APL Network Interface

Basic Information

TCP/IP comprises a set of protocols designed to communicate over the Internet. TCP/IP is designed to work across dissimilar networks and computers. As such, it also works within any type of network that can be attached to the Internet. This includes most networks in common use today.

TCP/IP comprises two layers. The lower level is IP - the Internet Protocol; the higher level is TCP - Transmission Control Protocol.

The Internet Protocol (IP) is designed to deliver small packets of data across a network from one machine to another. Within each data packet there is a header that specifies the destination machine. IP works like the postal service:

- The address is on the packet.
- The address is all that is needed to make the delivery
- IP uses services available to it to figure out where to deliver the packet.
- A direct connection between sender and receiver is not required.
- There is no checking to see that the packet has arrived intact.
- Packets may arrive in a different sequence from which they were sent.

The Transmission Control Protocol (TCP) allows a connection to be established, with large volumes of data transmitted over the connection as if the connection were permanent and hardwired. TCP takes care of the obvious requirements not covered by IP:

- It provides a reliable stream transport service.
- It takes large messages, disassembles them into packets and hands them off to IP for delivery.
- It reassembles packets into the original message form at the receiving end.
- It handles error recovery and message re-transmission.

- "Sockets" is an abstraction originally implemented in 4.3 BSD UNIX for communications over the Internet. Sockets provides a set of functions to make connections between machines on a network and transfer data between these machines using a variety of protocols. Sockets provides access to the TCP/IP protocols as well as others.

Contents

TCP/IP in APL64

APL64 implements access to the communications protocols using the system function ⎕ni, the network interface. ⎕ni is a sockets interface for the APL+Unix and APL64 systems; it implements a set of sockets functions common to BSD sockets and WinSock 1.1. ⎕ni also adds some platform specific features. In APL64, ⎕ni accepts certain *properties*, *event-handlers*, and *methods*, which are analogous to the actions you use with ⎕wi, the windows interface function for building a graphical user interface. In particular, the asynchronous notification event handlers allow you to implement your communications application in a manner that takes advantage of the underlying event-driven architecture of Windows and respond with actions at appropriate times without needing to rely on continual polling.

Stream Communication

The primary steps in a TCP/IP application are:

1. Establish a socket connection between machines.
2. Send and Receive data between machines.
3. Close the connection.

Creating a Sockets Connection

The first step on both the server and client sides is to create a socket. This is done with the Socket method; arguments specify the address family, communications type and protocol. If you want to use the defaults which are internet, stream communication, and TCP/IP, you can omit the arguments. The function returns a completion code and the socket handle. For purposes of this example, assume this is the server socket.

```
      ⎕ni 'Socket'
0 0 3
```

The Socket method creates a communications endpoint. It consists of an open socket and a socket handle. The socket handle is a small integer that serves as the program handle to the socket. From this point forward, the process refers to this socket by the handle which in this example is 3.

On the server side, the next step is to name the socket. This is done with the Bind method. You specify the host network address and the port number as arguments. Sockets methods are available to determine the host address and well-known port numbers (numbers for certain services are standardized on all systems; see GetServByName). Other port numbers may be chosen by convention.

```
      3 ⎕ni 'Bind' 2002 '192.168.160.212'
0 0
```

The address and port constitute the socket name. They serve to identify the socket to its client, but one more step is needed before a client can Connect. The socket needs to be in Listen state to hear a Connect request from a client. A Connect request before this will be rejected. Listen also allows a queue depth to be specified.

```
      3 ⎕ni 'Listen'
0 0
```

If both sides of your communication are APL-based, the action and arguments to create a socket on the client side are the same. The handle the system returns for each socket is unique to each process. For purposes of this example, assume this is the client socket.

```
      ⎕ni 'Socket'
0 0 5
```

Now the client can issue a Connect specifying the client socket handle and the server socket name (port number and internet address).

```
      5 ⎕ni 'Connect' 2002 '192.168.160.212'
¯1 10035
      ⎕ni 'Error' 10035
WSAEWOULDBLOCK: Operation would block
```

Note: The error code WSAEWOULDBLOCK occurs when using Connect if the socket is still in the default fakeblocking mode, since the connection has not yet been established. You can use the onConnectNotify event handler to receive a message notifying you of the connection.

The connection is now completed by action on the server side. When an incoming Connect is pending, using the Accept method creates a new socket, with a new handle. This new socket completes the connection with the client socket. The original socket is then free to accept further connect requests.

```
      3 ⎕ni 'Accept'
 0 0 7 2 1026 192.168.160.218
```

The new socket is number 7, the third element of the result of the Accept method. All read and write actions on the server side for this connection will use this new socket. It inherits the characteristics and event handlers, if any, of the original socket except that it is not in listening mode. It has a one-to-one connection with the socket on the client. This behavior allows the server to create multiple connections from one listening socket without reconfiguring each new connected socket when a connection request is received.

Sending and Receiving Data

Now that the sockets are connected, intermachine communication begins. Either side can initiate communication. The standard convention is that the client does the first send, the server does the first receive; this allows the client to request a service. The Select method or the ReadNotify event can be used to detect arriving data.

The Send method sends data over a connected socket.

rc ← *socket* ⎕ni 'Send' *data datatype flags*

The datatypes for ⎕ni 'Send' are: 'char', 'bool', 'int', 'float', and 'apl'.
For types other than 'apl', APL64 coerces the type of the data if possible. If not, a domain error occurs. When sending Boolean data, you should send multiples of eight values.

The flags field may contain one of the following:

'dontroute'	avoid using local routing tables (intended for network debugging)
'oob'	send character out of band (for urgent signal)

The Recv method receives data sent over a connected socket

(*rc, data*) ← *socket* ⎕ni 'Recv' *datatype flags data_length*

The datatypes are the same as for send. The data are interpreted according to the specified type. If the data were transmitted assuming a different datatype, unintended results may occur. If you use `'apl'` as the datatype to receive data sent with any other datatype, unintended results will occur, including the possibility of crashing the system.

The flags field may contain the following:

`'peek'`	read data without removing it from the system buffers.
`'oob'`	receive out of band characters.

Unneeded trailing parameters may be omitted. If you do not specify a datatype, the default is `'apl'`. The *flags* parameter defaults to an empty vector. The *data_length* parameter defaults to `0`. If the data length is `0` on Recv, the system reads the current data length from the buffer. If the type specified is `'apl'`, the entire array is read and reconstituted.

`7 ⎕ni 'Recv' 'char'`	Reads the contents of the buffer from socket `7` as character data.
`7 ⎕ni 'Recv' 'apl'`	Reads an entire APL array on socket `7`.
`5 ⎕ni 'Send'` (*data array*)	Sends an APL array on socket `5`.

Closing the Connection

Closing the connection is done with the Close method. To give advance notification to the remote machine, you can use the Shutdown method. This allows cleanup activities to take place before the connection is closed.

A Simple Example

This example shows a simple communication where the client sends a character string containing a request to the server, which executes the request as an APL statement and returns the result.

Server Side

⍝ *For purposes of the example, we use the scalar dyadic plus function to create an array that consists of its row numbers:*

```
      rugged←(100 10 ⍴ 0) +[1]⍳100
```

⍝ *These statements create a socket and wait for the client:*

```
     ⎕ni 'Socket'
0 0 11
     11 ⎕ni 'Bind' 100 '192.168.160.3'
0 0
     11 ⎕ni 'Listen'
0 0
```

Client Side

⍝ *These statements create a socket on the client side and attempt to make a connection:*

```
     ⎕ni 'Socket'
0 0 31
     31 ⎕ni 'Connect' 100 '192.168.160.3'
¯1 10035
```

Server Side

⍝ *This statement completes the connection:*

```
      11 ⎕ni 'Accept'
0 0 12 2 1057 192.168.160.212
```

⍝ *This statement sends a character string that defines three rows of the array:*

```
    31 ⎕ni 'Send' 'rugged[3 50 99;]' 'char'
0 0 16
```

⍝ *These statements read the string, execute the statement and return the result as apl data:*

```
      var←12 ⎕ni 'Recv' 'char'
      data←⍎(⊃var[3])
      12 ⎕ni 'Send' data
0 0 152
```

⍝ *This statement reads the result:*

```
     31 ⎕ni 'Recv'
0 0      3  3  3  3  3  3  3  3  3  3
        50 50 50 50 50 50 50 50 50 50
        99 99 99 99 99 99 99 99 99 99
```

Controlling your Application

Because communication using sockets is inherently asynchronous and unpredictable, your application must recognize the appropriate times to take certain actions. The listen, connect, and accept actions must occur in the correct order. The send action can be performed only when the buffers are available (which may be a factor when you are transmitting large amounts of data). You may also need to specify whether your application will recognize other Windows messages during a long send or receive action.

