26 Sockets Module (sockets.hhf)

The HLA Standard Library provides two mechanisms that support network communications via sockets: a low-level library (whose functions appear in the *sock* namespace) and a higher-level socket class that makes it almost trivial to create client/server applications. The low-level sock library provides a thin veneer over the low-level OS calls (largely to make the calls portable across all OSes). Those wanting to write networking applications using traditional BSD-style socket calls should consider using the *sock* module rather the low-level OS calls.

Note: HLA also provides access to certain low-level OS API primitives that directly access the native OS' networking code. Please see the discussions of the native OS functions for more details on such low-level network access.

26.1 The SOCK Module

The *sock* namespace in the HLA standard library contains a fair set of functions that behave largely like the original BSD socket functions (that most OSes support). These functions, by and large, work just like the low-level socket calls that most OSes support. Largely, these function smooth out some data type differences between various OSes (e.g., the definition of the *fd_set_t* data type) so that function calls (and the data types of their arguments) are consistent across all OSes, regardless of the underlying data types a particular OS might use.

26.2 Socket Initialization and Cleanup

Before using any socket functions, you must first call the *sock.socketInit* function to initialize the socket library. When you are done using sockets in an application, you must call *sock.socketCleanup* to free system resources and shut down the socket system. These are HLA standard library functions that are not particularly related to the underlying OS socket API. You must call these function before any other socket operations and when you're done using the sockets.

sock.socketInit;

This function initializes the socket library for the HLA standard library. You must call this function exactly one in any application that makes other low-level socket calls (before making those calls). Note that this function may increment an internal reference counter, so make sure you make a corresponding call to *sock.socketCleanup* before your application terminates.

```
sock.socketInit();
```

sock.socketCleanup;

This function undoes the effects of *sock.socketInit* and frees up any system resources reserved by *sock.socketInit*. You must call it exactly once when your application is done using sockets.

```
sock.socketCleanup();
```

26.3 Generic Socket Functions

A few functions in the *sock* namespace provide conversions on socket metadata. These functions include *sock.a_adrsToStr, sock.adrsToStr,* and *sock.strToAdrs*.

```
sock.a_adrsToStr( a:bigEndianDW );  @returns( "eax" );
sock.adrsToStr( a:bigEndianDW; s:string );
```

These functions take a dword parameter *in network byte order (big endian form)* and convert the address to the form "ddd.ddd.ddd.ddd" (where each "ddd" represents exactly three decimal digits). The sock.a_adrsToStr function allocates storage for the 15-character string on the heap and returns a pointer to the new string in the EAX register. The *sock.adrsToStr* function stores the string result into the string object passed as the second argument (*s*). The *sock.adrsToStr* function will raise an exception if *s* doesn't have sufficient storage to hold a 15-character strong.

```
    sock.a_adrsToStr( $01020304 );       // Produces "004.003.002.001"
    sock.adrsToStr( $04030201, s );      // Stores "001.002.003.004" into s
```

sock.strToAdrs(s:string); @returns("eax");

This function take a string parameter of the form "ddd.ddd.ddd.ddd" (where each "ddd" represents exactly three decimal digits) and converts it to a double word *in network byte order (big endian form)* and returns this value in the EAX register. This function raises an exception of there is a conversion error.

```
    sock.strToAdrs( "001.002.003.004" );// Produces $04030201 in EAX
```

26.4 Low-Level BSD-Style Socket Functions

The functions in this category correspond to the Berkeley (BSD) sockets functions. You should not assume that the data types passed to these functions are identical to those in BSD sockets. Some data types have been changed in order to make the HLA sockets module compatible across all the OSes that the HLA stdlib supports. It is the responsibility of all of these functions to do any necessary conversion prior to calling the OS-level socket API functions.

This documentation will not describe the functionality for each of these functions. See a discussion of the BSD sockets API (on the internet) for more details. If you are unfamiliar with low-level socket calls, you should either use the HLA standard library socket classes (which simplify network programming) or pick up a good book on making networking calls via the BSD sockets API. You can also look at the source code for the socket server and client classes in the HLA standard library for examples of these calls.

```
sock.accept
(
            s           :dword;
    var     addr        :sock.sockaddr;
    var     addrlen     :sock.socklen_t
);
```

The argument s is a socket that has been created with *sock.socket*, bound to an address with *sock.bind*, and is listening for connections after a *sock.listen* call. The *sock.accept* function extracts the first connection request on the queue of pending connections, creates a new socket with the same properties of *s* and allocates a new file descriptor for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, *sock.accept* blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, *sock.accept* returns an error as described below. The accepted socket may not be used to accept more connections. The original socket *s* remains open.

The argument *addr* is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the domain in which the communication is occurring. The *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*; on return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with *sock.SOCK_STREAM*.

It is possible to *sock.select* a socket for the purposes of doing a sock.accept by selecting it for read.

This function raises an *ex.SocketError* exception if any error occurs.

```
sock.bind
(
                sockfd          :dword;
    var         addr            :sockaddr;
                addrlen         :socklen_t
);
```

sock.bind assigns a name (that is, an IP address) to an unnamed socket. When a socket is created with *sock.socket* it exists in a name space (address family) but has no name (IP address) assigned. *sock.bind* requests that name (IP address) be assigned to the socket.

This function raises an *ex.SocketError* exception if any error occurs.

```
sock.connect
(
                s                       :dword;
    var         serv_addr               :sockaddr;
                addrlen                 :socklen_t
);
```

The parameter *s* is a socket. If it is of type *sock.SOCK_DGRAM*, this call specifies the peer with which the socket is to be associated; this address is that to which datagrams are to be sent, and the only address from which datagrams are to be received. If the socket is of type *sock.SOCK_STREAM*, this call attempts to make a connection to another socket. The other socket is specified by name (i.e., IP address), which is an address in the communications space of the socket. Each communications space interprets the name parameter in its own way. Generally, stream sockets may successfully connect only once; datagram sockets may use *sock.connect* multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address or an address with the address family set to *sock.AF_UNSPEC* .

This function raises an *ex.SocketError* exception if any error occurs.

```
sock.close( s:dowrd );
```

sock.close closes the socket whose handle is specified by the *s* descriptor passed as a parameter.

This function raises an *ex.SocketError* exception if any error occurs.

```
sock.listen
(
                s       :dword;
    backlog             :dword
);
```

To accept connections, a socket is first created with *sock.socket*, a willingness to accept incoming connections and a queue limit for incoming connections are specified with *sock.listen*, and then the connections are accepted with *sock.accept*. The *sock.listen* call applies only to sockets of type *sock.SOCK_STREAM* or *sock.SOCK_SEQPACKET*.

The backlog parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full the client may receive an error, or, if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.

This function raises an *ex.SocketError* exception if any error occurs.

```
sock.recv
(
                s           :dword;
    var         buf         :var;
                len         :dword;
                flags       :dword
);  @returns( "eax" );
```

```
sock.recvfrom
(
                s                   :dword;
    var         buf                 :var;
                len                 :dword;
                flags               :dword;
    var         from                :sockaddr;
```

```
    var         fromlen         :socklen_t
);  @returns( "eax" );
```

sock.recvfrom is used to receive messages from a socket, and may be used to receive data on a socket whether or not it is connection oriented.

If from is non-nil, and the socket is not connection-oriented, the source address of the message is filled in. Fromlen is a value-result parameter, initialized to the size of the buffer associated with from, and modified on return to indicate the actual size of the address stored there.

The *sock.recv* call is normally used only on a connected socket (see sock.connect) and is identical to *sock.recvfrom* with a NJLL *from* parameter.

On successful completion, both routines return the number of message bytes read in the EAX register. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see *sock.socket*). Note that if these functions read fewer bytes from the socket than specified by the *len* parameter, these functions do not raise an end-of-file exception (as is common for the socket class input routines).

The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested; this behavior is affected by the socket-level options *sock.SO_RCVLOWAT* and *sock.SO_RCVTIMEO* described in *sock.getsockopt*.

The *sock.select* call may be used to determine when more data arrives.

The flags argument to a *sock.recv* call is formed by or'ing one or more of the values:

sock.MSG_OOB process out-of-band data

sock.MSG_PEEK peek at incoming message

sock.MSG_WAITALL wait for full request or error

The *sock.MSG_OOB* flag requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at the head of the normal data queue, and thus this flag cannot be used with such protocols. The *sock.MSG_PEEK* flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data. The *sock.MSG_WAITALL* flag requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if a signal is caught, an error or disconnect occurs, or the next data to be received is of a different type than that returned.

This function raises an *ex.SocketError* exception if any error occurs.

```
sock.select
(
                nfds        :dword;
    var         readSet     :sock.fd_set_t;
    var         writeSet    :sock.fd_set_t;
    var         exceptSet   :sock.fd_set_t;
    var         timeout     :sock.timeval
);  @returns( "eax" );
```

The *sock.select* function examines the I/O descriptor sets whose addresses are passed in *readfds, writefds*, and *exceptfds* to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first *nfds* descriptors are checked in each set; i.e., the descriptors from 0 through *nfds-1* in the descriptor sets are examined. (Example: If you have set two file descriptors "4"and "17", *nfds* should not be "2", but rather "17 + 1" or "18".) On return, *sock.select* replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation.

Select() returns the total number of ready descriptors in all the sets in the EAX register.

Note that the *sock.fd_set_t* data type may not be equivalent to the *fd_set* data type used by the underlying operating system. In particular, you should not assume that this is a bit map. The function may choose to ignore the *nfds* parameter (which is present for historical reasons), but you should still set it up properly.

If timeout is a non-nil pointer, it specifies a maximum interval to wait for the selection to complete. If timeout is a NULL pointer, the select blocks indefinitely. To effect a poll, the timeout argument should be non-NULL, pointing to a zero-valued *sock.timeval* structure. Timeout is not changed by *sock.select*, and may be reused on subsequent calls, however it is good style to re-initialize it before each invocation of *sock.select*.

Any of readfds, writefds, and exceptfds may be given as nil pointers ifno descriptors are of interest.

The *sock.select* function returns the number of ready descriptors that are contained in the descriptor sets. If the time limit expires, *sock.select* returns 0. If *sock.select* raises an exception, the descriptor sets will be unmodified.

This function raises an *ex.SocketError* exception if any error occurs.

```
sock.send
(
                    s           :dword;
     var            buf         :var;
                    len         :dword;
                    flags       :dword      // MSG_* constants
);   @returns( "eax" );


sock.sendto
(
                    s           :dword;
     var            buf         :var;
                    len         :dword;
                    flags       :dword             // MSG_* constants
     var            _to         :sock.sockaddr;
                    tolen       :sock.socklen_t
);   @returns( "eax" );
```

The *sock.end* and *sock.sendto* functions are used to transmit a message to another

socket. The *sock.send* function may be used only when the socket is in a connected state, while sock.sendto may be used at any time.

