REXX/SOCKETS
VM REXX TCP/IP Socket Functions

Reference Manual

Document Number REXX/SOCKETS 2.01

2 June 1993

Arthur J. Ecock
Rainer F. Hauser

City University of New York
New York, USA
ECKCU at CUNYVM
eckcu@cunyvm.cuny.edu

IBM Research Laboratory
Zurich, Switzerland
RFH at ZURLVM1
rfh@zurich.ibm.com

Contents

1.0 Introduction

TCP/IP (Transmission Control Protocol/Internet Protocol) is a stack of communications protocols implemented on many different operating system platforms. The socket interface is a very popular and easy-to-use interface to the TCP/IP protocols. It is supported by TCP/IP for VM either via the VMCF or the IUCV API (application programming interface). The socket interface is usually described in the C programming language and is mainly used from this language.

REXX/SOCKETS is a REXX function package that provides access to the TCP/IP socket interface via the IUCV API through the RXSOCKET MODULE. It maps the socket calls from the C programming language to the REXX programming language. Thus, it enables a programmer to implement and test TCP/IP applications partially or completely in REXX using the convenient REXX paradigm to access the socket interface.

Functions for waiting on certain events are needed by many REXX programs. Especially, applications using communication protocols between different programs require such functions since one program needs to wait for an action by its communication partners. Such programs must be able to wait for one event from a list of expected events. Further, applications involving programs on different machines which either use the ASCII or the EBCDIC character encoding need also functions to translate from ASCII to EBCDIC encoding and vice versa. These functions are provided by REXX/WAIT and are used and supported by REXX/SOCKETS. (See REXXWAIT PACKAGE on the VMTOOLS conferencing disk.)

REXX/SOCKETS adds the function SOCKET, and REXX/WAIT adds the functions WAIT, SETVALUE, QUERYVALUE, RESETVALUE, AC2EC, EC2AC, CTYPE and CTABLE to the existing set of REXX built-in functions. REXX/WAIT also provides some basic event handlers for console, message, mail and time events, and REXX/SOCKETS provides an event handler for socket events.

Also REXX/SOCKETS provides support for ASCII/EBCDIC translation but only for all data sent and received on a socket. As soon as data exchanged on a socket consists of text and binary data, the REXX functions provided by REXX/WAIT must be used.

Before the REXX function provided by REXX/SOCKETS is described in detail, a few basic concepts of TCP/IP need to be described for REXX programmers using these functions. However, a user of the REXX interface to the TCP/IP socket interface must have some experience with communications in general and with TCP/IP in particular to make correct and efficient use of the socket interface.

A communication entity using TCP/IP protocols is identified by an address family and further identifiers specific to an address family. For AF_INET (the only address family supported by REXX/SOCKETS so far), these further identifiers are a port and an internet address. The internet address is a 32-bit quantity usually structured in one to four parts such as '128.228.1.2' when referenced. The internet address identifies a network interface within a host. The port is a number which allows the TCP/IP service to differentiate between different applications using the same network interface. Some ports are reserved for specific applications.

TCP/IP can be used to transport data from one application to another via two different protocols: TCP is a connection-oriented transport protocol, and UDP is a connectionless transport protocol. (The IP protocol is also a connectionless protocol but on a lower layer, and it is not further mentioned in this documentation.) When established, the connection between two programs in the connection-oriented case is a pair of queues (one for each direction). One program can add bytes to one side of the queue, and the other program can take out bytes from the other side. The pieces in which data is added do not have to correspond to the pieces as taken out. For example, one program may add 1000 bytes with one socket call and another 1000 bytes with another one. The other program may get the first 600 bytes with the first function call, the next 750 bytes with the second call and the last 650 bytes with a third call.

Some socket functions can be used in blocking or in non-blocking mode. In blocking mode, the function may wait and only returns when the function successfully completes or when it is clear that it cannot successfully complete. Initially, all functions use the blocking mode. In non-blocking mode, the function does

not wait but tells with a return code that it would block. In this mode, waiting must be provided by other means. REXX/SOCKETS supports waiting in non-blocking mode through REXX/WAIT and through the select socket call.

The first call of the REXX function SOCKET loads REXX/SOCKETS as a nucleus extension. The call `NUCXDROP RXSOCKET` terminates all socket sets for the session, closes all sockets and drops REXX/SOCKETS as a nucleus extension. Thus, this function should only be used with care.

2.0 REXX Function Syntax

REXX/SOCKETS adds the REXX built-in function SOCKET to the set of other REXX functions available. The first parameter in the REXX function SOCKET is the name of the subfunction to be called. The general behavior of the REXX function SOCKET is described.

2.1 Function SOCKET

The SOCKET function provides access to the TCP/IP socket interface. The actual socket calls are specified in the first parameter and is called **subfunction** in the following. The further parameters **arg** depend on the subfunction to be called.

> **SOCKET(subfunction,[arg],...,[arg])**
>
> Result: rc [string]

The result string contains several tokens separated by a blank such that it can easily be parsed by REXX. The first token is a return code. If the return code is zero, the further tokens depend on the subfunction called. If the return code is not zero, the second token is the name of the error, and the rest is the corresponding error message.

```
Socket('GetHostId')          ==   '0 9.4.3.2'
Socket('Recv',socket)        ==   '35 EWOULDBLOCK Operation would block'
```

2.2 Argument Names

The input parameters for the individual subfunctions of the REXX built-in function SOCKET and the items in the result string (except for the return code) together with their range and default values are described in alphabetical order in the following list:

backlog Number of pending connect requests (integer between 0 and 10)
Default: 10
Input to LISTEN

clientid Identifier for an application in the form: domain userid subtaskid
Input to GIVESOCKET, TAKESOCKET
Output of GETCLIENTID

connectinfo Further information for the socket set status 'Connected' ('Free' integer 'Used' integer)
Output of SOCKETSETSTATUS

count Number of socketids ready for the given operation (non-negative integer)
Output of SELECT

data Message string for data exchange (arbitrary character string)
Input to WRITE, SEND, SENDTO
Output of READ, RECV, RECVFROM

date Version date of the REXX/SOCKETS function package
Output of VERSION

domain Addressing family (only 'AF_INET' is supported so far)
Input to SOCKET, GETCLIENTID

domainname Name of the domain under which the processor falls (character string)
Output of GETDOMAINNAME

fcmd Operating characteristics command ('F_SETFL', 'F_GETFL')
Input to FCNTL

fullhostname Fully qualified hostname in the form: domainname.hostname
Input to GETHOSTBYNAME, RESOLVE
Output of GETHOSTBYADDR, RESOLVE

fvalue Operating characteristics value ('Blocking', '0', 'Non-Blocking', 'FNDELAY')
Input to FCNTL
Output of FCNTL

hissocketid Socket descriptor belonging to another clientid (socketid)
Input to TAKESOCKET

hostname Name of a host processor (string)
Input to GETHOSTBYNAME, RESOLVE
Output of GETHOSTNAME

how Part of a duplex connection ('BOTH', 'FROM', 'TO', 'READ', 'WRITE', 'READING', 'WRITING', 'RECEIVE', 'SEND', 'RECEIVING', 'SENDING')
Default: 'BOTH'
Input to SHUTDOWN

icmd Operating characteristics command

'FIONBIO'	Sets or clears non-blocking input-output for a socket depending on the value specified for ivalue. (The ivalue arguments can be 'On' or 'Off'.)
'FIONREAD'	Gets the number of immediately readable bytes for the socket and returns it as ivalue (a non-negative integer).
'SIOCATMARK'	Queries whether the current location in the data input is pointing to out-of-band data. (It returns either 'Yes' or 'No'.)
'SIOCGIFADDR'	Gets the network interface address. (The ivalue argument is an interface, and the function returns the address in the form: interface domain port ipaddress.)
'SIOCGIFBRDADDR'	Gets the network interface broadcast address. (The ivalue argument is an interface, and the function returns the address in the form: interface domain port ipaddress.)
'SIOCGIFCONF'	Gets the network interface configuration. (The ivalue argument is the maximum number of interfaces to be returned, and the function returns a list of interfaces in the form: interface domain port ipaddress.)
'SIOCGIFDSTADDR'	Gets the network interface destination address. (The ivalue argument is an interface, and the function returns the address in the form: interface domain port ipaddress.)
'SIOCGIFFLAGS'	Gets the network interface flags. (The ivalue argument is an interface, and the function returns the interface together with the flags as four hexadecimal digits and the symbolic names of those flags which are enabled.)
'SIOCGIFMETRIC'	Gets the network interface routing metric. (The ivalue argument is an interface, and the function returns the interface together with the metric as an integer.)
'SIOCGIFNETMASK'	Gets the network interface network mask. (The ivalue argument is an interface, and the function returns the mask in the form: interface domain port ipaddress.)