In general, it is expected that you will not invoke the Recv method until data are available to read. If you use the Recv method when no data are present, your application will freeze, return an error, or loop, depending on the mode of the socket (see below) and the datatype.

You can control your application either by checking the status of each socket (polling) on a regular basis and performing the appropriate action when the status changes, or you can use the Windows event notifications to trigger actions at the appropriate time. Traditional communications applications used polling techniques; however, the Windows paradigm (as with ⎕wi) is to use the messages provided by the operating system as event triggers and to have the application respond by means of its event handlers.

Controlling your Application using Methods

There are various methods for determining the state of your application. The Sockets method tells you what sockets exist in your application. The InUse method tells you which sockets have an action in progress on them. You can use the Control method to check how much data is in a buffer, to determine if urgent (out-of-band) characters are waiting to be read, to stop a looping read action (waiting for data in fakeblocking mode, or reading an APL array), and to allow or disallow Windows messages to be recognized during a looping read or send. You can also use Control to empty a read buffer, for example, when you are debugging an application, and to set a blocking mode.

The most powerful tool for polling is the Select method. You can use this method to check one or more selected sockets to determine if data are to be read (or if a connect is pending on a listening socket), if buffers are available for sending data, or if an error condition or urgent data are pending. When you use Select, you specify a socket number and which state or states you are checking. If you have multiple sockets to consider you can apply the method sequentially or you can query multiple sockets in the same call.

You can have the method return immediately, in which case your application can determine that no action is required or it can respond with an appropriate action. Your application must determine how often to poll using Select.

Alternatively, you can specify Select with a timer of a specific or indefinite length. In this case, the method returns when the state you are querying changes (or when the specified time elapses). If you are querying multiple sockets in the same call, the method returns when a queried state on any of them changes.

Using Select with a timer allows the operating system to notify your application of a change in state; however, it is not the same as using events. With the timer, your application initiates the timing of the response, whereas with events, the notification (that is, the event) triggers the response (the event handler).

Modes of Sockets

You must be aware of the various blocking modes on Windows in order to control your application. In particular, if you have programmed using sockets on UNIX, the Windows behavior will require different techniques. The three modes are:

- **Blocking Mode**
  In blocking mode, `⎕ni` with most methods returns only after completion. When a method cannot complete (such as Accept when there is no connection), it waits, blocking most other actions, until the current action is satisfied. If you invoke Recv in blocking mode when no data are available to read, your application waits. Whenever handlers are specified for a socket, you cannot set it to blocking mode.

  Blocking mode may be appropriate for simple applications; in particular, when you control both the client and the server, and thus have more knowledge of and control over the transmission of data, you may find blocking mode useful. It cannot, however, be considered robust. In the event of an unexpected failure, such as the interruption of a connection during a transmission, you may be forced to shut down your application abruptly.
- **Nonblocking Mode**
  In nonblocking mode, `⎕ni` returns with an error if there is nothing available to satisfy the action. The return code is WSAEWOULDBLOCK. By default, sockets are in nonblocking mode, except for the Send and Recv methods, which are fakeblocking.

Nonblocking mode provides the most flexibility and control; however, your application must be prepared to handle all the error conditions and to respond to all possible occurrences in any sequence.

- **Fakeblocking Mode**
  Actions that are blocking in UNIX are interruptible; in Windows they are not. To avoid locking up your system waiting for data to arrive, the system provides fakeblocking mode for Send and Recv. If you do a Recv with a data length specified in fakeblocking mode when no data are available to read, your application loops, doing repeated attempts to read interspersed with `⎕wgive` function calls to allow interruptions. Fakeblocking mode allows window paints to happen, also allows the user to break out of the loop using Ctrl+Break, and allows other Windows events, which could, for example, invoke the Control method to stop the Recv. (However, see the section "Suppressing Overlapping Actions" below.)

  Although fakeblocking mode is the default for the Send and Recv methods, it is not recommended for finished applications. It is most useful for learning the system, experimenting in immediate execution mode, and for building an application.

Modes affect only methods that operate on one specified socket. These modes do not apply to methods that operate on more than one socket, for example Select, nor to methods that operate without a socket, for example GetHostByName. You can set a mode on a socket using the Control method or the blockingmode property.

Notes on Receiving Data

When you invoke the Recv method, the amount of data returned is determined by the arguments to Recv, the blocking mode of the socket, and the amount of data in the WinSock buffer for the socket.

For the APL datatype, the system on the sending side encodes the data length as part of the transmission. The system on the receiving side does repeated reads to receive the entire APL array; it interprets the data length as part of its first read and continues to read until it has read the entire array.

If you are reading anything other than an APL array (character, Boolean, integer, or floating point datatypes), the length specification determines the behavior. If you do not specify a data length or if you specify zero, Recv immediately returns all the data in the buffer, up to `1024` bytes at a time. If there are no data in the buffer when you specify zero length or do not specify a length, the blocking mode determines the behavior. If the socket is in blocking mode, Recv waits until data are received in the buffer or the method is interrupted. Otherwise, the system returns the WSAEWOULDBLOCK error code.

If you specify a nonzero data length, Recv attempts to read exactly that much data from the socket. If that amount of data, or more, is present, the system returns the amount specified. If there is less than the amount specified in the buffer, and the socket is in blocking or fakeblocking mode, Recv continues to read until the data length is satisfied, an error condition occurs, or the method is interrupted. In either of the latter two cases, the system returns any data that are present along with the status or error codes. In nonblocking mode, the system returns the available data along with the WSAEWOULDBLOCK error code.

You can interrupt a Recv that is in fakeblocking mode at any time using the Control method with `'stop'` as the argument. You can interrupt a Recv that is in blocking mode using the Control method whenever the Recv method loops, for example, if some data arrive but not enough to satisfy the length condition specified.

Controlling your Application using Events

As an alternative or an adjunct to using the Select method, Windows triggers events that can cause asynchronous notification for sockets. You can use these event notifications to perform actions when and only when they are appropriate. Note that these events are named essentially for the action that is next to occur rather than an action that has just occurred. The events are:

Name	Trigger	Practical Meaning	Re-enabling Action
AcceptNotify	connection request received	ready to do an Accept (listening socket)	Accept
CloseNotify	connection close received	ready to do a Close	None
ConnectNotify	connection has been established	Send and Recv now possible (see note 1)	None
OobNotify	out-of-band character ready to read	ready to Recv with OOB flag	Recv or RecvFrom with OOB flag
ReadNotify	data available to read	ready to Recv	Recv or RecvFrom
WriteNotify	network system buffers available	ready to Send (see note 2)	Send or SendTo

Note 1: The ConnectNotify event occurs only for the socket and the application that performs the Connect action. It occurs when the connecting (client) side recognizes that an application (server) is listening at the address and the port that was specified in the Connect action. It generally occurs before the Accept action by the listening side.

Note 2: The WriteNotify event occurs once when the onWriteNotify event handler is set. Thereafter, it occurs only after the buffer has been full and writing was blocked, and then the buffer is again empty. It does not occur after each successful Send operation.

Setting event handlers with ⎕ni

You can use ⎕ni to set handlers for any combination of these events. As with the Windows Interface, event handlers are named by prefacing the event name with "on". Thus the event handler for the AcceptNotify event is onAcceptNotify. A handler can be any valid APL expression.

You can specify each event-name/event-handler pair in a separate statement. For example, to perform an Accept on socket 7 and read the new socket number.

```
7 ⎕ni 'onAcceptNotify' 'soc← 3⊃7 ⎕ni ''Accept'' '
```

You can use the Set method to specify multiple event handlers in a single statement. For example, to set handlers for read and close notification on socket 8:

```
8 ⎕ni 'Set' ('onReadNotify' 'rcvfn') ('onCloseNotify' 'endfn')
```

When data are available to read on socket 8, the system executes the APL codestring 'rcvfn'. When the partner does a Close, the APL codestring 'endfn' executes.

If you use multiple event handlers on one socket, it may be preferable to set all the handlers in one statement. Because of the underlying Sockets architecture, the system must reset a previously-specified event handler when you specify a new one in a separate statement. This may cause spurious event messages to your application. For example, if you have previously set an onWriteNotify event handler and you then set an onReadNotify event handler, the system will trigger the WriteNotify event by virtue of re-establishing the original handler (assuming the write buffer is empty).

To retrieve the list of events and handlers set for a socket, use the state property. To delete an event handler, you can assign an empty vector.

