EXHIBIT 64 PART 5

Handling Requests When the Transport Is Active

While the transport is actively sending or receiving data in the background, the user might request another send or receive operation from the In/Out Box. One way to handle such requests is to queue them up and append them to the current communication transaction or to start another connection when the transport is finished.

You can use the transport method `QueueRequest` to queue up requests for sending or receiving, if the transport already has an active communication session in progress. Call `QueueRequest` from the `SendRequest` or `ReceiveRequest` method, whichever one you receive as a result of a user request.

Depending on how you call it, you can make `QueueRequest` append the new request to a request in progress or start another connection when the current request is finished. To append the new request to one in progress, for the first parameter, specify the request frame of a request already in progress. A request frame is the frame passed to `SendRequest` or `ReceiveRequest` to begin the request in progress. The second parameter is the new request frame.

The following is an example of a `SendRequest` method in which `QueueRequest` is called to append the new request to the one in progress.

```
// SendRequest method
func (newRequest)
begin
if status <> 'idle then // check if I'm active
   // append to current request
   :QueueRequest(currentRequest, newRequest);
else
   // do a normal send here
end,
```

When a new request is appended to an in-progress request, items from the new request are returned from the `ItemRequest` method after all items from the in-progress request are exhausted. In this way, new items are sent as part of the current communication session.

To queue a new request so that the transport finishes its current transaction before beginning a new one, specify a symbol for the first parameter of `QueueRequest`. The symbol should be the name of a method that you want the system to call when the transport state returns to idle. Usually this is another `SendRequest` or `ReceiveRequest` method. The following is an example of a `SendRequest` method in which `QueueRequest` is called to defer a new request until the transport returns to the idle state.

```
// SendRequest method
func (newRequest)
begin
if status <> 'idle then // check if I'm active
   // wait for idle and then call SendRequest again
   :QueueRequest('SendRequest, newRequest);
else
   // do a normal send here
end,
```

Canceling an Operation

The system sends the `CancelRequest` message to the transport when the user cancels the current transaction or for other reasons, such as when the system wants to turn off. This method must be defined by all transports.

When a transport receives this message, it must terminate the communication operation as soon as possible.

The `CancelRequest` method should return non-`nil` if it is OK to turn off power immediately after this method returns, or `nil` if it is not OK to turn off power immediately. In the latter case, the system waits until your transport returns to the idle state before turning off. This allows you to send an asynchronous cancel request to your communication endpoint, for example, and still return immediately from the `CancelRequest` method. When you receive the callback message from your endpoint cancel request confirming cancellation, use the `SetStatusDialog` method to set the transport status to idle to alert the system that it is OK to turn off.

Obtaining an Item Frame

The system sends the `NewItem` message to the transport to obtain a new item frame to make a new In/Out Box entry.

This method is supplied by the `protoTransport`, but should be overridden by your transport to fill in extra values your transport uses. If you override this method, you must first call the inherited `NewItem` method, as shown in the example below. The item frame returned by the `NewItem` method should contain default values for your transport.

The item frame returned by the default method supplied in `protoTransport` is not yet a soup entry. The `item.category` slot is initialized to the `appSymbol` slot in your transport. For more information on the `item` frame, see the section "Item Frame" beginning on page 22-2.

The `NewItem` message is sent to your transport during both send and receive operations. When the user sends an item, the system sends the `NewItem` message to the transport to create a new In/Out Box entry before opening a routing slip

for the item. This allows the transport an opportunity to add its own slots to the item frame.