Input to IOCTL

ipaddress Internet address (in dotted notation, 'INADDR_ANY', 'ANY', 'INADDR_BROADCAST', 'BROADCAST')
Input to GETHOSTBYADDR, RESOLVE
Output of GETHOSTID, RESOLVE

ipaddresslist List of internet addresses (in dotted notation)
Output of GETHOSTBYNAME

ivalue Operating characteristics value (depending on the value specified for icmd)
Input to IOCTL
Output of IOCTL

length Length of data (non-negative integer)
Output of READ, WRITE, RECV, SEND, RECVFROM, SENDTO

level Protocol level ('SOL_SOCKET', 'IPPROTO_TCP')
Input to GETSOCKOPT, SETSOCKOPT

maxdesc Number of pre-allocated sockets in a socket set (integer between 1 and 2,000)
Default: 40
Input to INITIALIZE
Output of INITIALIZE

maxlength Maximum data length (integer between 1 and 100,000)
Default: 10,000
Input to READ, RECV, RECVFROM

name Network address in the form: domain portid ipaddress
Input to BIND, CONNECT, SENDTO
Output of ACCEPT, RECVFROM, GETPEERNAME, GETSOCKNAME

newsocketid Newly created socket descriptor (socketid)
Output of SOCKET, ACCEPT, TAKESOCKET

optname Option name

'SO_ASCII' Sets or clears the data encoding characteristics for the socket. When activated, the SO_EBCDIC flag gets cleared, and it is ensured that all data sent and received on the socket gets translated to ASCII if needed. When disabled, no data translation from and to ASCII is applied. This option has only an effect on hosts which use EBCDIC data encoding but can also be used on ASCII systems. (The optvalue arguments can be 'On', optionally followed by the filename of a translation table, or 'Off'. As output, 'On' or 'Off' is returned optionally followed by the name of the translation table used, if translation will be applied to data.)

'SO_BROADCAST' Sets or clears the ability to broadcast messages. If this option is enabled, it allows the application to send broadcast messages over the socket, if the interface specified in the destination support broadcasting of packets. This option has no meaning for stream sockets. (The optvalue argument can be 'On' or 'Off'.)

'SO_DEBUG' Sets or clears the debugging option. (The optvalue argument can be 'On' or 'Off'.)

'SO_EBCDIC' Sets or clears the data encoding characteristics for the socket. When activated, the SO_ASCII flag gets cleared, and it is ensured that all data sent and received on the socket gets translated to EBCDIC if needed. When disabled, no data translation from and to EBCDIC is applied. This option has only an effect on hosts which use ASCII data encoding but can also be used on EBCDIC systems. (The optvalue arguments can be 'On', optionally followed by the filename of a translation table, or 'Off'. As output, 'On' or

'Off' is returned optionally followed by the name of the translation table used, if translation will be applied to data.)

'SO_ERROR' Returns any pending error on the socket and clears the error status. It can be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors that are not returned explicitly by one of the socket calls. (The optvalue argument can be 'On' or 'Off'.)

'SO_KEEPALIVE' Sets or clears the TCP keep-alive mechanism for a stream socket. When activated, the keep-alive mechanism periodically sends a packet on an otherwise idle connection. If the remote TCP does not respond to the packet or to retransmissions of the packet, the connection is terminated with the error ETIMEDOUT. (The optvalue argument can be 'On' or 'Off'.)

'SO_LINGER' Lingers on close if data is present. When this option is enabled and there is unsent data present when the subfunction CLOSE is called, the calling application is blocked during the CLOSE subfunction call until the data is transmitted or the connection has timed out. If this option is disabled, the TCPIP virtual machine waits to try to send the data. Although the data transfer is usually successful, it cannot be guaranteed, because the TCPIP virtual machine waits only a finite amount of time trying to send the data. The CLOSE subfunction call returns without blocking the caller. This option has meaning only for stream sockets. (The optvalue argument can be 'On', optionally followed by a number, or 'Off'. If 'On' is selected, the default number is 120.)

'SO_OOBINLINE' Sets or clears reception of out-of-band data. When this option is enabled, it causes out-of-band data to be placed in the normal data input queue as it is received, making it available to the RECV and RECVFROM subfunction without having to specify the MSG_OOB flag in those calls. When this option is disabled, it causes out-of-band data to be placed in the priority data input queue as it is received, making it available to the RECV and RECVFROM subfunctions only by specifying the MSG_OOB flag in those calls. This option has meaning only for stream sockets. (The optvalue argument can be 'On' or 'Off'.)

'SO_REUSEADDR' Sets or clears local address reuse. When enabled, this option allows local addresses that are already in use to be bound. This alters the normal algorithm used in the BIND subfunction call. The system checks at connect time to be sure that no local address and port have the same foreign address and port. The error EADDRINUSE is returned if the association already exists. (The optvalue argument can be 'On' or 'Off'.)

'SO_SNDBUF' Returns the size of the send buffer in bytes. (No optvalue argument is allowed.)

'SO_TYPE' This option returns the type of the socket. On return, optvalue is one of 'SOCK_STREAM', 'SOCK_DGRAM', or 'SOCK_RAW'. (No optvalue argument is allowed.)

'TCP_NODELAY' Sets or clears the TCP_NODELAY flag. (The optvalue argument can be 'On' or 'Off'.)

Input to GETSOCKOPT, SETSOCKOPT

optvalue Option value ('On', 'Off' and so on)
Input to SETSOCKOPT
Output of GETSOCKOPT

portid Port number (non-negative integer between 0 and 65,535, 'INPORT_ANY', 'ANY')
Input to GETSERVBYPORT
Output of GETSERVBYNAME, GETSERVBYPORT

protocol Protocol name ('0', 'IPPROTO_UDP', 'UDP', 'IPPROTO_TCP', 'TCP')
Default: '0'
Input to SOCKET

protocolname Name of a network protocol ('TCP', 'UDP', 'IP' and so on)
Input to GETPROTOBYNAME, GETSERVBYNAME, GETSERVBYPORT
Output of GETPROTOBYNUMBER, GETSERVBYNAME, GETSERVBYPORT

protocolnumber Number of a network protocol (non-negative integer)
Input to GETPROTOBYNUMBER
Output of GETPROTOBYNAME

recvflags Receive flags ('MSG_OOB', 'OOB', 'OUT_OF_BAND', 'MSG_PEEK', 'PEEK')
Default: ''
Input to RECV, RECVFROM

sendflags Send flags ('MSG_OOB', 'OOB', 'OUT_OF_BAND', 'MSG_DONTROUTE', 'DONTROUTE')
Default: ''
Input to SEND, SENDTO

service Name of the TCP/IP virtual machine
Default: To be determined from TCPIP DATA
Input to INITIALIZE
Output of INITIALIZE

servicename Name of a service ('FTP' and so on)
Input to GETSERVBYNAME
Output of GETSERVBYNAME, GETSERVBYPORT

severreason Reason for the socket set status 'Severed'
Output of SOCKETSETSTATUS

socketid Socket descriptor (socketid)
Input to BIND, LISTEN, CONNECT, ACCEPT, SHUTDOWN, CLOSE, GIVESOCKET, READ, WRITE, RECV, SEND, RECVFROM, SENDTO, GETPEERNAME, GETSOCKNAME, GETSOCKOPT, SETSOCKOPT, FCNTL, IOCTL

socketidlist List of socket descriptors (socketids)
Input to SELECT
Output of SELECT

status Status of a socket set ('Free', 'Connected', 'Severed', 'Hanging')
Output of SOCKETSETSTATUS

subtaskid Name for a socket set (up to 8 printable characters as allowed in CMS filenames and without blanks)
Input to INITIALIZE, TERMINATE, SOCKETSET, SOCKETSETSTATUS
Output of INITIALIZE, TERMINATE, SOCKETSET, SOCKETSETSTATUS

subtaskidlist List of subtaskids
Output of SOCKETSETLIST

type Socket type ('SOCK_STREAM', 'STREAM', 'SOCK_DGRAM', 'DATAGRAM', 'SOCK_RAW', 'RAW')
Input to SOCKET

timeout Maximum number of seconds to wait (non-negative integer, 'FOREVER')
Default: 'FOREVER'
Input to SELECT

version Version number of the REXX/SOCKETS function package
Output of VERSION

2.3 Error Messages and Return Codes

The first call of the REXX built-in function SOCKET loads RXSOCKET as a nucleus extension. The following error and warning messages may be displayed during this process or during the INITIALIZE subfunction.