Events and Actions

When an event occurs, the system executes the specified APL event handler. The system variable `⎕narg` contains an integer error code or zero. The system variable `⎕nevent` contains a character vector event name (such as ReadNotify). The system variable `⎕nself` contains the socket handle as a one-element vector. The execution stack entry for the event includes the socket number and the event-handler name:

```
>[Socket 8; onReadNotify]
```

Once an event occurs on a socket, that same event does not occur on that socket again until a re-enabling action for that event is executed. The re-enabling actions are listed in the table above. Note that the re-enabling action occurs on the same side of the communication as the event. For example, if the client side sends data, the presence of data triggers the onReadNotify event handler. If the server side does a Recv on only part of the data that were sent, either because you limited the read length or because the number of characters exceeded the length of the buffer, the presence of more data triggers another ReadNotify event. It does not require a second Send. Thus, one Send can require multiple Recv actions, or one Recv can accommodate multiple Send actions, even when you are recognizing these with event handlers.

When an Accept is done on a socket, the resulting new socket inherits all of the event handlers in effect for the listening socket. This is extremely useful behavior for a server. You can open the socket, Bind it, set onAcceptNotify to perform the Accept action and onReadNotify to perform a Recv, and put it in Listen state. Then the system is ready to have multiple clients connect and send data.

Note that no socket actions may be performed on a socket for which the Recv action is currently in progress. An attempt to perform an action returns a result of `1 1`. You can check to see if a socket currently has an action in progress on it by using the InUse socket method.

vector_of_sockets `⎕ni 'InUse'`

This method returns a vector with one element per element in vector_of_sockets. The values are

- `0` – socket not in use
- `1` – socket in use
- `¯1` – not a valid socket.

Note that if you perform multiple Recv actions in your onReadNotify event handler, you may cause what seem to be anomalous events. When you read multiple strings of data before your event handler returns, each Recv action may generate another ReadNotify event. When your event handler returns, there can be a stack of ReadNotify events that will call your handler again after the buffer is empty.

This problem does not occur when you are reading APL data. The multiple Recv actions that the system performs when reading an APL array do not trigger multiple ReadNotify events.

Caution: Under some circumstances when the system is receiving APL data, other Windows events and execution of APL code may occur. While the event handler is running, the Recv action remains suspended. Remaining in this state for a significant amount of time may have undesirable effects on the application, including blocking any action on the sending side.

Suppressing Overlapping Actions

There is yet another feature you can use to control your application. By default, when using event handlers, the system can process Windows messages during a long read or write action, or while waiting for more data to complete a read. The system allows this behavior because it reads blocks of `1024` bytes (unless you have specified a shorter length). After each read, the system performs a `⎕wgive` to allow other Windows messages to be handled.

If you want to prevent your application from handling those messages during a read or write, you can use the Control method on the appropriate socket with `'nogives'` as the argument. This suppresses the messages until the read or write action is complete. However, note that if you use fakeblocking mode and specify a length parameter for reading data, you probably do not want to suppress Windows messages. If your application triggers the read action when not enough data are present to satisfy the condition, your system will be blocked, and you will not be able to interrupt it using the Control method.

In general, it is most straightforward to use nonblocking mode, particularly when you do not control both ends of the communication stream.

Limitations on using Out-of-Band Data

You may find only limited utility in the out-of-band data facility under the Windows operating system. Some texts recommend avoiding it entirely. Among other problems, you may encounter inconsistencies among various implementations of the underlying TCP/IP protocol as to which character in the read buffer is marked as oob.

When using the oob flag with the Send or SendTo methods, you should send only one character (byte) at a time. Although the system will pass more data, only one character is marked as oob on the receiving end of the transmission. You can build up a multi-character message by sending characters with multiple actions, but even then you must be careful. If you send single characters in a tight loop, not all of them may be received as oob data.

Depending on the modes of the sockets and whether each side is set to allow gives during a transmission, you may or may not be able to actually interleave an out-of-band character with normal data over a single connection. If you are using multiple socket connections, for example with the ftp protocol, you may not need to distinguish oob data.

A Simple Example using Event Handlers

This example assumes you have set up a socket as a server, and the client has agreed to send an out-of-band single-character transmission to specify the datatype of the ensuing transmissions.

```
15 ⎕ni 'onReadNotify' 'Readfn'
15 ⎕ni 'onOobNotify'  "typecode "3 ⊃ ⎕nself ⎕ni 'Recv' 'char' 'oob' "
15 ⎕ni 'onAcceptNotify' '15 ⎕ni ''Accept'' ⋄ data←"" '

    ∇Δ Readfn
[1]     :select  typecode
[2]       :case ,'c'
[3]         type←'char'
[4]       :case ,'b'
[5]        type  ← 'bool'
[6]      :case ,'i'
[7]        type ← 'int'
[8]      :case ,'f'
[9]        type ← 'float'
[10]      :case ,'a'
[11]        type ← 'apl'
[12]    :endselect
[13]  data ← data, (⎕nself ⎕ni 'Recv' type)[3]
     ∇
```

Using Event Handlers for a Web-based Application

For generalized and well documented examples of using `⎕ni` in APL64 to access the Web, see the `INTERNET.W3` workspace, written by Gary Bergquist, in the Examples folder. His examples include an APL web browser that will connect to a web server or a proxy server, download pages and files, and parse HTML. He also shows an APL web server that can handle requests from a standard web browser.

A very elaborate communications facility is demonstrated in the `ITALK.W3` workspace, also in the Examples folder; in addition to demonstrating internet access functions, it uses a technique of defining and erasing each function as it is used so that no trace of the communications shows in your active workspace. A simpler and less polished example of a chat facility is shown below using datagrams.

Using Datagrams

The SendTo and RecvFrom methods are intended to be used with datagrams as part of the Universal Datagram Protocol. This is a connectionless protocol. To use the SendTo and RecvFrom methods, create a socket on both the server and client; specify the dgram socket type and udp protocol.

```
⎕ni 'Socket' 'inet' 'dgram' 'udp'
```

Then do a `⎕ni 'Bind'` on each side. The IP address should be that of the local host on each side. Port numbers may be chosen independently on each side. After the Bind is done on each side, you can use SendTo and RecvFrom without using Connect or Accept. You should use datatypes other than `'apl'`.

```
result ← socket ⎕ni 'RecvFrom' datatype   flags
result ← socket ⎕ni 'SendTo' data   datatype   flags   port   inet_address
```

Note that to send a datagram, you must use the port number and address of the other side.

A Rudimentary Chat Facility using Datagrams

The functions below define a simple chat facility that uses datagrams to send messages without a connection first being established. The facility assumes that you know the host names of other machines using the facility and that all participants have agreed on a common port number. The list of possible partners and the host names of their respective machines are contained in a nested vector defined in the Make function.

The primary element of the user interface is a RichEdit control where you can type a message and where incoming messages are appended. Incoming messages are bulleted and indented to distinguish them from typed input. Two buttons allow you to send your typed message and clear the edit box.

There are six functions in the `MSSNGR` workspace. In reversed order of complexity, these functions are:

- `Reset` – which closes all open sockets when the Messenger form closes.
- `ClearBox` – which deletes all text from the Message box.
- `FindStart` – which saves the cursor position when you start typing. This function is called by the onModified event handler.
- `Send` – which captures all the text from the saved cursor position to the end of the box and sends a datagram to the selected recipient.
- `Read` – which captures an incoming datagram and displays it in the message box. This function also inserts a carriage return before the incoming message and bullets the message to distinguish it from material typed into the box. This function is called by the onReadNotify event handler of the socket.
- `Messenger_Make` – which creates and binds a socket for datagram communication; defines the table of potential message partners; creates the form for the user interface; and defines the event handlers.