The address of the target is given by to with *tolen* specifying its size. The length of the message is given by *len*. If the message is too long to pass atomically through the underlying protocol, the function raises an *ex.SocketError* exception and the message is not transmitted.

No indication of failure to deliver is implicit in a *sock.send* call.

If no messages space is available at the socket to hold the message to be transmitted, then *sock.send* normally blocks, unless the socket has been placed in non-blocking I/O mode. The *sock.select* call may be used to determine when it is possible to send more data.

The flags parameter may include one or more of the following:

```
sock.MSG_OOB        /* process out-of-band data */
sock.MSG_DONTROUTE/* bypass routing, use direct interface */
```

The flag *sock.MSG_OOB* is used to send ``out-of-band'' data on sockets that support this notion (e.g. sock.SOCK_STREAM); the underlying protocol must also support ``out-of-band'' data. *sock.MSG_DONTROUTE* is usually used only by diagnostic or routing programs.

The call returns the number of characters sent in the EAX register.

This function raises an *ex.SocketError* exception if any error occurs.

```
sock.socket(int domain, int type, int protocol);@returns( "eax" );
```

sock.socket creates an endpoint for communication and returns a descriptor. The domain parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. These families are defined in the include file sockets.hhf.

The currently understood formats are

```
sock.AF_UNIX        (UNIX internal protocols),
sock.AF_INET        (ARPA Internet protocols),
```

The socket has the indicated type, which specifies the semantics of communication. Currently defined types are:

```
sock.SOCK_STREAM
sock.SOCK_DGRAM
```

A *sock.SOCK_STREAM* type provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism may be supported. A *sock.SOCK_DGRAM* socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length).

The protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the communication domain in which communication is to take place.

Sockets of type *sock.SOCK_STREAM* are full-duplex byte streams, similar to pipes. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a *sock.connect* call. Once connected, data may be transferred using some variant of the *sock.send* and *sock.recv* calls. When a session has been completed a sock.close may be performed. Out-of-band data may also be transmitted as described in *sock.send* and received as described in sock.recv.

The communications protocols used to implement a *sock.SOCK_STREAM* stream insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error by raising an *ex.SocketError* exception. The protocols optionally keep sockets ``warm'' by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (e.g. 5 minutes). An ex.SocketError exception is raised if a process sends on a broken stream; this causes naive processes, which do not handle the exceptoin, to exit.

The operation of sockets is controlled by socket level options. *sock.setsockopt* and *sock.getsockopt* are used to set and get options, respectively.

If the call is successful, the return value (in EAX) is a descriptor referencing the socket.

This function raises an *ex.SocketError* exception if any error occurs.

```
sock.setsockopt
(
                    s           :dword;
                    level       :dword;
                    optname     :dword;
    var             optval      :var;
                    optlen      :socklen_t
);   @returns( "eax" );


sock.getsockopt
(
                    s           :dword;
                    level       :dword;
                    optname     :dword;
    var             optval      :var;
                    optlen      :socklen_t
);   @returns( "eax" );
```

sock.getsockopt and *sock.setsockopt* manipulate the options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost ``socket'' level.

When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the socket level, level is specified as *sock.SOL_SOCKET*. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, level should be set to the protocol number of TCP.

The parameters *optval* and *optlen* are used to access option values for *sock.setsockopt*. For *sock.getsockopt* they identify a buffer in which the value for the requested option(s) are to be returned. For *sock.getsockopt*, *optlen* is a value-result parameter, initially containing the size of the buffer pointed to by *optval*, and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, *optval* may be NULL.

Optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file sockets.hhf contains definitions for socket level options, described below.

Most socket-level options utilize a dword parameter for *optval*. For *sock.setsockopt*, the parameter should be non-zero to enable a Boolean option, or zero if the option is to be disabled. *sock.SO_LINGER* uses a record *sock.linger* parameter, defined in socket.hhf, which specifies the desired state of the option and the linger interval (see below). *sock.SO_SNDTIMEO* and *sock.SO_RCVTIMEO* use a *sock. timeval* parameter, defined in sockets.hhf.

The following options are recognized at the socket level. Except as noted, each may be examined with *sock.getsockopt* and set with *sock.setsockopt*.

```
sock.SO_DEBUG       enables recording of debugging information
sock.SO_REUSEADDR   enables local address reuse
sock.SO_REUSEPORT   enables duplicate address and port bindings
sock.SO_KEEPALIVE   enables keep connections alive
sock.SO_DONTROUTE   enables routing bypass for outgoing messages
sock.SO_LINGER      linger on close if data present
sock.SO_BROADCAST   enables permission to transmit broadcast messages
sock.SO_OOBINLINE   enables reception of out-of-band data in band
sock.SO_SNDBUF      set buffer size for output
sock.SO_RCVBUF      set buffer size for input
sock.SO_SNDLOWAT    set minimum count for output
sock.SO_RCVLOWAT    set minimum count for input
sock.SO_SNDTIMEO    set timeout value for output
sock.SO_RCVTIMEO    set timeout value for input
sock.SO_TYPE        get the type of the socket (get only)
sock.SO_ERROR       get and clear error on the socket (get only)
sock.SO_NOSIGPIPE   do not generate SIGPIPE, instead return EPIPE
```

sock.SO_DEBUG enables debugging in the underlying protocol modules. *sock.SO_REUSEADDR* indicates that the rules used in validating addresses supplied in a *sock.bind* call should allow reuse of local addresses. *sock.SO_REUSEPORT* allows completely duplicate bindings by multiple processes if they all set *sock.SO_REUSEPORT* before binding the port. This option permits multiple instances of a program to each receive UDP/IP multicast or broadcast datagrams destined for the bound port. *sock.SO_KEEPALIVE* enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified via an *ex.SocketError* exception when attempting to send data. *sock.SO_DONTROUTE* indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

sock.SO_LINGER controls the action taken when unsent messages are queued on socket and a *sock.close* is performed. If the socket promises reliable delivery of data and *sock.SO_LINGER* is set, the system will block the process on the close attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the *sock.setsockopt* call when *sock.SO_LINGER* is requested). If *sock.SO_LINGER* is disabled and a close is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

The option *sock.SO_BROADCAST* requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the socket system. With protocols that support out-of-band data, the *sock.SO_OOBINLINE* option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with *sock.recv* without the *sock.MSG_OOB* flag. Some protocols always behave as if this option is set. *sock.SO_SNDBUF* and *sock.SO_RCVBUF* are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute limit on these values.

sock.SO_SNDLOWAT is an option to set the minimum count for output operations. Most output operations process all of the data supplied by the call, delivering data to the protocol for transmission and blocking as necessary for flow control. Nonblocking output operations will process as much data as permitted subject to flow control without blocking, but will process no data if flow control does not allow the smaller of the low water mark value or the entire request to be processed. A *sock.select* operation testing the ability to write to a socket will return true only if the low water mark amount could be processed. The default value for *sock.SO_SNDLOWAT* is set to a convenient size for network efficiency, often 1024. *sock.SO_RCVLOWAT* is an option to set the minimum count for input operations. In general, receive calls will block until any (non-zero) amount of data is received, then return with the smaller of the amount available or the amount requested. The default value for *sock.SO_RCVLOWAT* is 1. If *sock.SO_RCVLOWAT* is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value orthe requested amount. Receive calls may still return less than the low water mark if an error occurs, a signal is caught, or the type of datanext in the receive queue is different than that returned.

sock.SO_SNDTIMEO is an option to set a timeout value for output operations. It accepts a struct timeval parameter with the number of seconds and microseconds used to limit waits for output operations to complete. If a send operation has blocked for this much time, it returns with a partial count or raises an exception if no data were sent. In the current implementation, this timer is restarted each time additional data are delivered to the protocol, implying that the limit applies to output portions ranging in size from the low water mark to the high water mark foroutput. *sock.SO_RCVTIMEO* is an option to set a timeout value for input operations. It accepts a record *sock.timeval* parameter with the number of seconds and microseconds used to limit waits for input operations to complete. In the current implementation, this timer is restarted each time additional data are received by the protocol, and thus the limit is in effect an inactivity timer. If a receive operation has been blocked for this much time without receiving additional data, it returns with a short count or with the error EWOULDBLOCK if no data were received. The struct

The *timeval* parameter must represent a positive time interval otherwise *sock.setsockopt* raises an *ex.SocketError* exception.

Finally, *sock.SO_TYPE* and *sock.SO_ERROR* are options used only with *sock.getsockopt*. *sock.SO_TYPE* returns the type of the socket, such as *sock.SOCK_STREAM*; it is useful for servers that inherit sockets on startup. *sock.SO_ERROR* returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

This function raises an *ex.SocketError* exception if any error occurs.

`sock.setTimeout( s:dword; timeout: sock.timeval );`

sock.setTimeout sets the timeout period for both transmission and reception on the socket specified by the socket descriptor *s*. Note: this function is a convenience function that calls *sock.setsockopt* to set the timeout periods. There is no equivalent *gettimeout* function; call *sock.getsockopt* if you need to retrieve one of the timeout periods. If you need to set the receive timeout period independently of the send timeout period, you will need to call *sock.setsockopt* to achieve this.

This function raises an *ex.SocketError* exception if any error occurs.

`sock.gethostname( s:string );`

The *sock.gethostname* makes a copy of (one of) the host name string(s) and stores this into the string variable you pass as an argument. Some systems can have more than one host name. In such a case, *sock.gethostname* returns one of the names (arbitrary choice).

This function raises an *ex.SocketError* exception if any socket-based error occurs. This function raises an ex.StringOverflow error if the hostname string is too long to fit in the storage allocated for the *s* argument. It can raise other exceptions if the value of *s* is bad.

`sock.gethostbyname( s:string; var hstent:sock.hostent );`

The *sock.gethostbyname* function fills in a *sock.hostent* data structure you pass by reference with host information based on the host name you pass as a string (s) to the function. Here is the current definition of the sock.hostent data structure (note that this is subject to change over time, so always use the *sock.hostent* type rather than manually creating this data structure yourself):

```
hostent:record

    h_name        :zstring;
    h_aliases     :dword;
    h_addrtype    :sock.sa_family_t;
    padding0      :word;
    h_length      :word;
    padding1      :word;
    h_addr_list   :dword;

endrecord;
```

The *h_aliases* field is a pointer to a sequence of dword addresses, terminated by a NULL address, each of which points at a zstring containing an alternate name for the host. You must not modify this array and you must not modify the strings its entries point at.

The *h_addrtype* field contains the type of the address being returned. This is usually *sock.AF_INET*.

The *h_length* field contains the length, in bytes, of each address in the address list.

The *h_addr_list* is a pointer to an array of pointers to network addresses for the host. Note that these addresses are stored in network byte order (big endian) form. The list is terminated with a NULL pointer.