RXSOCK004E Insufficient storage available to load RXSOCKET as a Nucleus Extension

RXSOCK008E Unable to establish RXSOCKET as a Nucleus Extension

RXSOCK012E RXSOCKET requires TCPIP Version 2 or higher

RXSOCK028W File TCPIP DATA * not found

The REXX built-in function SOCKET returns a return code as the first token of the result string. The following values are defined by REXX/SOCKETS for all subfunctions:

0 Normal return
1 EPERM - Not owner
2 ENOENT - No such file or directory
3 ESRCH - No such process
4 EINTR - Interrupted system call
5 EIO - I/O error
6 ENXIO - No such device or address
7 E2BIG - Arg list too long
8 ENOEXEC - Exec format error
9 EBADF - Bad file number
10 ECHILD - No children
11 EAGAIN - No more processes
12 ENOMEM - Not enough memory
13 EACCES - Permission denied
14 EFAULT - Bad address
15 ENOTBLK - Block device required
16 EBUSY - Device busy
17 EEXIST - File exists
18 EXDEV - Cross-device link
19 ENODEV - No such device
20 ENOTDIR - Not a directory
21 EISDIR - Is a directory
22 EINVAL - Invalid argument
23 ENFILE - File table overflow
24 EMFILE - Too many open files
25 ENOTTY - Inappropriate ioctl for device
26 ETXTBSY - Text file busy
27 EFBIG - File too large
28 ENOSPC - No space left on device
29 ESPIPE - Illegal seek
30 EROFS - Read-only file system
31 EMLINK - Too many links
32 EPIPE - Broken pipe
33 EDOM - Argument too large
34 ERANGE - Result too large
35 EWOULDBLOCK - Operation would block
36 EINPROGRESS - Operation now in progress
37 EALREADY - Operation aleady in progress
38 ENOTSOCK - Socket operation on non-socket
39 EDESTADDRREQ - Destination address required

40 EMSGSIZE - Message too long
41 EPROTOTYPE - Protocol wrong type for socket
42 ENOPROTOOPT - Option not supported by protocol
43 EPROTONOSUPPORT - Protocol not supported
44 ESOCKTNOSUPPORT - Socket type not supported
45 EOPNOTSUPP - Operation not supported on socket
46 EPFNOSUPPORT - Protocol family not supported
47 EAFNOSUPPORT - Address family not supported by protocol family
48 EADDRINUSE - Address already in use
49 EADDRNOTAVAIL - Cant assign requested address
50 ENETDOWN - Network is down
51 ENETUNREACH - Network is unreachable
52 ENETRESET - Network dropped connection on reset
53 ECONNABORTED - Software caused connection abort
54 ECONNRESET - Connection reset by peer
55 ENOBUFS - No buffer space available
56 EISCONN - Socket is already connected
57 ENOTCONN - Socket is not connected
58 ESHUTDOWN - Cant send after socket shutdown
59 ETOOMANYREFS - Too many references: cant splice
60 ETIMEDOUT - Connection timed out
61 ECONNREFUSED - Connection refused
62 ELOOP - Too many levels of symbolic links
63 ENAMETOOLONG - File name too long
64 EHOSTDOWN - Host is down
65 EHOSTUNREACH - Host is unreachable
66 ENOTEMPTY - Directory not empty
67 EPROCLIM - Too many processes
68 EUSERS - Too many users
69 EDQUOT - Disc quota exceeded
70 ESTALE - Stale NFS file handle
71 EREMOTE - Too many levels of remote in path
72 ENOSTR - Not a stream device
73 ETIME - Timer expired
74 ENOSR - Out of stream resources
75 ENOMSG - No message of desired type
76 EBADMSG - Not a data message
77 EIDRM - Identifier removed
78 EDEADLK - Deadlock situation detected/avoided
79 ENOLCK - No record locks available
80 ENONET - Machine is not on the network
81 ERREMOTE - Object is remote
82 ENOLINK - The link has been severed
83 EADV - Advertise error
84 ESRMNT - SRMOUNT error
85 ECOMM - Communication error on send
86 EPROTO - Protocol error
87 EMULTIHOP - Multihop attempted
88 EDOTDOT - Cross mount point
89 EREMCHG - Remote address changed
90 ECONNCLOSED - Connection closed by peer
1000 EIBMBADCALL - Bad socket-call constant
1001 EIBMBADPARM - Bad parm
1002 EIBMSOCKETOUTOFRANGE - Socket out of range
1003 EIBMSOCKINUSE - Socket in use
1004 EIBMIUCVERR - IUCV error
2001 EINVALIDRXSOCKETCALL - Syntax error in RXSOCKET parameter list
2002 ECONSOLEINTERRUPT - Console interrupt
2003 ESUBTASKINVALID - Subtask ID invalid

2004 ESUBTASKALREADYACTIVE - Subtask already active
2005 ESUBTASKNOTACTIVE - Subtask not active
2006 ESOCKETNOTALLOCATED - Socket could not be allocated
2007 EMAXSOCKETSREACHED - Maximum number of sockets reached
2008 ESOCKETALREADYDEFINED - Socket already defined
2009 ESOCKETNOTDEFINED - Socket not defined
2010 ETCPIPSEVEREDPATH - TCPIP severed IUCV path
2011 EDOMAINSERVERFAILURE - Domain name server failure
2012 EINVALIDNAME - Invalid "name" received from TCPIP server
2013 EINVALIDCLIENTID - Invalid "clientid" received from TCPIP server
2014 EINVALIDFILENAME - Invalid filename specified
2015 ENUCEXTFAILURE - Error during NUCEXT function
2051 EFORMATERROR - Format error
2052 ESERVERFAILURE - Server failure
2053 EUNKNOWNHOST - Unknown host
2054 EQUERYTYPENOTIMPLEMENTED - Query type not implemented
2055 EQUERYREFUSED - Query refused
3001 EIUCVINVALIDPATH - Invalid IUCV path-id
3002 EIUCVPATHQUIESCED - IUCV path quiesced
3003 EIUCVMSGLIMITEXCEEDED - IUCV message limit exceeded
3004 EIUCVNOPRIORITY - IUCV priority message not allowed on this path
3005 EIUCVSMALLBUFFER - IUCV buffer too small
3006 EIUCVBADFETCH - IUCV Fetch Protection Exception
3007 EIUCVBADADDRESS - IUCV Addressing Exception on answer buffer
3008 EIUCVBADMSGCLASS - IUCV conflicting message class/path/msgid
3009 EIUCVPURGEDMSG - IUCV message was purged
3010 EIUCVBADMSGLENGTH - IUCV negative message length
3011 EIUCVPARTNERNOTLOGGEDON - IUCV target userid not logged on
3012 EIUCVPARTNERNOTINITIALIZED - IUCV target userid not enabled for IUCV
3013 EIUCVPATHLIMITEXCEEDED - IUCV path limit exceeded
3014 EIUCVPARTNERPATHLIMIT - IUCV partner path limit exceeded
3015 EIUCVNOTAUTHORIZED - IUCV not authorized
3016 EIUCVINVALIDCPSYSTEMSERVICE - IUCV invalid CP System Service
3018 EIUCVINVALIDMSGLIMIT - IUCV invalid message limit
3019 EIUCVBUFFERALREADYDECLARED - IUCV buffer already declared
3020 EIUCVPARTNERSEVERED - IUCV partner severed path
3021 EIUCVPARTNERNOPRMDATA - IUCV cannot accept data in parmlist
3022 EIUCVSENDLISTINVALID - IUCV SEND buffer list is invalid
3023 EIUCVINVALIDBUFFERLENGTH - IUCV negative length in answerlist
3024 EIUCVINVALIDLISTLENGTH - IUCV total list length is invalid
3025 EIUCVPRMMSGANSLISTCONFLICT - IUCV PRMMSG/answer-list conflict
3026 EIUCVBUFFERLISTNOTALIGNED - IUCV buffer list not aligned
3027 EIUCVANSWERLISTNOTALIGNED - IUCV answer list not aligned
3028 EIUCVNOCONTROLBUFFER - IUCV no control buffer
3048 EIUCVFUNCTIONNOTSUPPORTED - IUCV function not supported

2.4 General Remarks

This section presents some general remarks for REXX programmers using the SOCKET function and its subfunctions to access the socket interface.