You can find the `MSSNGR` workspace in the `Examples` directory of the appropriate version of this APL64 system. You can run this facility with others by changing line `17` of `Messenger_Make` to contain the names and machine host names of the participants and loading the workspace on each machine.

```
   ∇ Reset
[1]  (⎕ni 'Sockets') ⎕ni 'Close'
   ∇

   ∇ ClearBox
[1]  Box ⎕wi 'text' ''
[2]  Box ⎕wi 'selbullet' 'none'
[3]  Box ⎕wi 'selindents' 0 0 0
[4]   ⍝  Reset the modified property for next typing
[5]  'Messenger' ⎕wi 'Modify' 0
   ∇

   ∇ FindStart
[1] ⍝  Saves the cursor position
[2]  :if 1= ⎕warg[1]
[3]  :andif 1=⎕warg[4]
[4]  start←1⊃Box ⎕wi 'selection'
[5]  :end
   ∇

   ∇ Send;message;address;You
[1]  ⍝  Captures the new text and sends a datagram
[2]  ⍝  Identify the target for your message
[3]   You←1+'Messenger.cbRecipients' ⎕wi 'value'
[4]   address←⊃table[You;3]
[5]
[6]  ⍝  Select the text for the message
[7]   Box ⎕wi 'selection' (start-1) ¯1
[8]   message←Box ⎕wi 'seltext'
[9]
[10] ⍝  Send the message
[11]  :if 0=⍴message
[12]      →end
[13]  :else
[14]      0 0⍴sock ⎕ni 'SendTo' message 'char' '' port address
[15]  :endif
[16]
[17] ⍝  Append a carriage return and move cursor to end
[18]  Box ⎕wi 'selection' ¯1 0
[19]  Box ⎕wi 'seltext' ⎕tcnl
[20]  Box ⎕wi 'selindents' 0 0 0
[21]  Box ⎕wi 'selection' ¯1 0
[22]  Box ⎕wi 'modified' 0
[23]  end:
   ∇

   ∇ Read;message
[1]  ⍝  Reads, formats and displays the incoming datagram
[2]   message←3⊃sock ⎕ni 'RecvFrom' 'char'
[3]
[4]  ⍝  Append a carriage return and move cursor to end
[5]   Box ⎕wi 'selection' ¯1 0
```

```
[6]   Box ⎕wi 'seltext' ⎕tcnl
[7]   Box ⎕wi 'selection' ¯1 0
[8]   Box ⎕wi 'ScrollCaret'
[9]
[10] ⍝  Append the message at a bullet
[11]  Box ⎕wi 'selindents'  0 2 0
[12]  Box ⎕wi 'selbullet' 'bullet'
[13]  0 0⍴Box ⎕wi 'seltext' message
[14]
[15] ⍝  Undo bulleting
[18]  Box ⎕wi 'selindents'  0 0 0
[19]  Box ⎕wi 'selbullet' 'none'
[20]  Box ⎕wi 'selection' ¯1 0
[21]
[22] ⍝  Reset the modified property for next typing
[23]  Box ⎕wi 'modified' 0
   ∇

   ∇ Messenger_Make;name;machine;hostnames;guys;i;Mgr
[1]  ⍝'Messenger_Make -- Re-created 5/01
[2]  ⍝'This program functions similar to a chat utility.
[3]
[4]   Reset
[5]   'Messenger' ⎕wi 'Delete'
[6]
[7]  ⍝ Establish port number for all participants and set up socket
for this machine
[8]   port← 2007
[9]   name← 3 ⊃ ⎕ni 'GetHostName'
[10]  machine← 1⊃ 7⊃ ⎕ni 'GetHostByName' name
[11]  sock← 3⊃ ⎕ni 'Socket' 'inet' 'dgram' 'udp'
[12]  0 0⍴sock ⎕ni 'Bind' port (,⊃machine)
[13]  0 0⍴sock ⎕ni 'onReadNotify' 'Read'
[14]
[15] ⍝ Set up table of participants by retrieving ip addresses for
their host names
[16]  table← 1 3⍴ 'name' 'hostname' 'address'
[17]  hostnames←'jeremy' 'wbetmainj' 'mark' 'dropped' 'john' 'walkerj'
[18]  guys←(⍴hostnames)÷2
[19]  table←table⍪((guys, 2) ⍴hostnames),⊂''
[20]  :for i :in 1+⍳guys
[21]    table[i;3]←1↑ 7⊃⎕ni 'GetHostByName' (⊃table[i;2])
[22]  :endfor
[23]  table←table⍪'self' name machine
[24]
[25] ⍝ Create form and controls for interface
[26]  Mgr←'Messenger' ⎕wi 'New' 'Form' 'Close' ('where' 1 1 14 26)
[27]  Mgr ⎕wi 'border' 48
[28]  Mgr ⎕wi 'onClose' ('sock ⎕ni ''Close''')
[29]  Mgr ⎕wi 'onOpen' ('⎕wi ''visible'' 2')
[30]
[31]  0 0⍴'Messenger.lTo' ⎕wi 'New' 'Label' ('caption' 'To:')
```

```
[32]  'Messenger.lTo' ⎕wi ('where' 1 1 1 10)
[33]
[34]  0 0⍴'Messenger.cbRecipients' ⎕wi 'New' 'Combo' ('where' 2 1 4
8)
[35]  'Messenger.cbRecipients' ⎕wi ('style' 18)
[36]  'Messenger.cbRecipients' ⎕wi 'list' ((1 0↓table)[;1])
[37]
[38]  0 0⍴'Messenger.lMessage' ⎕wi 'New' 'Label' ('caption'
'Message:')
[39]  'Messenger.lMessage' ⎕wi ('where' 4 1 1 10)
[40]
[41]  Box←'Messenger.redMessage' ⎕wi 'New' 'RichEdit' ('where' 5 1 5
23.5)
[42]  Box ⎕wi 'style' 4 16 32 8192 16384
[43]  Box ⎕wi 'onModified' 'FindStart'
[44]
[45]  0 0⍴'Messenger.bnSend' ⎕wi 'New' 'Button' ('style' 1)
[46]  'Messenger.bnSend' ⎕wi ('where' 10.5 1.5 1.5 10)
[47]  'Messenger.bnSend' ⎕wi ('caption' 'Send Message')
[48]  'Messenger.bnSend' ⎕wi ('onClick' 'Send')
[49]
[50]  0 0⍴'Messenger.bnClear' ⎕wi 'New' 'Button' ('style' 1)
[51]  'Messenger.bnClear' ⎕wi ('where' 10.5 12.5 1.5 6)
[52]  'Messenger.bnClear' ⎕wi ('caption' 'Clear')
[53]  'Messenger.bnClear' ⎕wi ('onClick' 'ClearBox')
[54]
[55]  'Messenger' ⎕wi 'Show
   ∇
```

You could add small features to this rudimentary facility, for example, a label on the form to show the sender of a message. You could expand it more, for example, to capture the address of a sender who is not in the table. And, you could even package it so it leaves no trace in the workspace as mentioned above.

Reference Section

⎕ni Network Interface

Syntax:
result ← ⎕ni '*action_name*' *arguments*
result ← *socket* ⎕ni '*action_name*' *arguments*

Result:

For most actions, including setting properties and event handlers, ⎕ni returns a multi-element result. The first two elements report the state of the action. The first element is a status code. A status code of 0 indicates a successful action. A status code of ¯1 indicates an error in the underlying sockets action. A status code of 1 indicates a non-error condition that prevents completion of the action. This status code usually occurs only on a socket that is, or has been, involved in a communication.

For a 0 status code, the next element will also be zero. For a ¯1 status code, the next element is an integer error value; you can look up the meaning using the Error method of ⎕ni.

For a `1` status code, the next element is one of a small number of integer values that further define the condition preventing successful completion.

- If the partner socket has been closed, this element has a value of `0`. A close should be done on the local socket.
- If a receive is pending on a socket when another action is attempted on that same socket, this element has a value of `1`. The action will not succeed until the Recv is completed.
- If the Control method with a `'stop'` argument has been performed during a Recv or a Send, this element has a `2`.

When the action name is a method, elements three and beyond provide the functional result of the action. This may include socket numbers, internet addresses, data sent over the network, or other values. These elements may follow regardless of the status code and error values.

Not every method follows this format; the socket methods that always return information and have no normal error return conditions do not follow the status code format.

`⎕ni` uses the following methods, properties, event handlers and system variables.

Socket Methods		**Socket Properties**	**Socket Event Handlers**
Accept	InUse	blockingmode	OnAcceptNotify
Bind	Listen	condition	OnCloseNotify
Close	Recv	events	OnConnectNotify
Connect	RecvFrom	family	OnOobNotify
Control	Ref	methods	OnReadNotify
Error	Select	properties	OnWriteNotify
Errors	Send	protocol	
GetHostByName	SendTo	state	**Socket system variables**
GetHostName	Set	type	`⎕narg`
Get Peer Name	Shutdown	`⍙`udp	`⎕nevent`
GetProtoByName	Socket		`⎕nself`
GetServByName	Sockets		
GetSockName	WSAStartup		

Socket Methods

Accept

Purpose:
Completes a connection on a listening socket. An Accept is typically done following a connect by a partner. Accept returns when a connection is completed. A new socket is created that completes the connection with the partner. The original socket remains listening and available for subsequent connections. The new socket inherits the event handlers defined for the listening socket.

Syntax:
result ← *socket* `⎕ni 'Accept'`

Arguments:
socket is the socket handle for the listening socket.

Result:
The result of Accept is a six-element nested vector, or (on error) a two-element numeric vector.

Element `1`: `0` for completion of a connection, `¯1` for an error.
Element `2`: `0` on completion of connection, error code on error.
Element `3`: New socket number.
Element `4`: Address family; should be `2` (inet).
Element `5`: Port number of partner.
Element `6`: A character vector representing the network address of the partner.

Bind

Purpose:
Binds a socket to a port and an internet address. The port and address are used by a remote system to identify the target in a Connect request.

