pointed to by *optval*. In this way, the buffer size can be tailored for specific application needs, such as increasing the buffer size for high-volume connections.

SO_RCVLOWAT
sets the receive low-water mark.

SO_RCVTIMEO
sets the timeout value for a receive operation.

SO_REUSEADDR
enables stream and datagram sockets to reuse local addresses. Local addresses that are already in use can then be bound. This alters the normal algorithm used in the SockBind() call. At connection time, the system checks whether the local addresses and ports differ from foreign addresses and ports. If not, the error value EADDRINUSE is returned.

SO_SNDBUF
Sets the buffer size for output. This option sets the size of the send buffer to the value contained in the buffer pointed to by *optval*. In this way, the send buffer size can be tailored for specific application needs, such as increasing the buffer size for high-volume connections.

SO_SNDLOWAT
sets the send low-water mark. This mark is ignored for nonblocking calls and not used in the Internet domain.

SO_SNDTIMEO
sets the timeout value for a send operation.

SO_USELOOPBACK
bypasses hardware where possible.

Except for SO_LINGER, all values are integral. SO_LINGER expects two blank delimited integers:

1. The l_onoff value. It is set to 0 if the SO_LINGER option is disabled.
2. the l_linger value. The l_linger field specifies the amount of time, in seconds, to be lingered on close. A value of 0 causes SockSoClose() to wait until disconnection completes.

Return values:

The value 0 indicates successful execution of the call. The value -1 indicates an error. You can get the specific error code SockSock_Errno() or SockPSock_Errno(). Possible values are:

EADDRINUSE
The address is already in use.

ENOTSOCK
socket is not a valid socket descriptor.

ENOPROTOOPT
optname is not recognized.

EINVAL
Invalid argument.

ENOBUFS
There is no buffer space available.

Note: SockSetSockOpt() interfaces with the C function setsockopt().

SockShutDown

The SockShutDown() call shuts down all, or part, of a full duplex connection. This call is optional.

Syntax:

```
rc = SockShutDown(socket, howto)
```

where:

socket
is the socket descriptor.

howto
is the condition of the shutdown of socket *socket*.

Because data flows in different directions are independent of each other, SockShutDown() allows you to independently stop data flows in one direction, or all data flows, with one API call. For example, you can enable yourself to send data but disable other senders to send data to you.

The *howto* parameter sets the condition for shutting down the connection to socket *socket*. It can be set to one of the following:

0 No more data can be received on socket *socket*.

1 No more output is allowed on socket *socket*.

2 No more data can be sent or received on socket *socket*.

Return values:

The value 0 indicates successful execution of the call. The value -1 indicates an error. You can get the specific error code SockSock_Errno() or SockPSock_Errno(). Possible values are:

ENOTSOCK
socket is not a valid socket descriptor.

EINVAL
howto was not set to a valid value.

Note: SockShutDown() interfaces with the C function shutdown().

SockSock_Errno

The SockSock_Errno() call returns the last error code set by a socket call. Subsequent socket API calls do not reset this error code.

Syntax:

```
errno = SockSock_Errno()
```

Note: SockSock_Errno() interfaces with the C function sock_errno().

SockSocket

The SockSocket() call creates an end point for communication and returns a socket descriptor representing the end point. Each socket type provides a different communication service.

Syntax:

```
socket = SockSocket(domain, type, protocol)
```

where:

domain
is the communication domain requested. It specifies the protocol family to be used. Currently, only the domain "AF_INET" is supported, which uses addresses in the Internet address format.

type
is the type of socket created. The following types are supported:

SOCK_STREAM
provides sequenced, two-way byte streams that are reliable and connection-oriented. It supports a mechanism for out-of-band data. Stream sockets are supported by the Internet ("AF_INET") communication domain.

SOCK_DGRAM
provides datagrams, which are connectionless messages of a fixed length whose reliability is not guaranteed. Datagrams can be received out of order, lost, or delivered several times. Datagram sockets are supported by the Internet ("AF_INET") communication domain.

SOCK_RAW
provides the interface to internal protocols, such as IP and ICMP. Raw sockets are supported by the Internet (″AF_INET″) communication domain.

protocol
is the protocol to be used with the socket. It can be ″IPPROTO_UDP″, ″IPPROTO_TCP″, or ″0″. If it is set to 0, which is the default, the system selects the default protocol number for the domain and socket type requested.

Sockets are deallocated with the SockClose() call.

Return values:

A non-negative socket descriptor return value indicates successful execution of the call. The return value -1 indicates an error. You can get the specific error code SockSock_Errno() or SockPSock_Errno(). Possible values are:

EMFILE
The maximum number of sockets are currently in use.

EPROTONOSUPPORT
The protocol is not supported in the specified domain or the protocol is not supported for the specified socket type.

EPFNOSUPPORT
The protocol family is not supported.

ESOCKTNOSUPPORT
The socket type is not supported.

Note: SockSocket() interfaces with the C function socket().

SockSoClose

The SockSoClose() call shuts down a socket and frees resources allocated to the socket.

Syntax:

```
rc = SockSoClose(socket)
```

where:

socket
is the socket descriptor of the socket to be closed.

This function is identical to SockClose().

Appendix. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Deutschland
Informationssysteme GmbH
Department 3982
Pascalstrasse 100

70569 Stuttgart
Germany

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement or any equivalent agreement between us.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

The following term is a trademark of the IBM Corporation in the United States, other countries, or both:

IBM

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

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