This function raises an *ex.SocketError* exception if any socket-based error occurs.

```
sock.gethostbyaddr
(
     var   addr         :var;
           len          :dword;
           _type        :dword;
     var   hstent:hostent
);
```

The *sock.gethostbyaddr* function fills in a *sock.hostent* data structure you pass by reference with host information based on the host whose address you pass to the function. The addr argument is the address of a network address data structure whose type is specified by the *type* argument (usually *sock.AFINET*) and the length of which is specified by the *len* argument. This function copies the host information to the hstent argument you pass by reference. The *h_name* field of the *sock.hostent* data structure will contain the primary name of the host.

This function raises an *ex.SocketError* exception if any socket-based error occurs.

```
sock.getpeername
(
           s            :dword;
     var   _name        :sock.sockaddr;
     var   namelen      :sock.socklen_t
);   @returns( "eax" );
```

The *sock.getpeername* function returns the IP address (the "name") of the peer connected to the socket specified by the *s* socket descriptor. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *_name*. On return it contains the actual size of the *_name* returned (in bytes). The name is truncated if the buffer provided is too small. Note that the term "name" here refers to an IP address, not the name of the peer machine. This confusion is unfortunate, but that's the way the BSD sockets system was designed.

This function raises an *ex.SocketError* exception if any socket-based error occurs.

```
sock.getsockname
(
           s            :dword;
     var   _name        :sock.sockaddr;
     var   namelen      :sock.socklen_t
);   @returns( "eax" );
```

The *sock.getsockname* function returns the IP address (the "name") of the socket machine specified by the *s* socket descriptor. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *_name*. On return it contains the actual size of the *_name* returned (in bytes). The name is truncated if the buffer provided is too small. Note that the term "name" here refers to an IP address, not the name of the peer machine. This confusion is unfortunate, but that's the way the BSD sockets system was designed.

This function raises an *ex.SocketError* exception if any socket-based error occurs.

```
sock.fd_zero( var fdset:sock.fd_set_t );
sock.fd_set( fd:dword; var fdset:sock.fd_set_t );
sock.fd_clr( fd:dword; var fdset:sock.fd_set_t );
sock.fd_isset( fd:dword; var fdset:sock.fd_set_t );@returns( "al" );
```

The *sock.fd_** functions manipulate "file descriptor sets" that the *sock.select* function uses. The *sock.fd_zero* function creates the empty set and stores this into the *fdset* argument that you pass as by reference to the function. The *sock.fd_set* function unions the file (socket) descriptor pass in *fd* into the set *fdset* that you pass by reference to the function. The *sock.fd_clr* function removes (if preset) the file descriptor you pass in the *fd* argument from the set *fd_set* that you pass by reference to the function. The *sock.fd_isset* function checks for set membership. That is, it checks to see if the file descriptor specified by *fd* is present in the set *fdset* that you pass

by reference to the function; this function returns true/false in the AL register to denote presence/absence of the file descriptor in the set.

Note: always use these functions to manipulate socket descriptor sets. Do not assume that the HLA stdlib data structures match that of the underlying OS (they don't). Do not assume that the current implementation will always be used in future versions of the HLA stdlib. By using these functions, you can avoid future problems.

This function raises an *ex.SocketError* exception if any error occurs.

26.5 Socket Classes

Warning: *Don't forget that HLA objects modify the values in the ESI and EDI registers whenever you call a class procedure, method, or iterator. Do not leave any important values in either of these registers when making calls to the socket object functions. If the use of ESI and EDI is a problem for you, you might consider using the sock module that does not suffer from this problem.*

The HLA Standard Library provides an object-oriented network socket access mechanism implemented via the *baseSocket_t, server_t, client_t, vBaesSocket_t, vServer_t,* and *vClient_t* classes. As is typical for classes appearing in the HLA Standard Library, you can create customized versions of the generic socket classes, selecting which class functions are procedures or methods. This lets you choose between efficient static linking and virtual (overload) method capability on a function by function basis. Unless otherwise specified, this document will use the terms *socket class, server class,* and *client class* to describe the generic socket classes rather than the specific instance of the these classes (which uses static linking for all functions).

The HLA Standard Library sockets module provide six predefined classes that simplify the use of sockets, particularly for client/server applications. There are three basic classes with two variants of each class (a static variant and a virtual variant). In HLA classes, there are three types of functions: (static) procedures, (dynamic) methods, and dynamic iterators. The only difference between a method and a procedure is how the program actually calls the function: the program calls procedures directly, it calls methods indirectly through an entry in the virtual method table (VMT). The system always calls object iterators indirectly through the VMT, so we will not consider them in this discussion. This section will discuss the impact of class procedures versus class methods in your programs.

Static procedure calls are very efficient, but you lose the benefits of inheritance and functional polymorphism when you define a function as a static procedure in a class. Methods, on the other hand, fully support polymorphic calls, but introduce some (in)efficiency issues. The following paragraphs describe some of the efficiency issues concerning the use of methods.

First of all, unlike static procedures, your program will link in all methods defined in your program *even if you don't explicitly call those methods*. Because the call is indirect, there really is no way for the assembler and linker to determine whether you've actually called the function, so it must assume that you do call it and links in the code for each method in the class. This can make your program a little larger because it may be including several socket class functions that you don't actually call.

The second efficiency issue concerning method calls is that they use the EDI register to make the indirect call (static procedure calls do not disturb the value in EDI). Therefore, you must ensure that EDI is free and available before making a virtual method call, or take the effort to preserve EDI's value across such a call.

A third, though extremely minor, efficiency issue concerning methods is that the class' VMT will need an extra entry in the virtual method table. As this is only four bytes per class (not per object), this probably isn't much of a concern.

The predefined *baseSocketClass_t* and *vBaseSocketClass, server_t and vServer_t, and client_t and vClient_t* classes differ in how they define the functions appearing in the class types. The non-virtual types (without the 'v' prefix) generally use static procedures for all functions, the *virtual* types (with the 'v' prefix) use methods for all class functions. Therefore, the non-virtual socket object types will make direct calls to all the functions (and only link in the procedures you actually call); however, the non-virtual socket objects do not support function polymorphism in derived classes. The virtual socket types do support polymorphism for all the class methods, but whenever you use these data types you will link in all the methods (even if you don't call them all) and calls to these methods will require the use of the EDI register.

It is important to understand that *baseSocketClass_t/vBaseSocketClass_t, server_t/vServer_t,* and *client_t/vClient_t* pairs are two separate types. Neither is derived from the other. Nor are the two types in each pair compatible with one another. You should take care not to confuse objects of these two types if you're using both types in the same program.

The *baseSocketClass_t* and *vBaseSocketClass_t* types are base types intended for creating derived types (e.g., *server_t/vServer_t* and *client_t/vClient_t*); you would not normally use these two types in your programs, instead, you would use some type derived from these base classes (such as *server_t/vServer_t, or client_t/vClient_t*).

26.6 A Quick Note

The following sections do not include sample code demonstrating the calling sequences for a couple of reasons:

For high level calls, the syntax depends on the object name and type.

Low-level calling sequences don't appear here because it doesn't really make sense to make a low-level object invocation.

These functions are really intended for use by programmers experienced with HLA's Object-oriented assembly facilities.

For the same reasons, there are no stack diagrams for these function calls. If you want more information on making calls to HLA class methods and procedures, please consult the HLA documentation.

In the following function descriptions, the symbol <object> is used to specify a socket class object or a pointer to a socket class object. Wherever this document uses the name "socket", you may substitute (as appropriate) *server_t*, *vServer_t, client_t, vClient_t,* or any other socket class you've created by subclassing *baseSocketClass_t* or *vBaseSocketClass_t*.

26.7 Client/Server Applications Using the Socket Classes

The HLA sockets module makes creating client/server applications in assembly language almost trivial. The HLA Stdlib provides four classes for this purpose: server_t, vServer_t, client_t and vClient_t. The classes with the 'v' prefix use virtual methods for all functions, those without the 'v' prefix use static procedures. Generally, you'll use the static versions of these classes unless you need to create derived classes from them and overload some methods in these classes. Using the static classes produces more efficient code, but you lose the ability to overload the class functions (i.e., polymorphism is not possible with static classes). Note that you do not have to chose both virtual or static classes for your client and server applications. That is, one application can use a virtual class and the other can use a static class. The server and client applications and independent with respect to the choice of virtual or static classes. The examples in this document will use static classes because they don't require polymorphism, but you can easily substitute a virtual class into these examples, as needed.

26.8 A Simple Server Application

A server application is one that is running waiting for some other application (the client) to communicate with it. In particular, the server application must be running before the client application attempts to connect to it. Usually, the client and server applications run on separate computer systems on the network, though it is perfectly possible (and common for testing purposes) to run both applications on the same computer. The important thing to understand is that the server must be running before the client application begins because the client assumes that the server is available to provide services when it attempts to connect to the server.

To create a server application, you begin by declaring a *server_t* variable, e.g.,

```
static
  myServer  :server_t;
```

The *server_t* data type inherits all the functions from the baseSocketClass_t (which this document will describe later) and it adds two methods you can call: *start* and *close*. Though the server_t (and baseSocketClass_t) type has some data fields, you should consider them private to the class and never access or modify them directly.

The *start* method has the following prototype:

```
method start
(
    adrs                :dword;
    port                :word;
    timeoutCallback     :thunk;
    connectionCallback  :procedure
); @returns( "eax" );
```

The first parameter is the IP address that this server will be listening for clients on. This is a 32-bit IP address *in little endian format*! This value is *not* in big endian (network byte order) form.

The second parameter is the port (socket) number that the server will listen on for a connection from a client. This is a 16-bit value *in little endian (**not** network byte order/big endian) format*! The combination of IP address and socket port number is what uniquely identifies a particular server.

The *timeoutCallback* parameter is a **thunk** that the *start* function calls before attempting to listen for a client and on each timeout period while listening for a client. For those unfamiliar with thunks, they are simply procedures embedded in other code; when called, the EBP register is initialized to the value of EBP in the surrounding code when the thunk was initialized. This means that the code in the thunk can access variables that are local to the code that the thunk is embedded in.

When the *start* method invokes the *timeoutCallback* thunk code, the EAX register contains the address of a *sock.timeval* object that controls the timeout period. On the first call, the *start* method has initialized this timeout to zero (which means infinite timeout period). If your thunk code does not change this value, then the server will wait indefinitely for a connection and will never again invoke the timeout thunk. If you would like to have your timeout thunk invoked on a periodic basis while the server is listening for a client connection (perhaps to update a progress bar or something like that to indicate the program is still operating), you should initialize the *sock.timeval* value pointed at by EAX. Before returning, the thunk should load a Boolean value (true or false, 1 or 0) into EAX to tell the *start* method whether it should quit. False/0 means "don't quit", true/1 means "quit" (and return to whomever called the *start* method).