Syntax:
result ← *socket* `⎕ni 'Bind'` *port inet_address*

Arguments:
socket is the socket handle for the listening socket.
port is the chosen port number for the server application. You can assign any integer between `1` and `65535` as a port number; however, on many UNIX systems, port numbers less than `1024` are used for well known functions. It is recommended that you use port numbers in the range `1025-5000`. You can specify zero to have the system assign a port number.
inet_address is, optionally, the internet address of the machine running the application. If you omit this argument, the value defaults to `0.0.0.0`; subsequently, the system supplies the primary host address.

Result:
The result of Bind is a two-element numeric vector.

Element `1`: `0` for successful completion, `¯1` for an error.
Element `2`: `0` for successful completion, error code on error.

Close

Purpose:
Terminates socket connections and delete the sockets.

Syntax:
result ← *sockets* `⎕ni 'Close'`

Arguments:
sockets is an integer scalar or vector with the socket handle or list of socket handles to be closed.

Result:
The result of Close is a two-element numeric vector.

Element `1:` `0` for successful completion, `¯1` for an error.
Element `2:` `0` for successful completion, error code on error.

Connect

Purpose:
Requests connection of a local socket to a remote socket. The remote socket should be bound and in listening state.

Syntax:
result ← *socket* `⎕ni 'Connect'` *port inet_address*

Arguments:
socket is the socket handle for the local socket.
port is the port number for the chosen application.
inet_address is the internet address of the remote target machine, or its host name.

Result:
The result of Connect is a two-element numeric vector.

Element `1:` `0` for completion of a connection, `¯1` for an error.
Element `2:` `0` for successful completion, error code on error.

Note: You may get an error indication (`¯1`) with the code for WSAEWOULDBLOCK when using Connect if the socket is still in the default fakeblocking mode. Despite this, the connection may occur successfully immediately thereafter with no additional return. If you have set an onConnectNotify event handler, the successful connection will trigger the handler.

Control

Purpose:
Sets and interrogates control states of a socket.

Syntax:
result ← *socket* `⎕ni 'Control'` *function*

Arguments:
socket is the socket handle to be controlled.
function is the specific control function to be performed. The valid values include:

`'nonblocking'`	sets socket to nonblocking mode.
`'blocking'`	sets socket to blocking mode.
`'fakeblocking'`	sets socket to fakeblocking mode (default state).
`'peek'`	returns the number of bytes of data available to read in the buffer.
`'oob'`	returns `1` if there is out-of-band data to read, `0` if not.
`'stop'`	interrupts a looping Recv or Send method.
`'flush'`	empties all the data from the buffer (usually used when debugging).
`'gives'`	allows other actions to occur during long reads or writes
`'nogives'`	blocks other actions during a long read or write

Result:
The result of Control is a two- or three-element vector.

Element `1`: `0` on successful completion, `¯1` on error.
Element `2`: `0` on successful completion, error code on error.
Element `3`: the number of bytes of data available when the argument is `'peek'`.
`1` for out of band data present, `0` for none, when the argument is `'oob'`.

Notes:
`'fakeblocking'` mode is useful under Windows to allow interruption of sockets functions.

The system allows an implicit `⎕wgive` each time a read or write action exceeds a `1`K buffer.

Error

Purpose:
Fetches a descriptive error message

Syntax:
result ← `⎕ni 'Error'` *error_number*

Arguments:
error_number is the value returned as the second element of a `⎕ni` function returning an error.

Result:
A character vector describing the error corresponding to *error_number*.

Errors

Purpose:
Returns the complete list of error numbers and error messages. You can use this list to become familiar with the errors and to write code to respond to them.

Syntax:
result ← ⎕ni 'Errors'

Result:
A three column matrix of status code, error number, and error message.

GetConst

Purpose:
Return the integer value of either named constant for a socket type.

Syntax:
result ← ⎕ni 'GetConst' *constant_name*

Arguments:
constant_name is one of the possible socket types, either: 'SOCK_STREAM' or 'SOCK_DGRAM'

Result:
The result of GetConst is a three-element integer vector where the first two elements are zero and the third element is the value of the named constant.

Note: You can use an expression such as (⎕ni 'GetConst' 'SOCK_STREAM), for example, in a :case statement to match the value returned by the GetSockOpt method for so_type. See the description of the new method GetSockOpt in this update manual.

GetHostByAddr

Purpose:
Fetches information about the host identified by an IP address.

Syntax:
result ← ⎕ni 'GetHostByAddr' *inet_address address_family*

Arguments:
inet_address is the internet address of the host machine.
address_family -- the only value currently allowed is 'inet', the family of internet protocols. This is the default.

Result:
The result of GetHostByAddr is a seven-element nested or two-element numeric vector.

Element 1: 0 for successful completion, ¯1 for an error.
Element 2: 0 for successful completion, error code on error.
Element 3: A character vector containing the name of the local host machine.
Element 4: A vector of character vectors. These are aliases for the host machine (if they exist).
Element 5: Host address family; should be 2 (inet).
Element 6: Length of address field in bytes (useful only in conjunction with an external routine).
Element 7: A vector of character vector address representations for the network addresses of the specified host machine.

GetHostByName

Purpose:
Fetches information about the named host.

Syntax:
result ← ⎕ni 'GetHostByName' *hostname*

Arguments:
hostname is a character vector containing a name of the host machine.

Result:
The result of GetHostByName is a seven-element nested or two-element numeric vector.

Element 1: 0 for successful completion, ¯1 for an error.
Element 2: 0 for successful completion, error code on error.
Element 3: A character vector containing the name of the local host machine.
Element 4: A vector of character vectors. These are aliases for the host machine (if they exist).
Element 5: Host address family; should be 2 (inet).
Element 6: Length of address field in bytes (useful only in conjunction with an external routine).
Element 7: A vector of character vector address representations for the network addresses of the specified host machine.

GetHostName

Purpose:
Fetches the network name of the local host.

Syntax:
result ← ⎕ni 'GetHostName'

Arguments:
none

Result:
The result of GetHostName is a three-element nested or (on error) a two-element numeric vector.

Element 1: 0 for successful completion, ¯1 for an error.
Element 2: 0 for successful completion, error code on error.
Element 3: A character vector containing the name of the local host machine.

GetPeerName

Purpose:
Identifies the partner who is connected on a particular socket.

Syntax:
result ← *socket* `⎕ni 'GetPeerName'`

Arguments:
socket is the socket handle for the local socket.

Result:
The result of GetPeerName is a six-element nested vector, or (on error) a two-element numeric vector.

Element `1`: `0` for successful completion, `¯1` for an error.
Element `2`: `0` for successful completion, error code on error.
Element `3`: The socket number of the local socket.
Element `4`: Address family; should be `2` (inet).
Element `5`: Port number of partner.
Element `6`: A character vector representing the network address of the partner.

GetProtoByName

Purpose:
Fetches information about the named protocol

Syntax:
result ← `⎕ni 'GetProtoByName'` *protocol_name*

Arguments:
protocol_name is a character vector containing name of a well-known protocol. The relevant values for APL64 are `'tcp'`, `'udp'`, and `'ip'`.

Result:
The result of GetProtoByName is a five-element nested or two-element numeric vector.

Element `1`: `0` for successful completion, `¯1` for an error.
Element `2`: `0` for successful completion, error code on error.
Element `3`: A character vector containing the name of the chosen protocol.
Element `4`: A vector of character vectors. These are aliases for the protocol (if they exist).
Element `5`: The assigned protocol number for the specified protocol.

GetProtoByNumber

Purpose:
Fetch information about the named protocol

Syntax:
result ← `⎕ni 'GetProtoByNumber'` *protocol_number*

Arguments:
protocol_number is a the number for a known protocol.

Result:
The result of GetProtoByNumber is a five-element nested or two-element numeric vector.

Element `1`: `0` for successful completion, `¯1` for an error.
Element `2`: Always `0`.

Element 3: A character vector containing the name of the chosen protocol.
Element 4: A vector of character vectors. These are aliases for the protocol (if they exist).
Element 5: The number for the specified protocol (same as the argument).

GetServByName

Purpose:
Fetches information about the named service

Syntax:
result ← ⎕ni 'GetServByName' *service_name*

Arguments:
service_name is a character vector containing name of a service. Services are generally well known, but may depend on the machine. Examples include 'ftp' (file transfer protocol) and 'mail'. You can view this information using Notepad. You can probably find the file c:\winnt\system32\drivers\etc\services under Windows NT 2000.

Result:
The result of GetServByName is a six-element nested or two-element numeric vector.

Element 1: 0 for successful completion, ¯1 for an error.
Element 2: 0 for successful completion, error code on error.
Element 3: A character vector containing the name of the chosen service.
Element 4: A vector of character vectors. These are aliases for the service (if they exist).
Element 5: The service port number.
Element 6: The service protocol name.