Most transports will want to add a `fromRef` slot to the item frame. This slot must contain a name reference that identifies the sender. This information is usually extracted from the sender's current owner card, or persona. You shouldn't just use the value of `GetUserConfig('currentPersona)` because it is simply an alias to a names file entry. Instead, construct a name reference from this value. For example:

```
persona := GetUserConfig('currentPersona);
dataDef := GetDataDefs(addressingClass);
fromRef := dataDef:MakeNameRef(persona,addressingClass);
```

Most transports will want to extract and send only the needed information from the `fromRef` name reference. For example, an e-mail transport would typically just extract the sender name and e-mail address from the name reference and send them as strings. One method of name reference data definitions that you can use to extract useful information from a name card includes `GetRoutingInfo`. Here is an example of using this method:

```
// extract just routing info using GetRoutingInfo
routeInfo:= datadef:GetRoutingInfo(fromRef);
// returns an array like this:
[{name: "Chris Smith", email: "cbsmith@apple.test.com"}]
```

The `GetRoutingInfo` method returns an array of at least one frame that has at least a `name` slot containing a string. Depending on the `addressingClass` slot passed to the `GetDataDefs` function, the returned frame also contains other information particular to the type of address used for the transport. In the example above, the frame also contains an `email` slot with an e-mail address.

If you want to add other slots to the `fromRef` frame, you can either define your own name reference data definition and override the method `GetItemRoutingFrame` (called by `GetRoutingInfo`), or add the slots you want to the `fromRef` frame by extracting them from the original name reference with the `Get` method. For example:

```
// use Get to extract info from certain slots
fromRef.myInfo := dataDef:Get(fromRef,'myInfo,nil);
```

Note that a sender may have multiple e-mail addresses and the transport should set the e-mail address in the `fromRef` frame to the one that is appropriate to itself. For example, for an internet e-mail transport, you would typically set the `fromRef`

e-mail address to the sender's internet address. Here's an example of code that sets the appropriate e-mail address in the `fromRef` object:

```
owner:=ResolveEntryAlias(GetUserConfig('currentPersona));
   if owner and GetRoot().cardfile then begin
      addrs := GetRoot().cardfile:BcEmailAddress(owner,
                              ['|string.email.internet|]);
      if addrs then
         fromRef := clone(addrs[0]);
   end
```

You can find a description of `BcEmailAddress` and other similar functions that extract information from Names soup entries in "Names Functions and Methods" (page 16-5) in *Newton Programmer's Reference*.

If, instead of extracting the address and sending it as a string, your transport sends addressing information as a frame, like the beam transport, you must remove any soup entry aliases from the name reference before it is transmitted. You can do this by using the name reference data definition method `PrepareForRouting`, as follows:

```
// strip the aliases from a name ref
fromRef := datadef:PrepareForRouting(fromRef);
```

In general, however, you should not send all the information in a user's persona with a message, since it can include personal or confidential information such as credit card numbers.

For more information about name references and the methods of name reference data definitions, see the section "Creating a Name Reference" beginning on page 21-27, and "Name References" (page 5-1) in *Newton Programmer's Reference*.

The following is an example of how to override the `NewItem` method during a send operation to add a `fromRef` slot:

```
// a sample overridden NewItem method
mytransport.NewItem := func(context) begin
   // first call inherited method to get default frame
   local item := inherited:NewItem(context);

   // get sender info and insert fromRef slot
   local persona:= GetUserConfig('currentPersona);
   local dataDef := GetDataDefs(addressingClass);
```

```
    if dataDef then
        begin
        item.fromRef := dataDef:MakeNameRef(persona,
                                            addressingClass);
        // add other slots or extract routing info here
        end;
    item;
end;
```

During a receive operation, the transport itself must invoke the `NewFromItem` method to get a new In/Out Box item frame. This method copies most slots from the received item to the new In/Out Box item frame. Additionally, it inserts the `destAppSymbol` slot value (if included) in the received frame into the `appSymbol` slot in the new frame.

Finally, the transport should call `ItemCompleted` to register the item in the In Box (see the following section).

Completion and Logging

After your transport finishes processing an item (either sending or receiving, with or without errors), you must send the transport the message `ItemCompleted`. This method must be used when an item is altered in any way. It performs several operations, including setting the state and error status of the item; sending the `ItemCompletionScript` callback message to the application; handling error conditions; and saving, logging, or deleting the item, depending on the logging preferences.

Send the `ItemCompleted` message only after your transport has completely processed an item. If you send this message before you know that the item was delivered successfully, for example, there's a possibility that the item will be lost.