Here is a typical thunk that sets up a one-second timeout period:

```
static
  timeout   :thunk;
  calls     :dword := 0;
      .
      .
      .
  thunk timeout :=
  #{
        // On entry to thunk, EAX contains the address of the timeout
        // variable. Set this as desired for the timeout (1 second,
        // in this case).

        mov( 1, (type sock.timeval [eax]).tv_sec );
        mov( 0, (type sock.timeval [eax]).tv_usec );

        // On successive calls, print a period to the stdout
        // to let an observer know that we're still listening
        // for a connection:

        cmp( calls, 0 );
        je dontPrintPeriod;

           stdout.putc( '.' );

        dontPrintPeriod:
        inc( calls );
        mov( 0, eax );   // Never quit

  }#;
```

After initializing the *timeout* thunk as shown above, you can pass the *timeout* thunk variable to the *start* method.

The fourth parameter to the *start* method is the address of a procedure that *start* will call when it connects with a client application over the network. This is a standard HLA procedure with no parameters. This procedure must preserve all registers it modifies. This *connectionCallback* procedure provides whatever service the client requires and generally operates in one of two modes:

The *connectionCallback* procedure directly provides whatever services the client requires and then returns. At that point, the server and client are disconnected and the server starts listening for a new client. In this mode, the server can provide services to only one client at a time.

The

connectionCallback procedure spawns a new process to handle the client's requests and then immediately returns to the *start* method. The server then begins listening for a new client connection while the spawned

process provides appropriate services for the client. This mode allows the server to provide services to multiple clients simultaneously.

Upon entry into the *connectionCallback* procedure, the EAX register contains a socket handle and the ESI register contains the address of a *server_t* (or *vServer_t*) object. You should save these values in local (not static!) variables. At the very least, you will need the object address in order to communicate with the client.

If you choose to spawn a new process to provide services to the client, you must make a copy of the server object address passed to you in the ESI register. This is because the value passed in ESI is the main object used by the *start* procedure and upon returning to *start* after you spawn the new process, *start* will write to the data fields of that object. This would be a disaster for the service process. You can avoid this problem by making a copy of the *server_t* (or *vServer_t*) using code like the following:

```
procedure connected;
      @nodisplay;
      @nostackalign;
      @noframe;
   var
      handle    :dword;
      object    :pointer to server_t;
      newObject :pointer to server_t;

   begin connected;

      push( ebp );
      mov( esp, ebp );
      sub( _vars_, esp );

      pushad();
      pushfd();

      mov( eax, handle );
      mov( esi, object );
      server_t.create();
      mov( esi, newObject );

      // Duplicate the server object:

      mov( esi, edi );
      mov( object, esi );
      mov( @size( server_t ), ecx );
      cld();
      rep.movsb();

      /*****************************************************************/

      // Spawn process here and pass it the address of the new object
      // contained in newObject.

      // . . . . . .

      /*****************************************************************/

      // Return to start, so it can handle other client requests

      popfd();
      popad();
      leave();
      ret( _parms_ );

   end connected;
```

Note that the *server_t start* method will never return unless the *timeout* thunk returns true in the EAX register. Therefore, if you elect to specify an indefinite timeout by storing zero into the sock.timeval object passed to the *timeout* thunk, the *start* method will never return to its caller.

Whenever a *connectionCallback* procedure (or the thread it spawns) finishes providing services for a connected client, it should call the server object's *close* method to disconnect the socket from the client and free up system resources,. If you've allocated a new socket object (e.g., prior to spawning a new process), you should call the server *destroy* method and also free the storage associated with the socket object before terminating the spawned process.

26.9 A Simple Client Application

Setting up a client application is even easier than setting up a server application. You declare an object of type *client_t* (or *vClient_t*), invoke the object's *create* procedure, and then call the *connect* method to connect to a listening server. Here is the prototype for the *connect* method:

```
method connect
(
    adrs   :dword;
    port   :word
); @returns( "eax" );
```

The *adrs* parameter is the IP address (in little endian form). The *port* parameter is the socket port number (a 16-bit value, also in little endian form). When you call this function, it will attempt to connect to the server at the specified IP address and port number. If a server is not available or it refuses to connect to the client program, the *connect* method will raise an ex.SocketError exception. Note that the *connect* method will not wait for a server to become available. If it goes to the specify IP address and port number and there isn't a server application listing on that port, the *connect* method will raise an exception.

Once the *connect* method returns (without raising an exception), you can assume that the client and server are connected and communication between the two application may commence. When the client is done using the services of the server, it should call the *client_t* (or *vClient_t*) *close* method to disconnect itself from the server.

26.10 Client/Server Communication

Once a client establishes a connection to a server, the two applications may exchange data. A socket supports bidirectional data transfer; that is, the server can send data to the client and receive data from the client, and the client may send data to and receive data from the server.

The *server_t/vServer_t* and *client_t/vClient_t* classes are derived class that inherit all the information from the *baseSocketClass_t/vBaseSocketClass_t* classes. The *baseSocketClass_t/vBaseSocketClass_t* classes define all the procedures and methods that the server and client objects use to communicate via the socket. Without question, the most generic I/O functions (and the ones you will probably use most commonly) are the *read* and *write* functions. These methods/procedures use prototypes like the following:

```
method read( var buf:var; len:dword );
    @returns( "eax" );

method write( var buf:var; len:dword );
    @returns( "eax" );
```

The first parameter (*buf*) is the address of some block of memory, the second parameter (*len*) is the number of bytes to read or write at the address in memory. This function returns the number of bytes read or the number of bytes written in the EAX register.

All socket I/O communication is subject to a *timeout period*. The base socket class defines a (private) data field that specifies the timeout period in seconds and microseconds. You can use the socket class' *setTimeout* and *setTimeout2* functions to specify the timeout period (the default is zero, which means wait indefinitely). Should a timeout occur during a socket *read* or *write* call, the function will immediately return without completing the I/O operation. The EAX register will contain the actual number of bytes read or written; so you can check the return result to determine if the I/O operation was complete.

In addition to the generic *read* and *write* functions, the base socket classes provide a full set of formatted I/O functions similar to those provided by the *stdout, stdin, stderr,* and *fileio* modules. The following sections will describe the use of those functions.

26.11 General Socket Class Operations

The functions in this category let you initialize socket objects, access fields of the socket objects, and perform other conversion and housekeeping tasks.

<*object*>.create; @returns("esi");
server_t.create; @returns("esi"); [to create dynamic objects]
client_t.create; @returns("esi"); [to create dynamic objects]

The socket classes provide a <*socket*>.*create* constructor which you should always call before making use of a client or server variable. For socket variables (as opposed to socket pointer variables), you should call this routine specifying the name of the socket variable. For socket pointer variables, you should call this routine using the class name and store the pointer returned in EAX into your file variable. For example, to initialize the following two socket objects, you would use code like the following:

```
var
  MyClientSocket   : client_t;
  clientPtr        : pointer to vClient_t;

              .
              .
              .

  MyClientSocket.create();

  vclient_t.create();
  mov( eax, clientPtr);
```

Note that the *vClient_t.create* constructor simply initializes the virtual method table pointer and does other necessary internal initialization. The constructor does not open a socket or perform other socket-related activities.

<*object*>.destroy; @returns("esi");
server_t.destroy; @returns("esi"); [to create dynamic objects]
client_t.destroy; @returns("esi"); [to create dynamic objects]

The socket classes provide a <*socket*>.*destroy* destructor which you should always call when you're done using a socket. For example, when you are done working with the MyClientSocket and the clientPtr objects from the previous examples, you should execute the following code:

```
  MyClientSocket.destroy();

  clientPtr.destroy();
```

The socket destructor frees up system resources in use by an active socket. Note: some of these resources are system wide and may not be automatically reclaimed when your program terminates. Be sure you always call the destructor to prevent system resource leaks.

<*object*>.close;
server_t.close;
client_t.close;

This method closes a socket opened via <*object*>.*start* or <*object*>.*connect*.

```
MyClientSocket.close();
clientPtr.close();
```

Note that calling the *destroy* method/procedure does not close the socket. You must always call the *close* function before calling *destroy*. The difference between the two is that the close function tells the OS you're done using the socket, the *destroy* method deallocates resources associated with the HLA Standard Library.

```
<serverObject>.start
(
    adrs                    :dword;
    port                    :word;
    timeoutCallback         :thunk;
    connectionCallback      :procedure
);
```

This method starts a server that listens on IP address *adrs* and socket *port* number port for a connection from a client.

After setting up the server-side socket (but before checking for a client attempting to connect), this function calls the thunk specified by the *timeoutCallback* parameter. It passes the address of a *sock.timeval* variable to the thunk in the EAX register. The *timeoutCallback* thunk should set this *sock.timeval* variable to an appropriate timeout value (zero mean indefinite timeout). Generally, the timeout value is non-zero because you want to check the status of the listening socket on a periodic basis; further, the only way the *start* function ever returns to the caller is via a signal from the *timeoutCallback* thunk; Therefore, if you do not specify a timeout value (that is, if you specify an indefinite timeout period by writing zeros to the *sock.timeval* object), then *start* will have no way to terminate (other than by manually killing the process).

The *timeoutCallback* thunk passes true/false (1/0) back to the start function in the EAX register. If EAX contains true, then the *start* function returns to the caller and terminates listening for a client connection. If EAX contains zero upon return, then the *start* function continues to listen for a socket connection or until the timeout period expires.

The *start* function calls the procedure pointed at by the *connectionCallback* parameter whenever the server accepts a connection from a client. Upon entry into the *connectionCallback* procedure, ESI will contain the address of the server object (<serverObject>) and EAX will contain a copy of the new socket connection handle. At this point, most programs do one of two things: either the procedure pointed at by *connectionCall* provides all the services required by the client (during which the server will not accept any more client connections), or the procedure can spawn a new thread to provide those services and then immediately return (allowing the *start* function to handle additional client requests while the new thread provides any necessary services to the connected client).

The simplest case is to have the *connectionCallback* procedure provide all the services without spawning an new thread. In this case, the *connectionCallback* procedure would store the value in ESI into a *server_t* (or *vServer_t*) pointer variable and then use that pointer variable with all the I/O functions described in the following sections. When your code finishes, it simply returns to the *start* function and the server continues listening for new connections. Note, however, that while your server code is providing those services, the *start* function is suspended and your server will not accept any other client requests. This mode of operation is great for peer-to-peer type socket communications where only two network nodes communicate at one time.

If you want to allow your server to handle multiple client requests simultaneously, the situation is more complicated. First of all, you cannot simply store away the pointer held in the ESI register; instead, you have to make a copy of that object for use by the new thread and pass the address of this copy to the thread. After creating the copy, you should spawn a new thread (passing the address of the new object to the thread) and then reference the copy of the object within that thread. The reason for making a copy of the server object is because the server will modify that object on the next client connection. This would create problems for the current server thread.

Note that you don't actually have to create a new *server_t* or *vServer_t* object. The data server thread will only need a *baseSocketClass_t* (or *vBaseSocketClass_t*) object, so you can create one of those objects and then copy the pertinent fields from the *server_t/vServer_t* object to the *baseSocketClass_t/vBaseSocketClass_t* object (use the <object>*.assign* function to copy data from one *baseSocketClass_t/vBaseSocketClass_t* object into the current object).