GetServByPort

Purpose:
Fetch information about the service associated with the specified port number

Syntax:
result ← ⎕ni 'GetServByPort' *port_number*

Arguments:
port_number is the number for a well-known port.

Result: The result of GetServByPort is a six-element nested or two-element numeric vector.

Element 1: 0 for successful completion, ¯1 for an error.
Element 2: Always 0.
Element 3: A character vector containing the name of the chosen service.
Element 4: A vector of character vectors. These are aliases for the service (if they exist).
Element 5: The service port number.
Element 6: The service protocol name. (In some Windows systems, this may return an invalid address message.)

Purpose:
Identifies the host who is connected on a particular socket. This is useful when you have multiple host IP addresses on the server. You can also use it to determine which port numbers are associated with which sockets.

Syntax:
result ← *socket* ⎕ni 'GetSockName'

Arguments:
socket is the socket handle for the local socket.

Result:
The result of GetSockName is a six-element nested vector, or (on error) a two-element numeric vector.

Element 1: 0 for successful completion, ¯1 for an error.
Element 2: 0 for successful completion, error code on error.
Element 3: The socket number.
Element 4: Address family; should be 2 (inet).
Element 5: Port number of host.
Element 6: A character vector representing the network address of the host.

Purpose:
Return a value representing whether an option has been set for a socket or return the option value.

Syntax:
result ← *socket* `⎕ni 'GetSockOpt'` *option_level option_name*

Arguments:
socket is the socket handle for the local socket.
option_level is a character vector
`'sol_socket'` means the option is set at the application layer
`'ipproto_ip'` means the option is set at the network layer.
option_name is a character vector naming a specific option whose value you want to return; all option names
that start with *so_* are used with the *sol_socket* option level.
`'so_acceptconn'` indicates whether a stream socket is currently listening
`'so_broadcast'` indicates whether a socket may send broadcast datagrams
`'so_debug'` indicates whether OS implementation-specific debug information is enabled or disabled; this is irrelevant for some systems.
`'so_dontlinger'` returns the enabled/disabled state for "not lingering"
`'so_dontroute'` indicates whether a multi-homed host may bypass standard routing
`'so_error'` returns the current socket error value and clears it
`'so_keepalive'` returns the enabled state of a keepalive mechanism on a stream socket
`'so_linger'` returns the enabled/disabled state for "lingering" and the time interval
`'so_oobinline'` returns the enabled/disabled state of receiving TCP urgent data, which ordinarily would be received "out of band," within the normal data stream
`'so_reuseaddr'` returns the enabled/disabled state of reusing a local socket name
`'so_rcvbuf'` returns the size of the receive buffer
`'so_sndbuf'` returns the size of the send buffer
`'so_type'` returns an integer that corresponds to the value of a named constant for "type"
`'tcp_nodelay'` (the only name you may use with *ipproto_ip*) returns the enabled/disabled
state that is the inverse of the enabled/disabled state of the Nagle algorithm, a standard method for reducing network congestion.

Result:
The result of GetSockOpt is a two-element vector in the case of an error; otherwise, a three-element vector for most argument pairs; for *sol_socket* and *so_linger*, the result has four elements.

For all options,
Element `1`: `0` for successful completion, `¯1` for an error.
Element `2`: `0` for successful completion, error code on error.

Element `3`:
For *so_rcvbuf* and *so_sndbuf*: Buffer size in bytes for receive or send operations respectively.
For *so_type*: An integer equal to the value of the named constants `SOCK_STREAM` or `SOCK_DGRAM`; you can retrieve these values using the *GetConst* method.
For *so_error*: An integer error code; you can retrieve the error message using the *Error* method.
For *so_acceptconn*, *so_broadcast*, *so_debug*, *so_dontlinger*, *so_dontroute*, *so_keepalive*, *so_oobinline*, *so_reuseaddr*, and *so_tcpnodelay*: `0` for False, `1` for True

For *so_linger*:
Element `3`: The linger state: `1` for enabled, `0` for disabled
Element `4`: The linger interval in seconds.

Notes: See the description of the *SetSockOpt* method for more information on the impact of some of these values and some recommendations.

InUse

Purpose:
Returns the operational state of a list of sockets.

Syntax:
result ← *socket_list* `⎕ni 'InUse'`

Arguments:
socket_list is a vector of socket handles for which you want to query the state.

Result:
Returns a vector of state values. The result is the same shape as the socket list. Each value of the result indicates the state of the corresponding socket handle in the list. Possible state values are:

`0`: socket does not have an action pending on it.
`1`: socket currently has an action pending on it.
`¯1`: socket number is not a valid socket handle.

Listen

Purpose:
Places a socket into listening state. Optionally, you can specify the queue depth for incoming Connect requests on the socket.

Syntax:
result ← *socket* `⎕ni 'Listen'` *queue_depth*

Arguments:
socket is the socket handle for the local socket.
queue_depth is the number of incoming Connect requests to queue. The default value is `5`, which is also the maximum value.

Result:
The result of Listen is a two-element numeric vector.

Element `1`: `0` for successful completion, `¯1` for an error.
Element `2`: `0` for successful completion, error code on error.

Purpose:
Receives data over a connected socket.

Syntax:
result ← *socket* `⎕ni 'Recv'` *datatype flags data_length*

Arguments:
socket is the socket handle for the local socket.
datatype is the type of data to be received.
`'char'` means interpret the incoming data as characters in `⎕av`.
`'bool'` means interpret the incoming data as boolean.
`'int'` means interpret the incoming data as integer.
`'float'` interpret the incoming data as floating point.
`'apl'` means interpret the incoming data as an APL array, maintaining an internal APL data structure. This is the default datatype.

flags are special purpose flags. This field is empty by default.
`'peek'` means read the data without removing it from the system buffers.
`'oob'` means receive out-of-band data (urgent messages).
data_length is a non-negative integer specifying the length of data to read. If you specify zero, which is the default, it means read whatever is in the buffer. If you specify any positive value, it means read that number of elements of the datatype specified for character or numeric data. For the `'apl'` datatype, you should specify zero or omit this parameter. For Boolean data, you should read multiples of eight values. If you read another number, the remaining values in the last byte are lost. Out-of-band data can be sent only one byte at a time; hence these data are limited to character or boolean types.

Result:
The result of Recv is a three-element vector.

Element `1`: `0` for successful receive, `¯1` for an error, `1` for non-error condition that prevents successful receive.
Element `2`: `0` for successful receive; error code on error; for non-error condition, `0` if partner closed socket, `1` if socket is in use, `2` if action stopped.
Element `3`: Received data if successful; if there are no data available to satisfy a non-APL Recv in nonblocking mode, element `3` is an empty vector.

Purpose:
Receives data over a socket. The socket is unconnected. RecvFromis normally used to receive datagrams via udp.

Syntax:
result ← *socket* `⎕ni 'RecvFrom'` *datatype flags*

Arguments:
socket is the socket handle for the local socket.
datatype is the type of data to be received.
`'char'` means interpret the incoming data as characters in `⎕av`.
`'bool'` means interpret the incoming data as boolean.
`'int'` means interpret the incoming data as integer.
`'float'` interpret the incoming data as floating point.
`'apl'` is an allowable specification for this method, but you should not, under ordinary circumstances, use APL arrays with datagrams. In fact, some Sockets implementations limit datagrams to as little as four bytes. Since APL data structures are all larger than four bytes, the `'apl'` datatype will not work in such Sockets implementations. You should always specify one of the other datatypes when using RecvFrom.
flags are special purpose flags. This field is empty by default.
`'peek'` means read the data without removing it from the system buffers.
`'oob'` means receive out-of-band data (urgent messages).

Note that RecvFrom does not allow a data length specification. It reads the next datagram from the buffers for the length of the datagram up to a limit of `1024` bytes.

Result:
The result of RecvFrom is a seven-element nested vector, or (on error) a two-element numeric vector.

Element `1`: `0` for successful receive, `¯1` for an error, `1` for a non-error condition that prevents successful read.
Element `2`: `0` for successful receive; error code on error; for non-error condition, `0` if partner closed socket, `1` if socket is in use, `2` if action stopped.
Element `3`: Received data if successful.
Element `4`: Receiving socket handle.
Element `5`: Address family; should be `2` (inet).
Element `6`: Port number of sending socket.
Element `7`: A character vector representing the network address of the sending socket.

Caution:
Some Sockets implementations limit datagrams to as little as four bytes. APL data structures are all larger than four bytes, so the `'apl'` datatype will not work in such Sockets implementations.

Ref

Purpose:
References multiple properties or event handlers for a socket.