1. In order to use socket related subfunctions of the SOCKET function, one socket set must be active. The INITIALIZE subfunction creates a socket set, and multiple calls of the INITIALIZE subfunction are allowed. The subtaskid for a socket set identifies the socket set and usually corresponds to the application name.

2. The argument called name consists of a domain, a portid and an ipaddress. As input to the SOCKET function, the ipaddress can be specified as a name to be resolved by the name server: 'CUNYVM' or

'CUNYVM.CUNY.EDU', for example. As an output, the ipaddress is always in the dotted form: '128.228.1.2', for example.

3. A socket can be in blocking or non-blocking mode. In blocking mode, subfunctions like SEND and RECV block the caller until the operation completed successfully or an error occurs. In non-blocking mode, the caller is never blocked, but the operation terminates immediately with a return code such as EWOULDBLOCK or EINPROCESS. The FCNTL or IOCTL subfunctions can be used to switch between blocking and non-blocking mode. Also the REXX built-in function SETVALUE allows changing the mode.

4. When a socket is in non-blocking mode, the REXX built-in function WAIT allows also waiting for socket events. Arrival of data on a socket for the READ or RECV subfunction, for example, is a possible event. When a socket is not ready for sending data because the buffer space is not available at the socket to hold the message to be transmitted, a REXX program can also wait until the socket is ready again for sending data.

5. Applications using the GIVESOCK and TAKESOCK functions need an agreed-upon mechanism for exchanging their clientids. The caller of the TAKESOCK function (the slave) needs to know also the socketid of the other program (the master). When the master issues CP AUTOLOG to start the slave, it can pass its part of the information as a parameter. The slave must still somehow signal when the TAKESOCK function successfully completed in order to allow the master to close the socket. For this purpose, or for the whole handshake needed to transfer a socket, SMSG's and the SMSG event handler of REXX/WAIT can be used.

6. The socket options SO_ASCII and SO_EBCDIC tell REXX/SOCKETS which data encoding is to be used for the socket. On VM, setting SO_EBCDIC on has no effect, and setting SO_ASCII on causes all incoming data on the socket to be translated from ASCII to EBCDIC and all outgoing data on the socket to be translated from EBCDIC to ASCII. If the translation table has not been specified explicitly with the SETSOCKOPT subfunction, REXX/SOCKETS uses the following hierarchy of translation tables:

   ```
   subtaskid 'TCPXLBIN *'
   userid    'TCPXLBIN *'
   'STANDARD  TCPXLBIN *'
   'RXSOCKET  TCPXLBIN *'
   Internal tables
   ```

   The first four are files searched in the given order. If none of those files are found, the internal tables corresponding to the ISO standard translation tables are used.

7. The fact that options for the GETSOCKOPT and SETSOCKOPT subfunctions and cmds for the IOCTL subfunction are listed does not necessarily mean that they are supported by the TCPIP service. It only means that REXX/SOCKETS forwards them to the TCPIP service and returns whatever the TCPIP service virtual machine returns.

8. Not all return codes listed for the REXX built-in function SOCKET will ever occur. Especially, most IUCV related return codes will be avoided by REXX/SOCKETS, but are still listed for completeness.

2.5 The REXX/WAIT Support

The name 'SOCKET' is used for the event handler provided by REXX/SOCKETS for socket events. Socket events can be specified as lists of socketids for read or write operations or for exceptions. The socketids in the lists must be in the currently active socket set. For all sockets in the active socket set, the list can be specified as '*' for read, write and exception events.

WAIT(...,'SOCKET' [**'READ' socketidlist**][**'WRITE' socketidlist**][**'EXCEPTION' socketidlist**],...**)**

Result: rc 'SOCKET' 'READ'|'WRITE'|'EXCEPTION' socketid

The function call waits for a socket event and returns 'SOCKET', a keyword ('READ', 'WRITE' or 'EXCEPTION') and the first socketid for which the respective event happened. It is a replacement for the SELECT subfunction of the REXX built-in function SOCKET allowing more flexible event handling.

A close on the other side of a connection is not reported as an exception, but as a READ event which returns zero bytes of data. When the CONNECT subfunction is called with a socket in non-blocking mode, it terminates with a return code EINPROGRESS, and the completion of the connection setup gets reported as a WRITE event on the socket. When the ACCEPT subfunction is called with a socket in in non-blocking mode, it terminates with a return code EWOULDBLOCK. However, the availability of a connection request gets reported as a READ event on the original socket, and the ACCEPT subfunction should only be called then.

SETVALUE('SOCKET' ['READ' socketidlist]['WRITE' socketidlist]['EXCEPTION' socketidlist])

Result: rc olddefaults

The function call sets the defaults for the socket set and returns the default settings active before the new values are set. Initially, the default is 'READ * WRITE * EXCEPTION *' meaning wait for any socket event on any socket in the active socket set.

SETVALUE('SOCKET' socketid 'BLOCKING'|'NON-BLOCKING')

Result: rc 'BLOCKING'|'NON-BLOCKING'

The function call allows setting a socket with given id to blocking or non-blocking mode. It returns the setting active before the new mode is set. (The socket must be in the currently active socket set.) This function can be used instead of the FCNTL and IOCTL (with the option FIONBIO) subfunction of the REXX built-in function SOCKET to switch between blocking and non-blocking mode.

QUERYVALUE('SOCKET VERSION')

Result: rc 'REXX/SOCKETS' version date

The function returns the name ('REXX/SOCKETS'), the version number and the version date of the REXX/SOCKETS function package.

QUERYVALUE('SOCKET DEFAULTS')

Result: rc defaults

The function call returns the current default settings for the currently active socket set.

QUERYVALUE('SOCKET' socketid)

Result: rc 'BLOCKING'|'NON-BLOCKING'

The function call returns the mode setting for the specified socket. (The socket must be in the currently active socket set.)

RESETVALUE('SOCKET')

Result: rc

The function call resets the defaults for the socket events for the active socket set and returns no data.

```
ResetValue('Socket')          ==   '0'
SetValue('Socket Read 5')     ==   '0 READ * WRITE * EXCEPTION *'
Wait('Socket Write 7 Read 5') ==   '0 SOCKET READ 5'            /* Perhaps */
```

Additional Return Codes: In addition to the common values defined by REXX/WAIT, the functions return the following return codes:

10 IUCV problem
11 No active socket set
12 Socket not in active socket set

3.0 SOCKET Subfunctions

All available subfunctions of the REXX built-in function SOCKET with their additional arguments are discussed in detail. Also their functionality and the equivalent function in the C programming language are shown if existing.

3.1 Subfunctions to Manipulate Socket Sets

A socket set is a number of pre-allocated sockets available to a single application. Multiple socket sets may be defined for one session, but only one can be active at a single point in time.

SOCKET('INITIALIZE',subtaskid,[maxdesc],[service])

Result: rc subtaskid maxdesc service

The function pre-allocates the number of sockets specified. As result, it returns the subtaskid, the maximum number of pre-allocated sockets and the name of the virtual machine providing the TCP/IP service. The socket set identified by the subtaskid will automatically become the active socket set, if the call was successful.

```
Socket('Initialize','myId')   ==   '0 myId 40 TCPIP'
```

SOCKET('TERMINATE',[subtaskid])

Result: rc subtaskid

The function closes all sockets of the socket set, releases the socket set. As result, it returns the subtaskid of the socket set which got terminated. If the subtaskid is not specified, the active socket set is terminated. If the active socket set is terminated, the next socket set in the stack (if available) becomes the active socket set.

```
Socket('Terminate','myId')    ==   '0 myId'
```

SOCKET('SOCKETSETLIST')

Result: rc subtaskidlist

The function returns a list of the subtaskids for all available socket sets in the current order of the stack.

```
Socket('SocketSetList')       ==   '0 myId firstId'
```

SOCKET('SOCKETSET',[subtaskid])

Result: rc subtaskid

The function returns the subtaskid of the active socket set, and optionally makes the specified socket set the active socket set.

```
Socket('SocketSet','firstId') ==   '0 myId'
```

SOCKET('SOCKETSETSTATUS',[subtaskid])

Result: rc subtaskid status [connectinfo|severreason]

The function returns the status of the socket set and possibly further information. If connected, it also returns the number of free and the number of allocated sockets in the socket set. If severed, also the reason why the socket set got severed from the TCP/IP service is returned. If the subtaskid is not specified, the active socket set is used. (Initialized socket sets should be in status 'Connected', and not initialized socket sets are in status 'Free'.)

```
Socket('SocketSetStatus')     ==   '0 myId Connected Free 17 Used 23'
```

Note: An initialized socket set which is not in status 'Connected' must also be terminated before the subtaskid can be reused.