Of course, don't forget that multithreaded applications have their own host of synchronization requirements. Also be aware that many of the stdlib functions are not (as this was being written) thread-safe, so be sure to protect stdlib calls with a mutex unless you are sure that the call you're making will function properly in a multithreaded environment.

```
MyServerSocket.start( $01020304, $1234, myTimeoutThunk, &connection );
```

<clientObject>.connect(IPadrs:dword; port :word);

This method connects a client to a server.

```
MyClientSocket.connect( $01020304, $1234 );
```

This function attempts to connect a client to a server. The IPadrs is the IP address of the server (in little endian form); port is the socket port number (also in little endian form). If the server is ready and willing to accept a socket connection, this function returns; if the server is not running, or is unwilling to connect, then this function raises an *ex.SocketError* exception.

<baseSocketClass_t>.assign(var src:baseSocketClass_t);
<vBaseSocketClass_t>.assign(var src:vBaseSocketClass_t);

These functions copy the pertinent data fields from the src operand to the current object. This is useful when creating a copy of a socket for use by a server thread (see the discussion in *start's* description).

```
MySocket.assign( (type baseSocketClass_t [esi]) );
```

<*object*>.setTimeout(timeout:sock.timeval);
<*object*>.setTimeout2(tv_sec:dword; tv_usec:dword);
<*object*>.getTimeout(var timeout:sock.timeval);

These functions get and set the internal timeout values for a socket object. The *getTimeout* function is an accessor that returns the value of the socket object's internal timeout value. The *setTimeout* and *setTimeout2* functions (which differ only insofar as how you pass the timeout argument) store their argument into the internal data field and they also notify the OS' socket package of the new timeout value.

```
mov( 1, timeValVar.tv_sec );
mov( 500_000, timeValVar.tv_usec );
MyClientSocket.setTimeout( timeValVar );
clientPtr.setTimeout2( 2, 0 );
  .
  .
  .
clientPtr.getTimeout( timeValVar );
```

Note: to ensure consistency with the system, the *getTimeout* function will actually write the internal timeout value to the system. This way, you're ensured that the value that *getTimeout* returns is the timeout value that the operating system will actually use.

<*object*>.getAdrs; @returns("eax");

This function returns the IP address associated with the object's socket. It returns the IP address in the EAX register in little endian form (not network byte order/big endian form).

```
MyClientSocket.getAdrs();
mov( eax, ipAdrs1 );
clientPtr.getAdrs();
mov( eax, ipAdrs2 );
```

<*object*>.setAdrs(adrs:dword);

This function stores the IP address passed as a parameter into the internal address field of the socket object. The *adrs* parameter contains the IP address in little endian form (not network byte order/big endian form). Note that this function is really intended for internal use by the socket classes. This function only stores the IP address

into the internal field. It does not update the IP address of any open socket and it does not change the IP address. The *client_t.connect* and *server_t.start* functions provide the mechanism for specifying the IP address of an internet connection. The *setAdrs* function exists so the *start* and *connect* functions can set the IP address in an object-oriented fashion. For that reason, this document is not providing any sample calls to this function.

<*object*>.getPort; @returns("ax");

This function returns the socket port number associated with the object's socket. It returns the port value in the AX register in little endian form (not network byte order/big endian form).

```
MyClientSocket.getPort();
mov( ax, port1 );
clientPtr.getPort();
mov( ax, port2 );
```

<*object*>.setPort(port:word);

This function stores the socket port number passed as a parameter into the internal port field of the socket object. The *port* parameter contains the port value in little endian form (not network byte order/big endian form). Note that this function is really intended for internal use by the socket classes. This function only stores the port number into the object's internal data field. It does not update the port number of any open socket and it does not change the port number in use. The *client_t.connect* and *server_t.start* functions provide the mechanism for specifying the port number of an internet connection. The *setPort* function exists so the *start* and *connect* functions can set the port value in an object-oriented fashion. For that reason, this document is not providing any sample calls to this function.

<*object*>.adrsToStr(s:string);
<*object*>.a_adrsToStr; @returns("eax");

These functions convert the IP address found in the socket's internal IP address data field to a string of the form "ddd.ddd.ddd.ddd". The *adrsToStr* function stores the string data into the string passed as the argument (raising an exception if that string has insufficient storage); the *a_adrsToStr* function allocates the storage on the heap and returns a pointer to that string in the EAX register.

```
MyClientSocket.adrsToStr( adrsStr );
clientPtr.a_adrsToStr();
mov( eax, adrsStr2 );
```

26.12 Miscellaneous Output

The following socket output routines all assume that you've opened the <object> socket variable via a call to *<serverObject>.start or <clientObject>.connect*.

<*object*>.write(var buffer:var; count:dword)

This method writes the number of bytes specified by the *count* parameter to the sockiet. The bytes starting at the address of the *buffer* byte are written to the file. No range checking is done on the *buffer*, it is your responsibility to ensure that the buffer contains at least *count* valid data bytes.

Note: Notice that the *buffer* parameter is an untyped reference parameter. Untyped reference parameters have special properties, so be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) if you are not absolutely sure you understand how untyped reference parameters operate.

```
HLA high-level calling sequence examples:
```

```
socketPtr.write( buffer, count );

// If bufPtr is a dword object containing the
// address of the buffer whose data you wish to
// write to the socket:

socketPtr.write( val bufPtr, count );

// The following writes the four-byte value of
// the bufPtr variable to the socket (an unusual
// operation):

socketPtr.write( bufPtr, 4 );
```

***<object>*.putbool(b:boolean);**

This procedure writes the string "true" or "false" to the <*object*> output socket depending on the value of the *b* parameter.

```
HLA high-level calling sequence examples:

socketPtr.putbool( boolVar );

// If the boolean is in a register (AL):

socketPtr.putbool( al );
```

***<object>*.newln( );**

This function writes a newline sequence (carriage return/line feed under Windows, linefeed under other operating systems) to the specified socket (<*object*>).

```
HLA high-level calling sequence examples:

socketPtr.newln();
```

26.13 Character, Character Set, and String Output

The following socket output routines all assume that you've opened the <object> socket variable via a call to <*serverObject*>*.start or* <*clientObject*>*.connect*.

<*object*>**.putc(c:char)**

Writes the character specified by the *c* parameter to the socket.

```
HLA high-level calling sequence examples:

socketPtr.putc( charVar );

// If the character is in a register (AL):

socketPtr.putc( al );
```

<*object*>.**putcSize(c:char; width:int32; fill:char)**

Outputs the character *c* to the socket specified by <object> using at least *width* output positions. If the absolute value of *width* is greater than one, then this function writes *fill* characters as padding characters during the output. If *width* is a positive value greater than one, then <*object*>.*putcSize* writes *c* left justfied in a field of *width* characters; if *width* is a negative value less than one, then <*object*>.*putcSize* writes *c* right justified in a field of *width* characters.

```
HLA high-level calling sequence examples:

socketPtr.putcSize( charVar, width, padChar );
```

<*object*>.**putcset(cst:cset);**

This function writes all the members of the *cst* character set parameter to the specified socket variable.

```
HLA high-level calling sequence examples:

socketPtr.putcset( csVar );
socketPtr.putcset( [ebx] ); // EBX points at the cset.
```

<*object*>.**puts(s:string);**

This procedure writes the value of the string parameter to the socket.

```
HLA high-level calling sequence examples:

socketPtr.puts( strVar );
socketPtr.puts( ebx ); // EBX holds a string value.
socketPtr.puts( "Hello World" );
```

<*object*>.**putsSize(s:string; width:int32; fill:char)**

This function writes the *s* string to the socket using at least *width* character positions. If the absolute value of *width* is less than or equal to the length of *s*, then this function behaves exactly like <*object*>.*puts*. On the other hand, if the absolute value of *width* is greater than the length of *s*, then <*object*>.*putsSize* writes *width* characters to the output file. This procedure emits the *fill* character in the extra print positions. If *width* is positive, then <*object*>.*putsSize* right justifies the string in the print field. If *width* is negative, then <*object*>.*putsSize* left justifies the string in the print field. Generally, people expect the string to be left justified, so you should ensure that this value is negative to achieve this.

```
HLA high-level calling sequence examples:

socketPtr.putsSize( strVar, width, ' ' );

// For the following, EBX holds the string value,
// ECX contains the width, and AL holds the pad
// character:

socketPtr.putsSize( ebx, ecx, al );

socketPtr.putsSize( "Hello World", 25, padChar );
```

<*object*>.**putz(z:zstring);**

This procedure writes the value of the zstring parameter to the socket.

```
HLA high-level calling sequence examples:

socketPtr.putz( zstrVar );
socketPtr.putz( ebx ); // EBX holds a zstring value.
socketPtr.putz( "Hello World" );
```

<*object*>.**putzSize(z:zstring; width:int32; fill:char)**

This function writes the *z* zstring to the socket using at least *width* character positions. If the absolute value of *width* is less than or equal to the length of *z*, then this function behaves exactly like <*object*>.*putz*. On the other hand, if the absolute value of *width* is greater than the length of *z*, then <*object*>.*putzSize* writes *width* characters to the output file. This procedure emits the *fill* character in the extra print positions. If *width* is positive, then <*object*>.*putzSize* right justifies the string in the print field. If *width* is negative, then <*object*>.*putzSize* left justifies the string in the print field. Generally, people expect the string to be left justified, so you should ensure that this value is negative to achieve this.

```
HLA high-level calling sequence examples:

socketPtr.putzSize( zstrVar, width, ' ' );

// For the following, EBX holds the string value,
// ECX contains the width, and AL holds the pad
// character:

socketPtr.putzSize( ebx, ecx, al );

socketPtr.putzSize( "Hello World", 25, padChar );
```

26.14 Hexadecimal Numeric Output

The following socket output routines all assume that you've opened the <object> socket variable via a call to *<serverObject>.start or <clientObject>.connect*.