Syntax:
result ← *socket* ⎕ni 'Ref' *properties*

Arguments:
You can reference a single property or event-handler property by naming it as the right argument to ⎕ni with no other values. The Ref method allows you to reference multiple properties by providing a nested vector of character vectors containing the property names as the additional arguments.

Result:
The result of Ref is a nested vector of values in the same order that you listed the properties.

Select

Purpose:
Detects when the specified sockets are ready to be read or written.

Syntax:
result ← ⎕ni 'Select' *time socket_states*

Arguments:
time specifies the length of time in milliseconds to wait for detection of one of the specified states. A value of 0 for time returns without waiting. A value of ¯1 waits indefinitely until one of the selected states occurs or the function is interrupted.

socket_states are pairs of values. They can be either a vector of length two or an N-by-two array. The first value in each pair is a socket handle. The second value indicates what states to check for by summing the state values. The state values are: 1, 2, or 4.

- 1 Checks for a readable state.
- 2 Checks for a writeable state.
- 4 Checks for readable state for out-of-band data or for an error condition.

These values may be combined in a sum to check for multiple states simultaneously.

Result:
The result of Select is a three-element nested or (on error) a two-element numeric vector.

Element 1: 0 for successful execution, ¯1 for an error.
Element 2: 0 for successful execution, error code on error.
Element 3: An N-by-two array on successful execution.

The *result* array is normally the same shape as the *socket_states* argument. The first column contains the socket handles. The second column contains the sum of the values representing the states that the socket is in (of those requested).

- 1 If the socket is part of a connection, this indicates there are data available to read or that the partner has done a close on the socket. (In the latter case, Recv returns the result 1 0). If the socket is a listening socket, this indicates that a connect has been done by a partner and an accept will now complete the connection.
- 2 This indicates that the write buffers are empty and a write can be done to the socket.
- 4 This indicates a readable state for out-of-band data or an error condition.

If you specify a nonzero value for time, and the time limit is exceeded, Select returns all zeroes. If you specify a nonzero value for time with a query on multiple sockets, the system returns the current values for all the queried sockets whenever any state selected for one of the sockets occurs.

Note that the Select action is not affected by blocking modes; Select always runs until a change of state occurs or its timer expires. Windows messages will be processed during the Select; you can also interrupt it with, for example by using the Control method with `'stop'` as an argument or by signaling an interrupt.

Send

Purpose:
Sends data over a connected socket.

Syntax:
result ← *socket* `⎕ni 'Send'` *data datatype flags*

Arguments:
socket is the socket handle for the local socket.
data is the data to be transmitted.
datatype is the type of data to be transmitted.
`'char'` means send only character data. The system signals an error if the data are not character data.
`'bool'` means coerce data to boolean. The system signals an error if the data are not boolean valued.
`'int'` means coerce data to integer. The system signals an error if the data are not integer valued.
`'float'` means coerce data to floating point. The system signals an error if the data are not numeric.
`'apl'` means send data as an APL array, maintaining the internal APL data structure. This is the default datatype.
flags are special purpose flags. This field is empty by default.
`'dontroute'` means avoid using local routing tables (intended for network debugging).
`'oob'` means send message out-of-band (for ugent messages).

For Boolean data, you should send multiples of eight values. If you send another number, the last byte is padded. Out-of-band data can be sent only one byte at a time; hence these data are limited to character or boolean types.

Result:
The result of Send is a three-element numeric vector.

Element `1`: `0` for successful send, `¯1` for an error, `1` for non-error condition that prevents successful send.

Element `2`: `0` for successful send; error code on error; for non-error condition, `0` if partner closed socket, `1` if socket is in use, `2` if action stopped.

Element `3`: Length of data transmitted. For the APL datatype, the length is in characters. For character, integer, and float datatypes the length is in units of the datatype. For Boolean data, the length is bytes (groups of `8` boolean values).

Purpose:
Sends datagrams to a target address.

Syntax:
result ← *socket* `⎕ni 'SendTo'` *data datatype flags port inet_address*

Arguments:
socket is the socket handle for the local socket.
data is the data to be transmitted.
datatype is the type of data to be transmitted.
`'char'` means send only character data. The system signals an error if the data are not character data.
`'bool'` means coerce the data to boolean. The system signals an error if the data are not boolean valued.
`'int'` means coerce the data to integer. The system signals an error if the data are not integer valued.
`'float'` means coerce the data to floating point. The system signals an error if the data are not numeric.
`'apl'` is an allowable specification for this method, but you should not, under ordinary circumstances, use APL arrays with datagrams. Specify one of the other datatypes when using SendTo.

flags are special purpose flags. This field is empty by default.
`'dontroute'` means avoid using local routing tables (intended for network debugging).
`'oob'` means send the message out-of-band (for urgent messages).

port is the port number chosen for the application to which you are sending.
inet_address is the internet address of the target machine to which you are sending.

Result:
The result of SendTo is a three-element numeric vector.

Element `1: 0` for successful send, `¯1` for an error, `1` for non-error condition that prevents successful send.

Element `2`: `0` for successful send; error code on error; for non-error condition, `0` if partner closed socket, `1` if socket is in use, `2` if action stopped.
Element `3`: Length of data transmitted in units of the datatype (except for Boolean, which is multiples of `8` values).

Set

Purpose:
Sets multiple properties or event handlers for a socket.

Syntax:
result ← *socket* `⎕ni 'Set'` *properties*

Arguments:
You can set a single property or event-handler property by naming it and the value you want to assign as the right argument to `⎕ni`. You can set multiple properties by providing a nested vector of nested vectors containing the property names and their values as the additional arguments.

Result:
The result of Set is a two-element numeric vector.

Element `1`: `0` for successful set, `¯1` for an error; `1` for non-error condition.
Element `2`: `0` if successful; error code on error; `1` if socket is in use.

SetSockOpt

Purpose:
Set a value representing whether an option for a socket is enabled or disabled or set the option value.

Syntax:
result ← *socket* `⎕ni 'SetSockOpt'` *option_level option_name option_value*

Arguments:
socket is the socket handle for the local socket.
option_level is a character vector
`'sol_socket'` means the option is set at the application layer
`'ipproto_ip'` means the option is set at the network layer.
option_name is a character vector naming a specific option whose value you want to set; all option names that start with *so_* are used with the *sol_socket* option level.

option_value varies with the option; the name-value pairs are described below:

You set the following options with a Boolean: `1` to enable the option, `0` to disable it.

`'so_broadcast'` allows a socket to send broadcast datagrams

`'so_debug'` allows OS implementation-specific debug information; irrelevant for some systems.

`'so_dontlinger'` enables immediate return from closing a socket whether the socket is in blocking or nonblocking mode; queued data are sent and the TCP/IP stack attempts a graceful close in the background. This is the default.

`'so_dontroute'` enables a multi-homed host to bypass standard routing mechanisms in favor of sending a packet to an interface card most local to the destination host.

`'so_keepalive'` enables a stream socket to detect a virtual circuit failure on an idle connection and report the error; however, using this facility is not highly recommended.

`'so_oobinline'` enables receiving TCP urgent data within the normal data stream; these data would ordinarily be received "out of band." The advantage of this option is that you can read urgent data without setting the oob flag on the *Recv* or *RecvFrom* method; the disadvantage is that you must scan each byte to detect oob data.

`'so_reuseaddr'` enables the use of a local socket name on more than one socket; while using this option may allow you to avoid the "address in use" error, it can have unexpected results.

`'tcp_nodelay'` (use with the *ipproto_ip* option_level) sounds like a good thing. However, enabling this option disables the Nagle algorithm, a standard method for reducing network congestion that is recommended but not required by the WinSock specifications.

You set `'so_rcvbuf'` or `'so_sndbuf'` with an integer representing the number of bytes you want for the receive or send buffer respectively.

You set `'so_linger'` with two values: a Boolean to enable or disable it and an integer representing the number of seconds you want for the timeout value. Enabling this option disables *so_dontlinger* and vice versa.

If so_linger is enabled with zero timeout, closing a socket initiates a forceful close and returns immediately; if enabled with non-zero timeout, closing a socket attempts a graceful close until the timeout expires, then does a forceful close. If the socket is in blocking mode, the function blocks until the close operation completes (either successfully or unsuccessfully).

If so_linger is disabled, the system attempts a graceful close until a default timeout expires, then does a forceful close. However, the close operation occurs in the background, so the socket does not block in either mode. This is the recommended state.

There are three option names that are valid for *GetSockOpt* that you cannot use with *SetSockOpt*; they are *so_acceptconn, so_error*, and *so_type*.

Result:

The result of SetSockOpt is a two-element numeric vector.

Element `1`: `0` for success, `¯1` for an error.

Element `2`: `0` for success, error code on error.