3.2 Subfunctions to Manipulate Sockets

A socket is an endpoint for communication that can be named and addressed in a network. It is represented by a socketid (also called socket descriptor). Any socketid used in a subfunction must be in the active socket set.

SOCKET('SOCKET',[domain],[type],[protocol])

Result: rc newsocketid

C socket call: socket(domain, type, protocol)

The function creates an endpoint for communication (i.e. a socket in the active socket set) and returns a socketid. Different types of sockets provide different communication services.

```
Socket('Socket')              ==   '0 5'
```

SOCKET('BIND',socketid,name)

Result: rc

C socket call: bind(s, name, namelen)

The function binds a unique local name to the socket with the given socketid. After calling the SOCKET subfunction, a socket does not have a name associated with it. However, it does belong to a particular addressing family. The exact form of a name depends on the addressing family. The BIND subfunction also allows servers to specify from which network interfaces they wish to receive UDP packets and TCP connection requests.

```
Socket('Bind',5,'AF_INET 1234 128.228.1.2')   ==   '0'
```

SOCKET('LISTEN',socketid,[backlog])

Result: rc

C socket call: listen(s, backlog)

The function which applies only to stream sockets performs two tasks: it completes the binding necessary for a socket (if BIND has not been called), and it creates a connection request queue of length specified as backlog for incoming connection requests. Once full, additional connection requests are ignored.

```
Socket('Listen',5,10)         ==   '0'
```

SOCKET('CONNECT',socketid,name)

Result: rc

C socket call: connect(s, name, namelen)

For stream sockets, this function performs two tasks: it completes the binding necessary for a socket (if BIND has not been called), and it attempts to establish a connection to another socket. For datagram sockets, this function specifies the peer for a socket. If the socket is in blocking mode, this function blocks the caller until the connection is established, or until an error is received. If the socket is in non-blocking mode, this function terminates with a return code EINPROGRESS (or another return code indicating an error).

```
Socket('Connect',5,'AF_INET 1234 128.228.1.2')       ==   '0'
Socket('Connect',5,'AF_INET 1234 CUNYVM')            ==   '0'
Socket('Connect',5,'AF_INET 1234 CUNYVM.CUNY.EDU')   ==   '0'
```

SOCKET('ACCEPT',socketid)

Result: rc newsocketid name

C socket call: accept(s, name, namelen)

This function is used by a server to accept a connection request from a client. It accepts the first connection on its queue of pending connections. It creates a new socketid with the same properties as the given socketid. If the queue has no pending connection requests, the function blocks the caller unless the socket is in non-blocking mode. If no connection request is pending and the socket is in non-blocking mode, the function terminates with a return code EWOULDBLOCK. The original socket remains available to accept more connection requests.

```
Socket('Accept',5)            ==   '0 6 AF_INET 5678 9.4.3.2'
```

SOCKET('SHUTDOWN',socketid,[how])

Result: rc

C socket call: shutdown(s, how)

This function shuts down all or part of a duplex connection.

```
Socket('ShutDown',6,'BOTH')   ==   '0'
```

SOCKET('CLOSE',socketid)

Result: rc

C socket call: close(s)

This function shuts down the socket associated with the socketid, and frees resources allocated to the socket. If the socketid refers to an open TCP connection, the connection is closed. If a stream socket is closed when there is input data queued, the TCP connection is reset rather than being cleanly closed.

```
Socket('Close',6)             ==   '0'
```

SOCKET('GIVESOCKET',socketid,clientid)

Result: rc

C socket call: givesocket(s, clientid)

The socket with the given socketid is transferred to another application. This function tells TCPIP to make the specified socket available to a TAKESOCKET subfunction issued by another application running on the same host. Any connected stream socket can be given. Typically, this function is used by a master program that obtains sockets by means of the ACCEPT subfunction and gives them to slave programs that handle one socket at a time.

```
Socket('GiveSocket',6,'AF_INET USERID2 hisId')   ==   '0'
```

SOCKET('TAKESOCKET',clientid,hissocketid)

Result: rc newsocketid

C socket call: takesocket(clientid, hisdesc)

This function acquires a socket from another program which obtains the other program's clientid and socketid by means not defined by TCP/IP. After successful calling the TAKESOCKET subfunction, the other program must close the socket.

```
Socket('TakeSocket','AF_INET USERID1 myId',6)   ==   '0 7'
```

3.3 Subfunctions to Exchange Data

On a connected stream socket and on datagram sockets, data can be sent and received. Different subfunctions are available for different purposes.

SOCKET('READ',socketid,[maxlength])

Result: rc length data

C socket call: read(s, buf, len)

This function reads up to maxlength bytes of data. If less than the number of bytes requested is available, the function returns the number currently available. If data is not available at the socket, the function waits for data to arrive and blocks the caller, unless the socket is in non-blocking mode. If the length is zero, the other side closed the stream socket.

```
Socket('Read',6)              ==   '0 21 This is the data line'
```

SOCKET('WRITE',socketid,data)

Result: rc length

C socket call: write(s, buf, len)

This function writes the given bytes of data. If writing the number of bytes requested is not possible, the function waits for writing to be possible. This blocks the caller, unless the socket is in non-blocking mode.

```
Socket('Write',6,'Some text')   ==   '0 9'
```

SOCKET('RECV',socketid,[maxlength],[recvflags])

Result: rc length data

C socket call: recv(s, buf, len, flags)

This function receives up to maxlength bytes of data (i.e., of the incoming message). If more than the number of bytes requested is available on a datagram socket, the function discards excess bytes. If less than the number of bytes requested is available, the function returns the number currently available. If data is not available at the socket, the function waits for data to arrive and blocks the caller, unless the socket is in non-blocking mode. If the length is zero, the other side closed the stream socket.

```
Socket('Recv',6)             ==   '0 21 This is the data line'
Socket('Recv',6,,'PEEK OOB')  ==   '0 24 This is out-of-band data'
```

SOCKET('SEND',socketid,data,[sendflags])

Result: rc length

C socket call: send(s, buf, len, flags)

This function sends the given bytes of data (i.e., the outgoing message) and applies to all connected sockets. If sending the number of bytes requested is not possible, the function waits for sending to be possible. This blocks the caller, unless the socket is in non-blocking mode.

```
Socket('Send',6,'Some text')               ==   '0 9'
Socket('Send',6,'Out-of-band data','OOB')   ==   '0 16'
```

SOCKET('RECVFROM',socketid,[maxlength],[recvflags])

Result: rc name length data

C socket call: recvfrom(s, buf, len, flags, name, namelen)

This function receives up to maxlength bytes of data (i.e., of the incoming message). If more than the number of bytes requested is available on a datagram socket, the function discards excess bytes. If less than the number of bytes requested is available, the function returns the number currently available. If data is not available at the socket, the function waits for data to arrive and blocks the caller, unless the socket is in non-blocking mode.

```
Socket('RecvFrom',6)          ==   '0 AF_INET 5678 9.4.3.2 9 Data line'
```

SOCKET('SENDTO',socketid,data,[sendflags],name)

Result: rc length

C socket call: sendto(s, buf, len, flags, name, namelen)

This function sends the given bytes of data (i.e., the outgoing message) to the given name. If sending the number of bytes requested is not possible, the function waits for sending to be possible. This blocks the caller, unless the socket is in non-blocking mode.

```
Socket('SendTo',6,'Some text',,'AF_INET 5678 9.4.3.2')                 ==   '0 9'
Socket('SendTo',6,'Some text',,'AF_INET 5678 ZURLVM1')                 ==   '0 9'
Socket('SendTo',6,'Some text',,'AF_INET 5678 ZURLVM1.ZURICH.IBM.COM')  ==   '0 9'
```

3.4 Subfunctions for Name Resolution

Various subfunctions allow a REXX program to obtain information about names, ipaddresses, clientids and so on. Other subfunctions call name servers to resolve ipaddress to symbolic names and vice versa.