<*object*>.**putb(b:byte);**

This procedure writes the value of *b* to the socket using exactly two hexadecimal digits (including a leading zero if necessary).

```
HLA high-level calling sequence examples:


socketPtr.putb( byteVar );

// If the character is in a register (AL):

socketPtr.putb( al );
```

<*object*>.**puth8(b:byte);**

This procedure writes the value of *b* to the socket using one or two hexadecimal digits (the minimum necessary).

```
HLA high-level calling sequence examples:


socketPtr.puth8( byteVar );
```

```
// If the character is in a register (AL):

socketPtr.puth8( al );
```

<*object*>.**puth8Size(b:byte; width:dword; fill:char)**

This procedure writes the value of *b* to the socket using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the <*object*>.*putcSize* function for more details on their behavior.

```
HLA high-level calling sequence examples:

socketPtr.puth8Size( byteVar, width, padChar );
```

<*object*>.**putw(w:word);**

This procedure writes the value of *w* to the socket using exactly four hexadecimal digits (including leading zeros if necessary).

```
HLA high-level calling sequence examples:


socketPtr.putw( wordVar );

// If the word is in a register (AX):

socketPtr.putw( ax );
```

<*object*>.**puth16(w:word);**

This procedure writes the value of *w* to the socket using 1-4 hexadecimal digits (the minimum necessary).

```
HLA high-level calling sequence examples:


socketPtr.puth16( wordVar );

// If the word is in a register (AX):

socketPtr.puth16( ax );
```

<*object*>.**puth16Size(w:word; width:dword; fill:char)**

This procedure writes the value of *w* to the socket using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the <*object*>.*putcSize* function for more details on their behavior.

```
HLA high-level calling sequence examples:

socketPtr.puth16Size( wordVar, width, padChar );
```

<*object*>.**putd(dw:dword);**

This procedure writes the value of *d* to the socket using exactly eight hexadecimal digits (including leading zeros if necessary). If the stdlib global underscores value (see the conversions module for details) contains true, then this function will also print an underscore between the fourth and fifth digits.

```
HLA high-level calling sequence examples:


socketPtr.putd(dwordVar );

// If the dword value is in a register (EAX):

socketPtr.putd( eax );
```

<*object*>.**puth32(dw:dword);**

This procedure writes the value of *d* to the file using the minimum number of hexadecimal required. If the stdlib global underscores value (see the conversions module for details) contains true, then this function will also print an underscore between the fourth and fifth digits (if there are at least five digits in the number).

```
HLA high-level calling sequence examples:


socketPtr.puth32( dwordVar );

// If the dword is in a register (EAX):

socketPtr.puth32( eax );
```

<*object*>.**puth32Size(d:dword; width:dword; fill:char)**

This procedure writes the value of *d* to the socket using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the <*object*>.*putcSize* function for more details on their behavior.

```
HLA high-level calling sequence examples:


socketPtr.puth32Size( dwordVar, width, ' ' );

// If the dword is in a register (EAX):

socketPtr.puth32Size( eax, width, cl );
```

<*object*>.**putq(q:qword);**

This procedure writes the value of *q* to the socket using exactly 16 hexadecimal digits (including leading zeros if necessary). If the stdlib global underscores value (see the conversions module for details) contains true, then this function will also print an underscore between each group of four digits.

```
HLA high-level calling sequence example:


socketPtr.putq( qwordVar );
```

<*object*>.**puth64(q:qword);**

This procedure writes the value of *q* to the socket using 1-16 hexadecimal digits (the minimum necessary). If the stdlib global underscores value (see the conversions module for details) contains true, then this function will also print an underscore between each group of four digits.

```
HLA high-level calling sequence example:


socketPtr.puth64( qwordVar );
```

<*object*>.**puth64Size(q:qword; width:dword; fill:char)**

This procedure writes the value of *q* to the socket using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the <*object*>.*putcSize* function for more details on their behavior.

```
HLA high-level calling sequence example:


socketPtr.puth64Size( qwordVar, width, ' ' );
```

<*object*>.**puttb(tb:tbyte)**

This procedure writes the value of *tb* to the socket using exactly 20 hexadecimal digits (including leading zeros if necessary and an intervening underscores if underscore output is enabled).

```
HLA high-level calling sequence examples:


socketPtr.puttb( tbyteVar );
```

<*object*>.**puth80(tb:tbyte)**

This procedure writes the value of *tb* to the socket using 1-20 hexadecimal digits (the minimum necessary) and an intervening underscores if underscore output is enabled).

```
HLA high-level calling sequence examples:


socketPtr.puth80( tbyteVar );
```

<*object*>.**puth80Size(tb:tbyte; width:dword; fill:char)**

This procedure writes the value of *tb* to the socket using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the <*object*>.*putcSize* function for more details on their behavior.

```
HLA high-level calling sequence examples:


socketPtr.puth80Size( tbyteVar, width, ' ' );
```

<*object*>.**putl(l:lword)**

This procedure writes the value of *l* to the socket using exactly 32 hexadecimal digits (including leading zeros if necessary and an intervening underscores if underscore output is enabled).

```
HLA high-level calling sequence examples:


socketPtr.putl( lwordVar );
```

<*object*>.**puth128(l:lword)**

This procedure writes the value of *l* to the socket using 1-32 hexadecimal digits (the minimum necessary) and an intervening underscores if underscore output is enabled).

```
HLA high-level calling sequence examples:


socketPtr.puth128( lwordVar );
```

<*object*>.**puth128Size(l:lword; width:dword; fill:char)**

This procedure writes the value of *l* to the socket using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the <*object*>.*putcSize* function for more details on their behavior.

```
HLA high-level calling sequence examples:


socketPtr.puth128Size( tbyteVar, width, ' ' );
```

26.15 Signed Integer Numeric Output

The following socket output routines all assume that you've opened the <object> socket variable via a call to <*serverObject*>.*start or* <*clientObject*>.*connect*.

These routines convert signed integer values to string format and write that string to the <*object*> socket. The <*object*>.*putxxxSize* functions contain *width* and *fill* parameters that let you specify the minimum field width when outputting a value.

If the absolute value of *width* is greater than the number of print positions the value requires, then these functions output *width* characters to the socket. If *width* is non-negative, then these functions right-justify the value in the output field; if value is negative, then these functions left-justify the value in the output field.

These functions print the *fill* character as the padding value for the extra print positions.

xxxSize(value, width, fill);

Assuming "value" requires five print positions, "width" is eight, and fill is "f" then the xxxSize functions produce the string

f	f	f	V	A	L	U	E

Assuming "value" requires five print positions, "width" is minus eight, and fill is "f" then the xxxSize functions produce the string

V	A	L	U	E	f	f	f

<*object*>.puti8 (b:byte);

This function converts the eight-bit signed integer you pass as a parameter to a string and writes this string to the socket using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


socketPtr.puti8( byteVar );

// If the character is in a register (AL):

socketPtr.puti8( al );
```

<*object*>.puti8Size (b:byte; width:int32; fill:char);

This function writes the eight-bit signed integer value you pass to the specified output socket using the *width* and *fill* values as specified above.

```
HLA high-level calling sequence examples:

socketPtr.puti8Size( byteVar, width, padChar );
```

<*object*>.puti16(w:word);

This function converts the 16-bit signed integer you pass as a parameter to a string and writes this string to the socket using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


socketPtr.puti16( wordVar );

// If the word is in a register (AX):

socketPtr.puti16( ax );
```

<*object*>.puti16Size(w:word; width:int32; fill:char);

This function writes the 16-bit signed integer value you pass to the specified socket using the *width* and *fill* values as specified above.

```
HLA high-level calling sequence examples:

socketPtr.puti16Size( wordVar, width, padChar );
```

<*object*>.puti32(d:dword);

This function converts the 32-bit signed integer you pass as a parameter to a string and writes this string to the socket using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


socketPtr.puti32( dwordVar );
```

```
// If the dword is in a register (EAX):

socketPtr.puti32( eax );
```

<*object*>.puti32Size(d:dword; width:int32; fill:char);

This function writes the 32-bit value you pass as a signed integer to the specified socket using the *width* and *fill* values as specified above.

```
HLA high-level calling sequence examples:


socketPtr.puti32Size( dwordVar, width, ' ' );

// If the dword is in a register (EAX):

socketPtr.puti32Size( eax, width, cl );
```

<*object*>.puti64(q:qword);

This function converts the 64-bit signed integer you pass as a parameter to a string and writes this string to the socket using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


socketPtr.puti64( qwordVar );
```

<*object*>.puti64Size(q:qword; width:int32; fill:char);

This function writes the 64-bit value you pass as a signed integer to the specified socket using the *width* and *fill* values as specified above.

```
HLA high-level calling sequence examples:


socketPtr.puti64Size( qwordVar, width, ' ' );
```

<*object*>.puti128(l:lword);

This function converts the 128-bit signed integer you pass as a parameter to a string and writes this string to the socket using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


socketPtr.puti128( lwordVar );
```

<*object*>.puti128Size(l:lword; width:int32; fill:char);

This function writes the 128-bit value you pass as a signed integer to the specified socket using the *width* and *fill* values as specified above.

```
HLA high-level calling sequence examples:


socketPtr.puti128Size( lwordVar, width, ' ' );
```

26.16 Unsigned Integer Numeric Output

These routines convert unsigned integer values to string format and write that string to the socket. The *<object>.putxxxSize* functions contain *width* and *fill* parameters that let you specify the minimum field width when outputting a value.

If the absolute value of *width* is greater than the number of print positions the value requires, then these functions output *width* characters to the socket. If *width* is non-negative, then these functions right-justify the value in the output field; if value is negative, then these functions left-justify the value in the output field.

These functions print the *fill* character as the padding value for the extra print positions.

<*object*>.putu8 (b:byte)

This function converts the eight-bit unsigned integer you pass as a parameter to a string and writes this string to the socket using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


socketPtr.putu8( byteVar );

// If the character is in a register (AL):

socketPtr.putu8( al );
```

<*object*>.putu8size(b:byte; width:int32; fill:char)

This function writes the unsigned eight-bit value you pass to the specified socket using the *width* and *fill* values as specified above.

```
HLA high-level calling sequence examples:

socketPtr.putu8Size( byteVar, width, padChar );
```

<*object*>.putu16(w:word)

This function converts the 16-bit unsigned integer you pass as a parameter to a string and writes this string to the socket using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


socketPtr.putu16( wordVar );

// If the word is in a register (AX):

socketPtr.putu16( ax );
```

<*object*>.putu16size(w:word; width:int32; fill:char)

This function writes the unsigned 16-bit value you pass to the specified socket using the *width* and *fill* values as specified above.

```
HLA high-level calling sequence examples:

socketPtr.putu16Size( wordVar, width, padChar );
```

<*object*>.putu32(d:dword)

This function converts the 32-bit unsigned integer you pass as a parameter to a string and writes this string to the socket using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


socketPtr.putu32( dwordVar );

// If the dword is in a register (EAX):

socketPtr.putu32( eax );
```

<*object*>.putu32Size(d:dword; width:int32; fill:char)

This function writes the unsigned 32-bit value you pass to the specified socket using the *width* and *fill* values as specified above.

```
HLA high-level calling sequence examples:


socketPtr.putu32Size( dwordVar, width, ' ' );

// If the dword is in a register (EAX):

socketPtr.putu32Size( eax, width, cl );
```

<*object*>.putu64(q:qword)

This function converts the 64-bit unsigned integer you pass as a parameter to a string and writes this string to the socket using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


socketPtr.putu64( qwordVar );
```

<*object*>.putu64Size(q:qword; width:int32; fill:char);

This function writes the unsigned 64-bit value you pass to the specified socket using the *width* and *fill* values as specified above.

```
HLA high-level calling sequence examples:


socketPtr.putu64Size( qwordVar, width, ' ' );
```

<*object*>.putu128(l:lword)

This function converts the 128-bit unsigned integer you pass as a parameter to a string and writes this string to the socket using the minimum number of print positions the number requires.