Shutdown

Purpose:
Conditions a socket to be closed. This provides advance notice to the remote application before the socket is actually closed. This allows final communications to be cleared and any cleanup proccessing to be performed.

Syntax:
result ← *socket* `⎕ni 'Shutdown'` *how*

Arguments:
socket is the socket handle for the local socket.

how defines the shutdown behavior

- `0` means receives are disallowed.
- `1` means sends are disallowed.
- `2` means both sends and receives are disallowed.

Result:
The result of Shutdown is a two-element numeric vector.

Element `1:` `0` for successful completion, `¯1` for an error.
Element `2:` `0` for successful completion, error code on error.

Socket

Purpose:
Creates a socketand returns the socket handle.

Syntax:
result ← `⎕ni 'Socket'` *address_family type protocol*

Arguments:
address_family -- the only value currently allowed is `'inet'`, the family of internet protocols. This is the default.
type can be one of the following values:
`'stream'` designates a continuous connection. This is the default.
`'dgram'` Datagrams are used for short communication over non-continuous connections. This type can be used to exchange information prior to establishing a stream connection on another socket.

protocol specifies the protocol to be used.
`'tcp'` is the protocol used in conjunction with a stream connection. This is the default.
`'udp'` is the protocol used in conjunction with a datagram connection.

If you use `'ip'`, the system converts the protocol based on the socket type.

Result:
The result of Socket is a two- or three-element numeric vector.

Element `1:` `0` for success, `¯1` for an error.
Element `2:` `0` for success, error code on error.
Element `3:` New socket number, if successful.

Sockets

Purpose:
Returns a list of active socket handles.

Syntax:
result ← `⎕ni 'Sockets'`

Result:
The result of Sockets is a vector of socket handles for all current sockets.

WSAStartup

Purpose:
Performs the required initialization for sockets actions to take place. The system performs this action automatically before it performs the first socket action you specify. You can also invoke it yourself, for example, for use in debugging.

Syntax:
result ← `⎕ni 'WSAStartup'`

Result:
The result of WSAStartup is information about the WinSock DLL in use on the local host; the method name stands for WinSock Asynchronous Startup.

Socket Event-Handler Properties

For each event, you can specify an event handler, which is an APL expression. When the event occurs, the system executes the specified handler. Because of the way WinSock implements socket events, each time you set an event handler for a socket, the system resets any other event handler that was previously set. This may cause the previous event handlers to fire. To avoid such occurrences, you can use the Set method to set all handlers for a socket simultaneously.

Three new system variables, `⎕nevent`, `⎕nself`, and `⎕narg`, provide detailed information about the event and the socket it occured on.

Syntax:
result ← socket `⎕ni` event handler
result ← *socket* `⎕ni 'Set'` `(`*event1 handler1*`)` `(`*event2 handler2*`)…(`*event6 handler6*`)`

Arguments:
A two-element vector of character vectors. The first element is the event name. The second element is the APL expression that is the corresponding event handler.

Using the Set method, you can set any number of socket event-handlers in one call.

Result:
The result is a two element numeric vector.

Element `1:` `0` on successful completion, `¯1` on error.
Element `2:` `0` on successful completion, error code on error.

onAcceptNotify

Trigger:
A connection requesthas been received.

Remarks:
An Accept will now complete the connection, creating a new socket. The socket for which this event occurs must perform an Accept before another AcceptNotify event can occur for it. The connection requests are queued up to the limit specified in the Listen action. Note that the event that triggers this event handler occurs only for the listening socket (usually considered a server function).

onCloseNotify

Trigger:
A connection close has been received.

Remarks:
A Close should be done on the local socket. There is no re-enabling action for this event; you cannot effectively use this socket again.

onConnectNotify

Trigger:
A connection has been established.

Remarks:
Send and Recv will now work on this socket. There is no re-enabling action for this event; you can do multiple sends and receives, but you cannot form another connection. Note that the event that triggers this event handler occurs only for the socket on which the Connect action has been performed (usually considered the client function).

onOobNotify

Trigger:
An out-of-band character has arrived on this socket.

Remarks:
You can read out-of-band data with the `'oob'` flag set. The socket that receives this event must perform a Recv or RecvFrom with the appropriate flag before it can receive another OobNotify event. Out-of-band data can be sent only one byte at a time; if more data are sent, only one byte is marked as oob; however, using Recv with the flag set reads multiple oob characters if they are present.

onReadNotify

Trigger:
Data have arrived on this socket.

Remarks:
You can now read these data. The socket that receives this event must perform a Recv or RecvFrom before it can receive another ReadNotify event.

onWriteNotify

Trigger:
Network system buffers are available.

Remarks:
You can now send data on this socket. The socket that receives this event must perform a Send or SendTo before it can receive another WriteNotify event.

Socket Properties

blockingmode

Description:
Character vector that specifies the mode of this socket. The valid values are `'blocking'`, `'fakeblocking'`, or `'nonblocking'`. The default value in Windows is fakeblocking.

Remarks:
You can set this property directly or change the blocking mode by using the Control method. If you set event handlers on a socket, the system automatically puts it into in nonblocking mode. Fakeblocking mode is meaningful only when you are doing a Recv or a Send; if you set fakeblocking, other actions are nonblocking. See the "Modes of Sockets" section above for a description of the modes.

condition (read-only)

Description:
Character vectors that describes the condition of the socket. The possible values are `'open'`, `'bound'`, `'listening'`, or `'connected'`.

Remarks:
A socket is open after you create it. It is bound or listening after you perform the corresponding actions. A socket is connected after a successful connect or accept.

events (read-only)

Description:
Nested vector of character vectors that lists the events for which you can write `⎕ni` event handlers.

Remarks:
You can reference this property directly; it is a read-only property. The event-handler properties have the same names with an on prefix; for example, the AcceptNotify event triggers the onAcceptNotify event handler. You can determine which events have handlers defined by referencing the state property of a socket.

family (read-only)

Description:
Character vector that specifies the address family that is an argument to the Socket method when you create a socket. The only allowed value is `'inet'`.

Remarks:
When family is returned as the result of a method, it is an integer value; the value `2` should equate to inet.

methods (read-only)

Description:
Nested vector of character vectors that lists the methods you can use with `⎕ni`.

Remarks:
This property provides a list of the methods you can use with `⎕ni`; you can reference it directly.

properties (read-only)

Description:
Nested vector of character vectors that lists all of the properties for this socket.

Remarks:
This property provides a list of the properties you can use for a socket; it includes any user-defined properties that are currently set on the socket you specify as a left argument.

protocol (read-only)

Description:
Character vector that specifies the protocol that is an argument to the Socket method when you create a socket. The possible values are `'tcp'`, `'udp'`, or `'ip'`.

Remarks:
You use the value `'tcp'` for stream communication. You use `'udp'` (Universal Datagram Protocol) for datagrams. If you use `'ip'`, the system converts the protocol based on the socket type.

state (read-only)

Description:
Nested matrix of properties and their values for this socket. This matrix does not contain the static properties, such as properties, or state itself.

Remarks:
This property is a snapshot of the specification of a socket. When you reference it, you see the properties that have changing values. You cannot change this property directly, but the system updates it every time you change a property setting for a socket.

type (read-only)

Description:
Character vector that specifies the socket type that is an argument to the Socket method when you create a socket.

Remarks:
The possible values are `'stream'` and `'dgram'`.

User-defined Properties (Δ`udp`) Associated with Sockets

Just as you can create user-defined properties associated with APL64 Windows objects, you can create analogous user-defined properties associated with sockets. Simply name the property using the delta symbol (∆) as the first character and follow the rules for naming APL variables. You can assign any value.

These variables are stored in Windows memory; they are not part of the APL workspace. If you reference a user-defined property before you have assigned a value, the system, by default, returns a scalar integer zero. If you assign a scalar zero to the property, it disappears from Windows memory. If you want to maintain the presence of this property for a given socket with the value zero, you can assign a vector zero. The property also disappears when you close the socket with which it is associated.

The syntax for assigning a user-defined property is
result ← *socket* `⎕ni '∆`*name*`'` *value*

System Variables Associated with Socket Events

Argument for a network socket event (`⎕narg`)

Purpose:
This session-related system variable contains the argument for a socket event. When a socket event occurs, `⎕narg` contains either an integer error code or zero.

Network socket event (`⎕nevent`)

Purpose:
This session-related system variable contains the name of the current socket event during a callback.

Current network socket (`⎕nself`)

Purpose:
This session-related system variable contains the number (handle) of the current socket during a callback. The value when no callbacks are pending is an empty vector unless you have assigned it when no callbacks were pending. You can assign a value with the number you want to designate as the current socket. Any actions you take with `⎕ni` using an implied left argument apply to the current socket.