SOCKET('GETCLIENTID',[domain])

Result: rc clientid

C socket call: getclientid(domain, clientid)

This function returns the identifier by which the calling program is known to the TCPIP virtual machine.

```
Socket('GetClientId')         ==   '0 AF_INET USERID1 myId'
```

SOCKET('GETDOMAINNAME')

Result: rc domainname

C socket call: getdomainname(name, namelen)

This function returns the name of the domain under which the processor running the program falls.

```
Socket('GetDomainName')       ==   '0 ZURICH.IBM.COM'
```

SOCKET('GETHOSTID')

Result: rc ipaddress

C socket call: gethostid()

This function returns the ipaddress for the current host which is the default home internet address.

```
Socket('GetHostId')           ==   '0 9.4.3.2'
```

SOCKET('GETHOSTNAME')

Result: rc hostname

C socket call: gethostname(name, namelen)

This function returns the name of the host processor on which the program is running.

```
Socket('GetHostName')          ==   '0 ZURLVM1'
```

SOCKET('GETPEERNAME',socketid)

Result: rc name

C socket call: getpeername(s, name, namelen)

This function returns the name of the peer connected to the given socket.

```
Socket('GetPeerName',6)        ==   '0 AF_INET 1234 128.228.1.2'
```

SOCKET('GETSOCKNAME',socketid)

Result: rc name

C socket call: getsockname(s, name, namelen)

This function returns the name to which the given socket was bound. Stream sockets are not assigned a name, until after a successful call to either the BIND, the CONNECT or the ACCEPT subfunction.

```
Socket('GetSockName',7)        ==   '0 AF_INET 5678 9.4.3.2'
```

SOCKET('GETHOSTBYADDR',ipaddress)

Result: rc fullhostname

C socket call: gethostbyaddr(addr, addrlen, domain)

This function tries to resolve the host name through a name server, if one is present.

```
Socket('GetHostByAddr','128.228.1.2')   ==   '0 CUNYVM.CUNY.EDU'
```

SOCKET('GETHOSTBYNAME',hostname|fullhostname)

Result: rc ipaddresslist

C socket call: gethostbyname(name)

This function tries to resolve the host name through a name server, if one is present. (It returns all ipaddresses for multi-home hosts each separated by a blank.)

```
Socket('GetHostByName','CUNYVM')            ==   '0 128.228.1.2'
Socket('GetHostByName','CUNYVM.CUNY.EDU')   ==   '0 128.228.1.2'
```

SOCKET('RESOLVE',ipaddress|hostname|fullhostname)

Result: rc ipaddress fullhostname

This function tries to resolve the host name through a name server, if one is present. (For multi-home hosts, only the default home internet address is returned.)

```
Socket('Resolve','128.228.1.2')       ==   '0 128.228.1.2 CUNYVM.CUNY.EDU'
Socket('Resolve','CUNYVM')            ==   '0 128.228.1.2 CUNYVM.CUNY.EDU'
Socket('Resolve','CUNYVM.CUNY.EDU')   ==   '0 128.228.1.2 CUNYVM.CUNY.EDU'
```

SOCKET('GETPROTOBYNAME',protocolname)

Result: rc protocolnumber

C socket call: getprotobyname(name)

This function returns the number of a network protocol specified by its name.

```
Socket('GetProtoByName','TCP')              ==   '0 6'
```

SOCKET('GETPROTOBYNUMBER',protocolnumber)

Result: rc protocolname

C socket call: getprotobynumber(name)

This function returns the name of a network protocol specified by its number.

```
Socket('GetProtoByNumber',6)                ==   '0 TCP'
```

SOCKET('GETSERVBYNAME',servicename,[protocolname])

Result: rc servicename portid protocolname

C socket call: getservbyname(name, proto)

This function returns the name of a service, the port and the name of the network protocol.

```
Socket('GetServByName','ftp','tcp')       ==   '0 FTP 21 TCP'
```

SOCKET('GETSERVBYPORT',portid,[protocolname])

Result: rc servicename portid protocolname

C socket call: getservbyport(name, proto)

This function returns the name of a service, the port and the name of the network protocol.

```
Socket('GetServByPort',21,'tcp')          ==   '0 FTP 21 TCP'
```

3.5 Auxiliary Subfunctions

A last group of subfunctions allows the REXX program to obtain the version number of the REXX/SOCKETS function package and other information. There are subfunctions to determine and to set socket options or the socket mode. The socket options are options associated with a socket and influence the behavior of a socket. There are also auxiliary subfunctions to determine the network configuration.

SOCKET('VERSION')

Result: rc 'REXX/SOCKETS' version date

The function returns the name ('REXX/SOCKETS'), the version number and the version date of the REXX/SOCKETS function package.

```
Socket('Version')            ==   '0 REXX/SOCKETS 2.00 16 December 1992'
```

SOCKET('SELECT','READ' socketidlist 'WRITE' socketidlist 'EXCEPTION' socketidlist,[timeout])

Result: rc count 'READ' socketidlist 'WRITE' socketidlist 'EXCEPTION' socketidlist

C socket call: select(nfds, readfds, writefds, exceptfds, timeout)

This function allows waiting on socket related events. (Note that the suggested method for waiting on socket related events is through the REXX built-in function WAIT. There are differences between the two methods. The SELECT subfunction always returns all socketids enabled while the WAIT function only returns one. However, the WAIT function tries to return socketids with fair scheduling.)

A close on the other side of a connection is not reported as an exception, but as a READ event which returns zero bytes of data. When the CONNECT subfunction is called with a socket in non-blocking mode, it terminates with a return code EINPROGRESS, and the completion of the connection setup gets reported as a WRITE event on the socket. When the ACCEPT subfunction is called with a socket in in non-blocking mode, it terminates with a return code EWOULDBLOCK. However, the availability of a connection request gets reported as a READ event on the original socket, and the ACCEPT subfunction should only be called then.

```
Socket('Select','Read 5 Write Exception',10)   ==   '0 1 READ 5 WRITE EXCEPTION'
```

SOCKET('GETSOCKOPT',socketid,level,optname)

Result: rc optvalue

C socket call: getsockopt(s, level, optname, optval, optlen)

This function gets options associated with a socket. Options can exist at multiple protocol levels. They are always present at the highest socket level.

```
Socket('GetSockOpt',5,'Sol_Socket','So_ASCII')       ==   '0 On STANDARD'
Socket('GetSockOpt',5,'Sol_Socket','So_Broadcast')   ==   '0 On'
Socket('GetSockOpt',5,'Sol_Socket','So_Error')       ==   '0 0'
Socket('GetSockOpt',5,'Sol_Socket','So_Linger')      ==   '0 On 60'
Socket('GetSockOpt',5,'Sol_Socket','So_Sndbuf')      ==   '0 8192'
Socket('GetSockOpt',5,'Sol_Socket','So_Type')        ==   '0 SOCK_STREAM'
Socket('GetSockOpt',5,'IPproto_TCP','TCP_NoDelay')   ==   '0 Off'
```

SOCKET('SETSOCKOPT',socketid,level,optname,optvalue)

Result: rc

C socket call: setsockopt(s, level, optname, optval, optlen)

This function sets options associated with a socket. Options can exist at multiple protocol levels. They are always present at the highest socket level.

```
Socket('SetSockOpt',5,'Sol_Socket','So_ASCII','On')       ==   '0'
Socket('SetSockOpt',5,'Sol_Socket','So_ASCII','On XTAB')  ==   '0'
Socket('SetSockOpt',5,'Sol_Socket','So_Broadcast','On')   ==   '0'
Socket('SetSockOpt',5,'Sol_Socket','So_Linger','On 60')   ==   '0'
Socket('SetSockOpt',5,'IPproto_TCP','TCP_NoDelay','On')   ==   '0'
```

SOCKET('FCNTL',socketid,fcmd,[fvalue])

Result: rc [fvalue]

C socket call: fcntl(s, cmd, data)