```
HLA high-level calling sequence examples:


socketPtr.putu128( lwordVar );
```

<*object*>.putu128Size(l:lword; width:int32; fill:char);

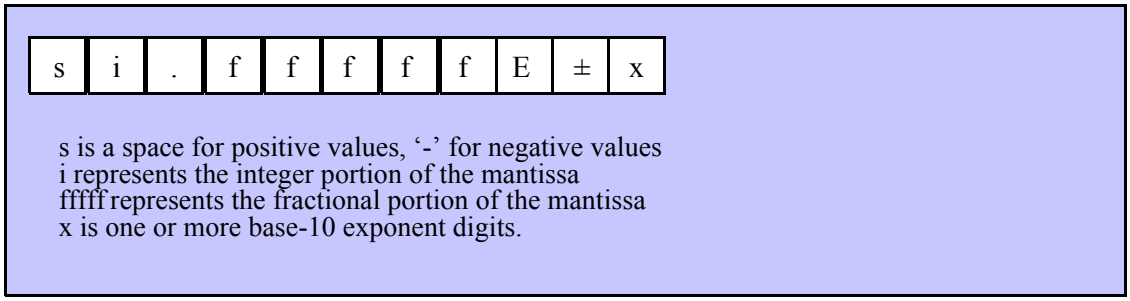
This function writes the unsigned 128-bit value you pass to the specified socket using the *width* and *fill* values as specified above.

```
HLA high-level calling sequence examples:


socketPtr.putu128Size( lwordVar, width, ' ' );
```

26.17 Floating-Point Numeric Output Using Scientific Notation

The floating point numeric output routines translate the three different binary floating point formats to their string representation and then write this string to the socket that <object> specifies. There are two generic classes of these routines: those that convert their values to exponential/scientific notation and those that convert their string to a decimal form.

The *<object>.pute80*, *<object>.pute64*, and *<object>.pute32* routines convert their values to a string using scientific notation. These three routines each have two parameters: the value to output and the field width of the result. These routines produce a string with the following format:



<*object*>.pute32(r:real32; width:uns32)

This function writes the 32-bit single precision floating point value passed in *r* to the socket using scientific/exponential notation. This procedure prints the value using *width* print positions in the file. *width* should have a minimum value of five for real numbers in the range 1e-9..1e+9 and a minimum value of six for all other values. Note that 32-bit extended precision floating point values support about 6-7 significant digits. So a *width* value that yields more than seven mantissa digits will produce garbage output in the low order digits of the number.

```
HLA high-level calling sequence examples:
```

```
socketPtr.pute32( r32Var, width );

// If the real32 value is in an FPU register (ST0):

var
  r32Temp   :real32;
  .
  .
  .
fstp( r32Temp );
socketPtr.pute32( r32Temp, 12 );
```

<object>.pute64(r:real64; width:uns32)

This function writes the 64-bit double precision floating point value passed in *r* to the socket using scientific/exponential notation. This procedure prints the value using *width* print positions in the file. *width* should have a minimum value of five for real numbers in the range 1e-9..1e+9 and a minimum value of six for all other values. Note that 64-bit double precision floating point values support about 15 significant digits. So a *width* value that yields more than 15 mantissa digits will produce garbage output in the low order digits of the number.

HLA high-level calling sequence examples:

```
socketPtr.pute64( r64Var, width );

// If the real64 value is in an FPU register (ST0):

var
  r64Temp:real64;
  .
  .
  .
fstp( r64Temp );
socketPtr.pute64( r64Temp, 12 );
```

<object>.pute80(r:real80; width:uns32)

This function writes the 80-bit extended precision floating point value passed in *r* to the socket using scientific/exponential notation. This procedure prints the value using *width* print positions in the file. *width* should have a minimum value of five for real numbers in the range 1e-9..1e+9 and a minimum value of six for all other values. Note that 80-bit extended precision floating point values support about 18 significant digits. So a *width* value that yields more than 18 mantissa digits will produce garbage output in the low order digits of the number.

HLA high-level calling sequence examples:

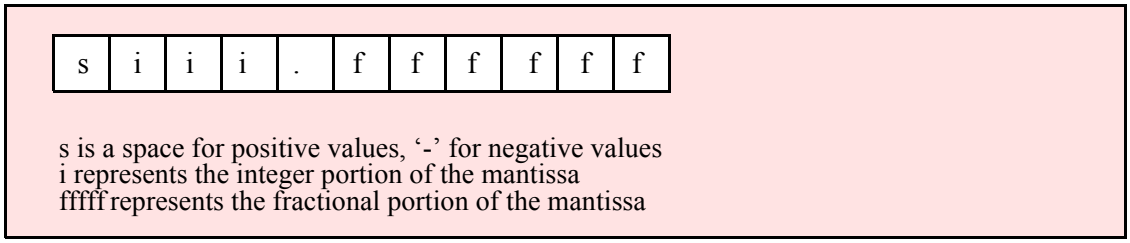
```
socketPtr.pute80( r80Var, width );

// If the real80 value is in an FPU register (ST0):

var
  r80Temp   :real80;
  .
  .
  .
fstp( r80Temp );
```

```
socketPtr.pute80( r80Temp, 12 );
```

26.18 Floating-Point Numeric Output Using Decimal Notation

Although scientific (exponential) notation is the most general display format for real numbers, real numbers you display in this format are very difficult to read. Therefore, the HLA socket class module also provides a set of functions that output real values using the decimal representation. Although you cannot (practically) use these decimal output routines for all real values, they are applicable to a wide variety of common numbers you will use in your programs.

These functions come in two varieties. The first variety requires four parameters: the real value to convert, the width of the converted value, the number of digit positions to the right of the decimal point, and a padding character. The second variety only requires the first three parameters and assumes the padding character is a space. These functions write their values using the following string format:



<*object*>.putr32(r:real32; width:uns32; decpts:uns32; fill:char)

This procedure writes a 32-bit single precision floating point value to the socket as a string. The string consumes exactly *width* characters in the output file. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than *width* characters, then this procedure uses the *fill* value as the padding character to fill the output with *width* characters.

```
HLA high-level calling sequence examples:


socketPtr.putr32( r32Var, width, decpts, fill );
socketPtr.putr32( r32Var, 10, 2, '*' );

// If the real32 value is in an FPU register (ST0):

var
  r32Temp   :real32;
  .
  .
  .
fstp( r32Temp );
socketPtr.putr32( r32Temp, 12, 2, al );
```

<*object*>.putr64(r:real64; width:uns32; decpts:uns32; fill:char)

This procedure writes a 64-bit double precision floating point value to <object> socket as a string. The string consumes exactly *width* characters in the output. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than *width* characters, then this procedure uses the value of *fill* as the padding character to fill the output with *width* characters.

```
HLA high-level calling sequence examples:
```

```
socketPtr.putr64( r64Var, width, decpts, fill );
socketPtr.putr64( r64Var, 10, 2, '*' );

// If the real64 value is in an FPU register (ST0):

var
  r64Temp   :real64;
  .
  .
  .
fstp( r64Temp );
socketPtr.putr64( r64Temp, 12, 2, al );
```

<*object*>.putr80(r:real80; width:uns32; decpts:uns32; fill:char)

This procedure writes an 80-bit extended precision floating point value to the socket as a string. The string consumes exactly *width* characters in the output. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than *width* characters, then this procedure uses the value of *fill* as the padding character to fill the output with *width* characters.

```
HLA high-level calling sequence examples:

socketPtr.putr80( r80Var, width, decpts, fill );
socketPtr.putr80( r80Var, 10, 2, '*' );

// If the real80 value is in an FPU register (ST0):

var
  r80Temp   :real80;
  .
  .
  .
fstp( r80Temp );
socketPtr.putr80( r80Temp, 12, 2, al );
```

26.19 Generic File Output

<object>.put(parameter_list)

<*object*>*.put* is a macro that automatically invokes an appropriate <*object*> output routine based on the type of the parameter(s) you pass it. This is a very convenient output routine and is probably the socket class formatted output call you will use most often in your programs. Keep in mind that this macro is not a single function call; instead, HLA translates this macro into a sequence of calls to procedures like *<object>.puti32, <object>.puts,* etc.

<*object*>*.put* is a macro that provides a flexible syntax for outputting data to the socket. This macro allows a variable number of parameters. For each parameter present in the list, <*object*>*.put* will call the appropriate routine to emit that data, according to the type of the parameter. Parameters may be constants, registers, or memory locations. You must separate each macro parameter with a comma.

Here is an example of a typical invocation of an <*object*>*.put* :

```
<object>.put( "I=", i, " j=", j, nl );
```

The above is roughly equivalent to

```
<object>.puts( "I=" );
<object>.puti32( i );
<object>.puts( " j=" );
<object>.puti32( j );
<object>.newln();
```

This assumes, of course, that *i* and *j* are *int32* variables.

The *<object>.put* macro also lets you specify the minimum field width for each parameter you specify. To print a value using a minimum field width, follow the object you wish to print with a colon and the value of the minimum field width. The previous example, using field widths, could look like the following:

```
<object>.put( "I=", i:2, " j=", j:5, nl );
```

Although this example used the literal decimal constants two and five for the field widths, keep in mind that register values and memory value (integers, anyway) are prefectly legal here.

For floating point numbers you wish to display in decimal form, you can specify both the minimum field width and the number of digits to print to the right of the decimal point by using the following syntax:

```
<object>.put( "Real value is ", f:10:3, nl );
```

The *<object>.put* macro can handle all the basic primitive types, including boolean, unsigned (8, 16, 32, 64, 128), signed (8, 16, 32, 64, 128), character, character set, real (32, 64, 80), string, and hexadecimal (byte, word, dword, qword, tbyte, lword).

If you specify a class variable (object) and that class defines a *toString* method, the *<object>.put* macro will call the associated *toString* method and output that string to the socket. Note that the *toString* method must dynamically allocate storage for the string by calling *str.alloc*. This is because *<object>.put* will call *str.free* on the string once it outputs the string.

There is a known "design flaw" in the *<object>.put* macro. You cannot use it to print HLA intermediate variables (i.e., non-local VAR objects). The problem is that HLA's syntax for non-local accesses takes the form "reg32:varname" and *<object>.put* cannot determine if you want to print *reg32* using *varname* print positions versus simply printing the non-local *varname* object. If you want to display non-local variables you must copy the non-local object into a register, a static variable, or a local variable prior to using *<object>.put* to print it. Of course, there is no problem using the other *<object>.putXXXX* functions to display non-local VAR objects, so you can use those as well.

Important(!), don't forget that method calls (e.g., the routines that *<object>.put* translates into) modify the values in the ESI and EDI registers. Therefore, it never makes any sense to attempt to print the values of ESI and EDI within the parameter list. All you will wind up doing is printing the address of the file variable (ESI) or the address of its virtual method table (EDI). If you need to write these two values to a file, move them to another register or a memory location first.

26.20 Generic File Input

The following socket input routines behave just like their standard input and file input counterparts (unless otherwise noted). Because of the nature of sockets, it is not possible to provide an "end-of-file" function that tests whether you're currently at the end of file on an input stream. End of file is determined by a timeout (set by the *setTimeout* and *setTimeout2* functions). Whenever a timeout occurs while the program is waiting for an input from a socket, the system translates that timeout into an *ex.EndOfFile* exception. Therefore, you should really surround all socket input requests with a **try..endtry** sequence that handles the *ex.EndOfFile* exception.

<object>.read(var buffer:var; count:dword)

This will probably be the most commonly-called input function in a typical socket-based application. This function reads *count* bytes from the socket and stores them into memory starting with the first byte of the *buffer* variable. This routine does not do any range checking. It is your responsibility to ensure that *buffer* is large enough to hold the data read.

Note: Notice that the *buffer* parameter is an untyped reference parameter. Untyped reference parameters have special properties, so be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) if you are not absolutely sure you understand how untyped reference parameters operate.