If `ItemCompleted` was called as the result of an error, it calls `HandleError` to translate the error code and notify the user. If you want to perform your own error notification, you can override the `HandleError` method.

Note that the `ItemCompleted` method in `protoTransport` sends the `ItemCompletionScript` callback message to the application only if the item contains a `completionScript` slot that is set to `true`. You must set this slot if you want the callback message to be sent. For more information on `ItemCompletionScript` see *Newton Programmer's Reference* (page 18-33).

To perform logging, `ItemCompleted` sends your transport the message `MakeLogEntry`, passing a log entry to which you can add slots. The `protoTransport` object includes a default `MakeLogEntry` method, but you should override this method to add transport-specific slots to the log entry.

The default method simply adds a `title` slot to the log entry. The `GetItemTitle` transport method is called to get the title.

Storing Transport Preferences and Configuration Information

Transports can store user-configurable preferences and other configuration information. Typically, you store several chunks of data that correspond to individual preferences or other kinds of configuration information that you want to save for your transport. You must use the transport methods `GetConfig` and `SetConfig` to retrieve and set configuration information for your transport.

Default preferences for a transport are set by the `defaultConfiguration` slot in the transport object. This slot holds a frame containing values that correspond to items in a preferences slip that lets the user set preferences for your transport. For more information about displaying a preferences slip to the user, see the section "Providing a Preferences Template" beginning on page 22-33.

If you don't want to use this preferences dialog or the setting of the `defaultConfiguration` slot in `protoTransport`, override the initial setting by creating your own default preferences frame and including it in the `defaultConfiguration` slot of your transport object. Note that you can't use a `_proto` slot in the default frame since the contents of the `defaultConfiguration` slot are stored in a soup and `_proto` slots can't be stored in soup entries.

Extending the In/Out Box Interface

Your transport can extend the In/Out Box interface if items the transport handles can be viewed in the In/Out Box. You can add additional actions to the In/Out Box Tag picker in the In/Out Box. The In/Out Box Tag picker is displayed when the user taps the Tag button in the In/Out Box, as shown here:

The In/Out Box Tag picker includes only the Put Away and Log items by default. You can add other transport-dependent items by implementing the `GetTransportScripts` method. For example, the picker shown above includes Reply and Forward items added by an e-mail transport to let the user perform those operations on e-mail directly in the In/Out Box.

When the user taps the Tag button, the system sends your transport the `GetTransportScripts` message, if you've implemented it. This method must return an array of frames that describe new items to be added to the In/Out Box Tag picker. The array is exactly the same as the `routeScripts` array that adds items

to the Action picker in an application. Here is an example of a return value that adds two items to the picker:

```
[ {title: "Reply", // name of action
icon: ROM_RouteReply, // picker icon
// called if action selected
RouteScript: func(target, targetView) begin ... end,
},
{title: "Forward", // name of action
icon: ROM_RouteForward, // picker icon
// called if action selected
RouteScript: func(target, targetView) begin ... end,
} ]
```

The `RouteScript` slot contains a method that is called if the user selects that item from the picker. Alternatively, in the `RouteScript` slot you can specify a symbol identifying a transport method, and then supply your transport symbol in another slot named `appSymbol`.

For more detailed information about the items in the array, see the section "Providing Application-Specific Routing Actions" beginning on page 21-22.

For the `icon` slot of each frame in the array, you can specify an icon that appears next to the name of the action in the picker. There are standard bitmaps available in the ROM for the following actions:

- reply, `ROM_RouteReply`
- forward, `ROM_RouteForward`
- add sender to the Names application, `ROM_RouteAddSender`
- copy text to Notes application, `ROM_RoutePasteText`

If you are adding one of these actions, use the indicated magic pointer constant for the standard bitmap, to keep the interface consistent among transports.

Also, when the user taps the Tag button, the system sends your transport the `CanPutAway` message, if you've implemented it. This method allows your transport to add a put away option for the item to the Put Away picker. This hook allows a transport to put away items that could not otherwise be put away. Remember that applications (or transports) that need to put away items must implement the `PutAwayScript` method.