This function allows a REXX program to control the operating characteristics of a socket. It allows setting a socket into blocking or non-blocking mode.

```
Socket('Fcntl',5,'F_SETFL','NON-BLOCKING')   ==   '0'
Socket('Fcntl',5,'F_GETFL')                  ==   '0 NON-BLOCKING'
```

SOCKET('IOCTL',socketid,icmd,[ivalue])

Result: rc [ivalue]

C socket call: ioctl(s, cmd, data)

This function allows a REXX program to control the operating characteristics of a socket. It allows setting a socket into blocking or non-blocking mode, but also allows it to determine or modify other operating characteristics.

```
Socket('Ioctl',5,'FionBio','On')         ==  '0'
Socket('Ioctl',5,'FionRead')             ==  '0 8192'
Socket('Ioctl',5,'SiocAtMark')           ==  '0 No'
Socket('Ioctl',5,'SiocGifConf',2)        ==  '0 TR1 AF_INET 0 9.4.3.2 TR2 AF_INET 0 9.4.3.3'
Socket('Ioctl',5,'SiocGifAddr','TR1')    ==  '0 TR1 AF_INET 0 9.4.3.2'
Socket('Ioctl',5,'SiocGifFlags','TR1')   ==  '0 TR1 0051 IFF_UP IFF_POINTOPOINT IFF_RUNNING'
Socket('Ioctl',5,'SiocGifMetric','TR1')  ==  '0 TR1 0'
Socket('Ioctl',5,'SiocGifNetMask','TR1') ==  '0 TR1 AF_INET 0 255.255.255.0'
```

4.0 REXX/SOCKETS Sample Programs

The RSCLIENT EXEC and RSSERVER EXEC sample programs are discussed. They are REXX programs showing the use of the functions provided by REXX/SOCKETS. The first program is the client code, and the second program is the server code for a simple application.

Before the client program can be started, the server program must run in another virtual machine. The two programs may run on different hosts, but in this case, the internet address of the host running the server program must be entered with the command starting the client program, and the two hosts must be connected via TCP/IP.

4.1 The REXX-EXEC RSCLIENT

The client sample program connects to the server sample program and receives data which is displayed on the screen. It uses its sockets in the blocking mode.

```
/*- RSCLIENT -- Example demonstrating the usage of REXX/SOCKETS ------*/
trace o
signal on syntax

/* Set error code values                                              */
ecpref = 'RXS'
ecname = 'CLI'
initialized = 0

parse arg argstring
argstring = strip(argstring)
if substr(argstring,1,1) = '?' then do
  say 'RSSERVER and RSCLIENT  are a pair of programs which provide an'
  say 'example of how to use REXX/SOCKETS to implement a service. The'
  say 'server must be started before the clients get started.        '
  say '                                                              '
  say 'The RSSERVER program runs in its own dedicated virtual machine'
  say 'and returns a number of data lines as requested to the client.'
  say 'It is started with the command:                               '
  say '    RSSERVER                                                  '
  say 'and terminated with the command:                              '
  say '    EXIT                                                      '
  say '                                                              '
  say 'The RSCLIENT program is used  to request a number of arbitrary'
  say 'data lines  from the  server  and can be run  concurrently any'
  say 'number  of times  by different  clients  until  the server  is'
  say 'terminated.  It is started with the command:                  '
  say '    RSCLIENT number <server>                                  '
  say 'where "number" is the number of data lines to be requested and'
  say '"server" is the ipaddress of the service virtual machine. (The'
  say 'default ipaddress is the one of the host  on which RSCLIENT is'
  say 'running, assuming that RSSERVER runs on the same host.)       '
  exit 100
end

/* Split arguments into parameters and options                        */
parse upper var argstring parameters '(' options ')' rest

/* Parse the parameters                                               */
parse var parameters lines server rest
if ¬datatype(lines,'W') then call error 'E', 24, 'Invalid number'
lines = lines + 0
if rest¬='' then call error 'E', 24, 'Invalid parameters'
```

```
/* Parse the options                                                 */
do forever
  parse var options token options
  select
    when token='' then leave
    otherwise call error 'E', 20, 'Invalid option "'token'"'
  end
end

/* Initialize control information                                    */
port = '1952'               /* The port used by the server           */
address command 'IDENTIFY ( LIFO'
parse upper pull userid . locnode .

/* Initialize                                                        */
call Socket 'Initialize', 'RSCLIENT'
if src=0 then initialized = 1
else call error 'E', 200, 'Unable to initialize RXSOCKET MODULE'
if server='' then do
  server = Socket('GetHostId')
  if src¬=0 then call error 'E', 200, 'Cannot get the local ipaddress'
end
ipaddress = server

/* Initialize for receiving lines sent by the server                 */
s = Socket('Socket')
if src¬=0 then call error 'E', 32, 'SOCKET(SOCKET) rc='src
call Socket 'Connect', s, 'AF_INET' port ipaddress
if src¬=0 then call error 'E', 32, 'SOCKET(CONNECT) rc='src
call Socket 'Write', s, locnode userid lines
if src¬=0 then call error 'E', 32, 'SOCKET(WRITE) rc='src

/* Wait for lines sent by the server                                 */
dataline = ''
num = 0
do forever

  /* Receive a line and display it                                   */
  parse value Socket('Read', s) with len newline
  if src¬=0 | len<=0'' then leave
  dataline = dataline || newline
  do forever
    if pos('15'x,dataline)=0 then leave
    parse var dataline nextline '15'x dataline
    num = num + 1
    say right(num,5)':' nextline
  end
end

/* Terminate and exit                                                */
call Socket 'Terminate'
exit 0

/* Calling the real SOCKET function                                  */
socket: procedure expose initialized src
  a0 = arg(1)
  a1 = arg(2)
  a2 = arg(3)
  a3 = arg(4)
  a4 = arg(5)
  a5 = arg(6)
  a6 = arg(7)
```

```
  a7 = arg(8)
  a8 = arg(9)
  a9 = arg(10)
  parse value 'SOCKET'(a0,a1,a2,a3,a4,a5,a6,a7,a8,a9) with src res
return res

/* Syntax error routine                                          */
syntax:
  call error 'E', rc, '==> REXX Error No.' 20000+rc
return

/* Error message and exit routine                                */
error: procedure expose ecpref ecname initialized
  type = arg(1)
  retc = arg(2)
  text = arg(3)
  ecretc = right(retc,3,'0')
  ectype = translate(type)
  ecfull = ecpref || ecname || ecretc || ectype
  address command 'EXECIO 1 EMSG (CASE M STRING' ecfull text
  if type¬='E' then return
  if initialized then do
    parse value Socket('SocketSetStatus') with . status severreason
    if status¬='Connected' then do
      say 'The status of the socket set is' status severreason
    end
    call Socket 'Terminate'
  end
exit retc
```

After parsing and testing the input arguments, the RSCLIENT EXEC obtains a socket set with the INITIALIZE subfunction and a socket with the SOCKET subfunction, connects to the server and writes the userid, the nodeid and the number of lines requested on the connection to the server. Afterward, it reads data in a loop and displays it on the screen until the data length is zero indicating that the server has closed the connection. When an error occurs, the client program displays the return code, determines the status of the socket set and terminates it.

The server adds '15'x at the end of each record, and the client uses this character to determine where a new record starts. When the connection gets closed abnormally, the client will not display partially received records.