```
HLA high-level calling sequence examples:


socketPtr.read( buffer, count );
socketPtr.read( [eax], 1024 );
```

<*object*>.readLn;

This function reads and discards all characters up to and including the newline sequence in the socket stream.

```
HLA high-level calling sequence examples:

socketPtr.readLn();
```

26.21 Character and String Input

The following functions read character data from a socket. Note that HLA's socket class module does not provide the ability to read character set data directly from the user. However, you can always read a string and then convert that string to a character set using the appropriate function in the *cset* module.

<*object*>.getc; @returns("al");

This function reads a single character from the socket and returns that character in the AL register.

```
HLA high-level calling sequence examples:

socketPtr.getc();
mov( al, charVar );
```

<*object*>.gets(s:string);

This function reads a sequence of characters from the socket through to the next end of line sequence and stores these characters (without the end of line sequence) into the string variable you pass as a parameter. Before calling this routine, you must allocate sufficient storage for the string. If <*object*>.*gets* attempts to read a larger string than the string's *MaxLen* value, <*object*>.*gets* raises a string overflow exception.

Note that this function does not store the end of line sequence into the string, though it does consume the end of line sequence. The next character a file class function will read from the socket will be the first character of the following line.

If the incoming socket data is a newline sequence, then <*object*>.*gets* consumes the end of line and stores the empty string into the *s* parameter.

```
HLA high-level calling sequence examples:


socketPtr.gets( inputStr );
socketPtr.gets( eax ); // EAX contains string value
```

<*object*>.a_gets; @returns("eax");

Like <*object*>.*gets*, this function also reads a string from the socket. However, rather than storing the string data into a string you supply, this function allocates storage for the string on the heap and returns a pointer to this string in the EAX register. You code should call *str.free* to release this storage when you're done with the string data.

The <*object*>.*a_gets* function imposes a line length limit of 4,096 characters. If this is a problem, you should modify the source code for this function to raise the limit. This functions raises an exception if you attempt to read a line longer than this internal limit.

```
HLA high-level calling sequence examples:
```

```
socketPtr.a_gets();
mov( eax, inputStr );
```

26.22 Signed Integer Input

The functions in this group read numeric values from the socket using a signed decimal integer format. These functions read the string data, translate it to numeric form, and return that numeric data in an appropriate location.

<*object*>.geti8; @returns("al");

This function reads a signed eight-bit decimal integer in the range -128..+127 from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of file (i.e., timeout). This function allows underscores in the interior of the number. The <*object*>*.geti8* function raises an appropriate exception if the input violates any of these rules or the value is outside the range -128..+127. This function returns the binary form of the integer in the AL register.

HLA high-level calling sequence examples:

```
socketPtr.geti8();
mov( al, i8Var );
```

<*object*>.geti16; @returns("ax");

This function reads a signed 16-bit decimal integer in the range -32768..+32767 from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The <*object*>*.geti16* function raises an appropriate exception if the input violates any of these rules or the value is outside the range -32768..+32767. This function returns the binary form of the integer in the AX register.

HLA high-level calling sequence examples:

```
socketPtr.geti16();
mov( ax, i16Var );
```

<*object*>.geti32; @returns("eax");

This function reads a signed 32-bit decimal integer in the (approximate) range ±2 Billion from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The <*object*>*.geti32* function raises an appropriate exception if the input violates any of these rules or the value is outside the range plus or minus two billion. This function returns the binary form of the integer in the EAX register.

HLA high-level calling sequence examples:

```
socketPtr.geti32();
```

```
mov( eax, i32Var );
```

<*object*>.geti64; @returns("edx:eax");

This function reads a signed 64-bit decimal integer from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. *The <object>.geti64* function raises an appropriate exception if the input violates any of these rules or the value is outside the range of a 64-bit signed integer. This function returns the 64-bit result in the EDX:EAX register pair (it returns the H.O. dword in EDX and the L.O. dword in EAX).

HLA high-level calling sequence examples:

```
socketPtr.geti64();
mov( edx, (type dword i64Var[4]) );
mov( eax, (type dword i64Var[0]) );
```

<*object*>.geti128(var l:lword);

This function reads a signed 128-bit decimal integer from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The *<object>.geti128* function raises an appropriate exception if the input violates any of these rules or the value is outside the range of a 128-bit signed integer. This function stores the 128-bit result into the lword you pass as a reference parameter.

HLA high-level calling sequence examples:

```
socketPtr.geti128( lwordVar );
```

26.23 Unsigned Integer Input

The functions in this group read numeric values from the socket using an unsigned decimal integer format. These functions read the string data, translate it to numeric form, and return that numeric data in an appropriate location.

<*object*>.getu8; @returns("al");

This function reads an unsigned eight-bit decimal integer in the range 0..+255 from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The *<object>.getu8* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..255. This function returns the binary form of the integer in the AL register.

HLA high-level calling sequence examples:

```
socketPtr.getu8();
mov( al, u8Var );
```

<*object*>.getu16; @returns("ax");

This function reads an unsigned 16-bit decimal integer in the range 0..+65535 from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The <*object*>.*getu16* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..65535. This function returns the binary form of the integer in the AX register.

```
HLA high-level calling sequence examples:


socketPtr.getu16();
mov( ax, u16Var );
```

<*object*>.getu32; @returns("eax");

This function reads an unsigned 32-bit decimal integer in the range 0..+4,294,967,295 from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The <*object*>.*getu32* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..4,294,967,295. This function returns the binary form of the integer in the EAX register.

```
HLA high-level calling sequence examples:


socketPtr.getu32();
mov( eax, u32Var );
```

<*object*>.getu64; @returns("edx:eax");

This function reads an unsigned 64-bit decimal integer from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The <*object*>.*getu64* function raises an appropriate exception if the input violates any of these rules or the value is outside the range $0..2^{64}-1$. This function returns the binary form of the integer in EDX:EAX register pair (EDX contains the H.O. dword, EAX holds the L.O. dword).

```
HLA high-level calling sequence examples:


socketPtr.getu32();
mov( eax, (type dword u64Var) );
mov( edx, (type dword u64Var[4]) );
```

<*object*>.getu128(var l:lword);

This function reads an unsigned 128-bit decimal integer from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The <*object*>.*getu64* function raises an appropriate exception if the input violates any of these rules or the value is outside the range $0..2^{128}-1$. This function returns the binary form of the integer in the lword parameter you pass by reference.

```
HLA high-level calling sequence examples:

socketPtr.getu128( u128Var );
```

26.24 Hexadecimal Input

***<object>*.geth8; @returns("al");**

This function reads an eight-bit hexadecimal integer in the range 0..$FF from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The *<object>.geth8* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..$FF. This function returns the binary form of the value in the AL register.

```
HLA high-level calling sequence examples:


socketPtr.geth8();
mov( al, h8Var );
```

***<object>*.geth16; @returns("ax");**

This function reads a 16-bit hexadecimal integer in the range 0..$FFFF from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The *<object>.geth16* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..$FFFF. This function returns the binary form of the value in the AX register.

```
HLA high-level calling sequence examples:


socketPtr.geth16();
mov( ax, h16Var );
```

***<object>*.geth32; @returns("eax");**

This function reads a 32-bit hexadecimal integer in the range 0..$FFFF_FFFF from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The *<object>.geth32* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..$FFFF_FFFF. This function returns the binary form of the value in the EAX register.

```
HLA high-level calling sequence examples:


socketPtr.geth32();
mov( eax, h32Var );
```

***<object>*.geth64; @returns("edx:eax");**

This function reads a 64-bit hexadecimal integer in the range 0..$FFFF_FFFF_FFFF_FFFF from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The *<object>.geth64* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..$FFFF_FFFF_FFFF_FFFF. This function returns the 64-bit result in the EDX:EAX register pair (EDX contains the H.O. dword, EAX contains the L.O. dword).

HLA high-level calling sequence examples:

```
socketPtr.geth64();
mov( edx, (type dword h64Var[4]) );
mov( eax, (type dword h64Var[0]) );
```

***<object>*.geth128(var l:lword);**

This function reads a 128-bit hexadecimal integer in the range 0..$FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The *<object>.getq* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..$FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF . This function stores the 128-bit result into the variable you pass as a reference parameter.

HLA high-level calling sequence examples:

```
socketPtr.geth128( lwordVar );
```

26.25 Floating-Point Input

***<object>*.getf; @returns("st0");**

This function reads an 80-bit floating point value in either decimal or scientific from from the socket and leaves the result sitting on the FPU stack. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by an optional minus sign and a sequence of characters that represent a floating point value. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. This function raises an appropriate exception if an error occurs.

HLA high-level calling sequence examples:

```
socketPtr.getf();
fstp( fpVar );
```

26.26 Generic File Input

<object>.get(List_of_items_to_read);

This is a macro that allows you to specify a list of variable names as parameters. The *<object>.get* macro reads an input value for each item in the list and stores the resulting value in each corresponding variable. This macro determines the type of each variable that you pass it and emits a call to the appropriate *<object>.getxxx* function to read the actual value. As an example, consider the following call to *<object>.get*:

```
socketPtr.get( i32, charVar, u16, strVar );
```

The macro invocation above expands into the following:

```
push( eax );
socketPtr.geti32( i32 );
socketPtr.getc();
mov( al, charVar );
socketPtr.geti16();
mov( ax, u16 );
socketPtr.gets( strVar );
pop( eax );
```

Notice that *<object>.get* preserves the value in the EAX register even though various *<object>.getxxx* functions use this register. Note that *<object>.get* automatically handles the case where you specify EAX as an input variable and writes the value to *[esp]* so that in properly modifies EAX upon completion of the macro expansion.

Note that *<object>.get* only supports eight-, sixteen-, and thirty-two bit integer input. If you need to read 64-bit or 128-bit values, you must use the appropriate *<object>.getx64* or *<object>.getx128* function to achieve this.