4.2 The REXX-EXEC RSSERVER

The server sample program waits for connect requests from clients, accepts them and sends data. It can handle multiple client requests in parallel. It uses its sockets in the non-blocking mode.

```
/*- RSSERVER -- Example demonstrating the usage of REXX/SOCKETS ------*/
trace o
signal on syntax

/* Set error code values                                         */
ecpref = 'RXS'
ecname = 'SER'
initialized = 0

parse arg argstring
argstring = strip(argstring)
if substr(argstring,1,1) = '?' then do
  say 'RSSERVER and RSCLIENT  are a pair of programs which provide an'
  say 'example of how to use REXX/SOCKETS to implement a service. The'
  say 'server must be started before the clients get started.       '
```

```
  say '                                                              '
  say 'The RSSERVER program runs in its own dedicated virtual machine'
  say 'and returns a number of data lines as requested to the client.'
  say 'It is started with the command:                               '
  say '    RSSERVER                                                  '
  say 'and terminated with the command:                              '
  say '    EXIT                                                      '
  say '                                                              '
  say 'The RSCLIENT program is used  to request a number of arbitrary'
  say 'data lines  from the  server  and can be run  concurrently any'
  say 'number  of times  by different  clients  until  the server  is'
  say 'terminated.  It is started with the command:                  '
  say '    RSCLIENT number <server>                                  '
  say 'where "number" is the number of data lines to be requested and'
  say '"server" is the ipaddress of the service virtual machine. (The'
  say 'default ipaddress is the one of the host  on which RSCLIENT is'
  say 'running, assuming that RSSERVER runs on the same host.)       '
  exit 100
end

/* Split arguments into parameters and options                       */
parse upper var argstring parameters '(' options ')' rest

/* Parse the parameters                                              */
parse var parameters rest
if rest¬='' then call error 'E', 24, 'Invalid parameters specified'

/* Parse the options                                                 */
do forever
  parse var options token options
  select
    when token='' then leave
    otherwise call error 'E', 20, 'Invalid option "'token'"'
  end
end

/* Initialize control information                                    */
port = '1952'               /* The port used for the service        */

/* Initialize                                                        */
say 'RSSERVER: Initializing'
call Socket 'Initialize', 'RSSERVER'
if src=0 then initialized = 1
else call error 'E', 200, 'Unable to initialize RXSOCKET MODULE'
ipaddress = Socket('GetHostId')
if src¬=0 then call error 'E', 200, 'Unable to get the local ipaddress'
address command 'REXXWAIT TEST'
if rc¬=0 then call error 'E', 200, 'Unable to load REXXWAIT MODULE'
call ResetValue 'All'
say 'RSSERVER: Initialized: ipaddress='ipaddress 'port='port

/* Initialize for accepting connection requests                      */
s = Socket('Socket')
if src¬=0 then call error 'E', 32, 'SOCKET(SOCKET) rc='src
call Socket 'Bind', s, 'AF_INET' port ipaddress
if src¬=0 then call error 'E', 32, 'SOCKET(BIND) rc='src
call Socket 'Listen', s, 10
if src¬=0 then call error 'E', 32, 'SOCKET(LISTEN) rc='src
call SetValue 'Socket' s 'Non-Blocking'
if wrc¬=0 then call error 'E', 36, 'Cannot set mode of socket' s

/* Wait for new connections and send lines                           */
linecount. = 0
```

```
wlist = ''
do forever

  /* Wait for an event                                              */
  if wlist¬='' then sockevtlist = 'Write'wlist 'Read *'
  else sockevtlist = 'Read *'
  if length(sockevtlist)>200 then sockevtlist = 'Write * Read *'
  event = Wait('Socket' sockevtlist,'Cons')
  if wrc¬=0 then call error 'E', 36, 'WAIT() rc='wrc
  parse upper var event event rest
  select

    /* Leave program                                                */
    when event='CONS' then do
      if rest='EXIT' then leave
      else say 'Enter "EXIT" to leave'
    end

    /* Accept connections from clients, receive and send messages   */
    when event='SOCKET' then do
      parse var rest keyword ts .

      /* Accept new connections from clients                        */
      if keyword='READ' & ts=s then do
        nsn = Socket('Accept',s)
        if src=0 then do
          parse var nsn ns . np nia .
          say 'RSSERVER: Connected by' nia 'on port' np 'and socket' ns
        end
      end

      /* Get nodeid, userid and number of lines to be sent          */
      if keyword='READ' & ts¬=s then do
        parse value Socket('Recv',ts) with len nid uid count .
        if src=0 & len>0 & datatype(count,'W') then do
          if count<0 then count = 0
          if count>5000 then count = 5000
          ra = 'by' uid 'at' nid
          say 'RSSERVER: Request for' count 'lines on socket' ts ra
          linecount.ts = linecount.ts + count
          call addsock(ts)
        end
        else do
          call Socket 'Close',ts
          linecount.ts = 0
          call delsock(ts)
          say 'RSSERVER: Disconnected socket' ts
        end
      end

      /* Get nodeid, userid and number of lines to be sent          */
      if keyword='WRITE' then do
        if linecount.ts>0 then do
          num = random(1,sourceline())      /* Return random-selected */
          msg = sourceline(num) || '15'x      /* line of this program */
          call Socket 'Send',ts,msg
          if src=0 then linecount.ts = linecount.ts - 1
          else linecount.ts = 0
        end
        else do
          call Socket 'Close',ts
          linecount.ts = 0
          call delsock(ts)
```

```
          say 'RSSERVER: Disconnected socket' ts
        end
      end

    end

    /* Unknown event (should not occur)                              */
    otherwise nop
  end
end

/* Terminate and exit                                                */
call Socket 'Terminate'
say 'RSSERVER: Terminated'
exit 0

/* Procedure to add a socket to the write socket list                */
addsock: procedure expose wlist
  s = arg(1)
  p = wordpos(s,wlist)
  if p=0 then wlist = wlist s
return

/* Procedure to del a socket from the write socket list              */
delsock: procedure expose wlist
  s = arg(1)
  p = wordpos(s,wlist)
  if p>0 then do
    templist = ''
    do i=1 to words(wlist)
      if i¬=p then templist = templist word(wlist,i)
    end
    wlist = templist
  end
return

/* Calling the real SOCKET function                                  */
socket: procedure expose initialized src
  a0 = arg(1)
  a1 = arg(2)
  a2 = arg(3)
  a3 = arg(4)
  a4 = arg(5)
  a5 = arg(6)
  a6 = arg(7)
  a7 = arg(8)
  a8 = arg(9)
  a9 = arg(10)
  parse value 'SOCKET'(a0,a1,a2,a3,a4,a5,a6,a7,a8,a9) with src res
return res

/* Calling the real WAIT function                                   */
wait: procedure expose initialized wrc
  a0 = arg(1)
  a1 = arg(2)
  a2 = arg(3)
  a3 = arg(4)
  a4 = arg(5)
  a5 = arg(6)
  a6 = arg(7)
  a7 = arg(8)
  a8 = arg(9)
  a9 = arg(10)
```

```
  parse value 'WAIT'(a0,a1,a2,a3,a4,a5,a6,a7,a8,a9) with wrc res
return res

/* Calling the real SETVALUE function                              */
setvalue: procedure expose initialized wrc
  a0 = arg(1)
  parse value 'SETVALUE'(a0) with wrc res
return res

/* Calling the real QUERYVALUE function                            */
queryvalue: procedure expose initialized wrc
  a0 = arg(1)
  parse value 'QUERYVALUE'(a0) with wrc res
return res

/* Calling the real RESETVALUE function                            */
resetvalue: procedure expose initialized wrc
  a0 = arg(1)
  parse value 'RESETVALUE'(a0) with wrc res
return res

/* Syntax error routine                                             */
syntax:
  call error 'E', rc, '==> REXX Error No.' 20000+rc
return

/* Error message and exit routine                                   */
error: procedure expose ecpref ecname initialized
  type = arg(1)
  retc = arg(2)
  text = arg(3)
  ecretc = right(retc,3,'0')
  ectype = translate(type)
  ecfull = ecpref || ecname || ecretc || ectype
  address command 'EXECIO 1 EMSG (CASE M STRING' ecfull text
  if type¬='E' then return
  if initialized then do
    parse value Socket('SocketSetStatus') with . status severreason
    if status¬='Connected' then do
      say 'The status of the socket set is' status severreason
    end
    call Socket 'Terminate'
  end
exit retc
```

The server sets up a socket for accepting connection requests by clients and waits in a loop on events reported by the REXX function WAIT. When 'exit' is entered on the keyboard, it terminates even if connections are still open. When a socket event occurs, it processes it accordingly. A READ event can occur on the original socket for accepting connection requests and on sockets for accepted socket requests. A WRITE event can only occur on sockets for accepted socket requests.

A READ event on the original socket for connection requests means that a connection request from a client occurred. READ events on other sockets indicate that there is either data to receive or that the client closed the socket. WRITE events indicate that the server can send more data. The server program only sends one line of data in response to a WRITE event.

Note that the server program keeps a list of sockets to which it wants to write. It keeps this list in order to avoid unwanted socket events. The TCP/IP protocol was not designed for one single-threaded program communicating on many different sockets, but for multi-threaded applications where one thread processes only events from one single socket. Therefore, it is not very easy to write robust server applications in a single-threaded environment.

Index

S

T

V

W