Whenever an item belonging to your transport is displayed in the In/Out Box, the In/Out Box also sends your transport the `IOBoxExtensions` message. This hook lets your transport add functionality to items in the In/Out Box by adding to the list of view definitions available for an item.

Application Messages

Applications can send messages directly to a single transport or to all transports by using the `TransportNotify` global function. This mechanism serves as a general way for applications to communicate with transports. Here is an example of using this function:

```
TransportNotify('_all,'AppOpened,[appSymbol])
```

The In/Out Box uses this mechanism to send three different messages to transports: `AppOpened`, `AppClosed`, and `AppInFront`. The `AppOpened` message notifies the transport that an application has opened and is interested in data from the transport. The In/Out Box sends this message to all transports when it opens. This method is not defined by default in `protoTransport` since it's transport-specific. If you want to respond to the `AppOpened` message, you must define this method in your transport.

This message is designed to support applications that might poll for data, such as a pager. For example, when the application is open, it can notify the transport with this message so that the transport can poll more frequently (and use more power) than when the application is closed. Another use might be for an application to notify a transport that automatically makes a connection whenever the application is open.

The `AppClosed` message notifies the transport that an application has closed. The In/Out Box sends this message to all transports when it closes. Again, this method is not defined by default in `protoTransport` since there is no default action—it's transport-specific. If you want to respond to the `AppClosed` message, you must define this method in your transport.

Note that more than one application can be open at a time in the system. If you want your transport to do something like disconnect when it receives this message, keep track of how many times it's received the `AppOpened` message and don't actually disconnect until it receives the same number of `AppClosed` messages.

The `AppInFront` message notifies the transport of a change in the frontmost status of an application—either the application is no longer frontmost, or it now is. The In/Out Box sends this message to all transports when another application is opened in front of the In/Out Box view, or when the In/Out Box view is brought to the front. Note that the `AppInFront` message is not sent when an application is opened or closed, so you need to check for the `AppOpened` and `AppClosed` messages to catch those occurrences.

Again, this method is not defined by default in `protoTransport` since there is no default action—it's transport-specific. If you want to respond to the`AppInFront` message, you must define this method in your transport. Not that this method is used only in special circumstances and is not needed by most transports.

Error Handling

The default exception handling method implemented by `protoTransport` is `HandleThrow`, which catches and handles exceptions resulting from any supplied transport methods such as `SendRequest` and `ReceiveRequest`. You must provide your own exception handler for any methods that you define, or you can pass them to `HandleThrow`, as follows:

```
try begin
   ... // do something
   Throw();
onException |evt.ex| do
   :HandleThrow();
end
```

When handling an exception, `HandleThrow` first calls `IgnoreError` to give your transport a chance to screen out benign errors. If `IgnoreError` returns `true`, `HandleThrow` returns `nil` and stops.

Assuming the error is not rejected by `IgnoreError`, `HandleThrow` next checks to see if an item is currently being processed. If so, it sends your transport the `ItemCompleted` message and returns `true`. Note that `ItemCompleted` calls `HandleError` to display an error alert to the user. If no item is currently being processed, `HandleThrow` sends the `HandleError` message itself to display an error alert.

The `HandleError` method calls `TranslateError` to give your transport a chance to translate an error code into an error message that can be displayed to the user. If your transport can't translate the error (for example, because it's a system-defined error) you should simply call the inherited `TranslateError` method, which handles system-defined errors.

Power-Off Handling

The `protoTransport` object registers a power-off handler with the system whenever the transport is not in the idle state. If the system is about to power off, this power-off handler sends the transport the `PowerOffCheck` message.

The default `PowerOffCheck` method in `protoTransport` displays a slip asking the user to confirm that it is OK to break the connection. Then, when the power is about to be turned off, the system sends the transport the `CancelRequest` message and waits until the transport is idle before turning the power off.

You can override the default `PowerOffCheck` method if you wish.

There is also a power-on handler that sends a `CancelRequest` message to the transport when the system turns on after shutting down unexpectedly while the transport is active.

Providing a Status Template

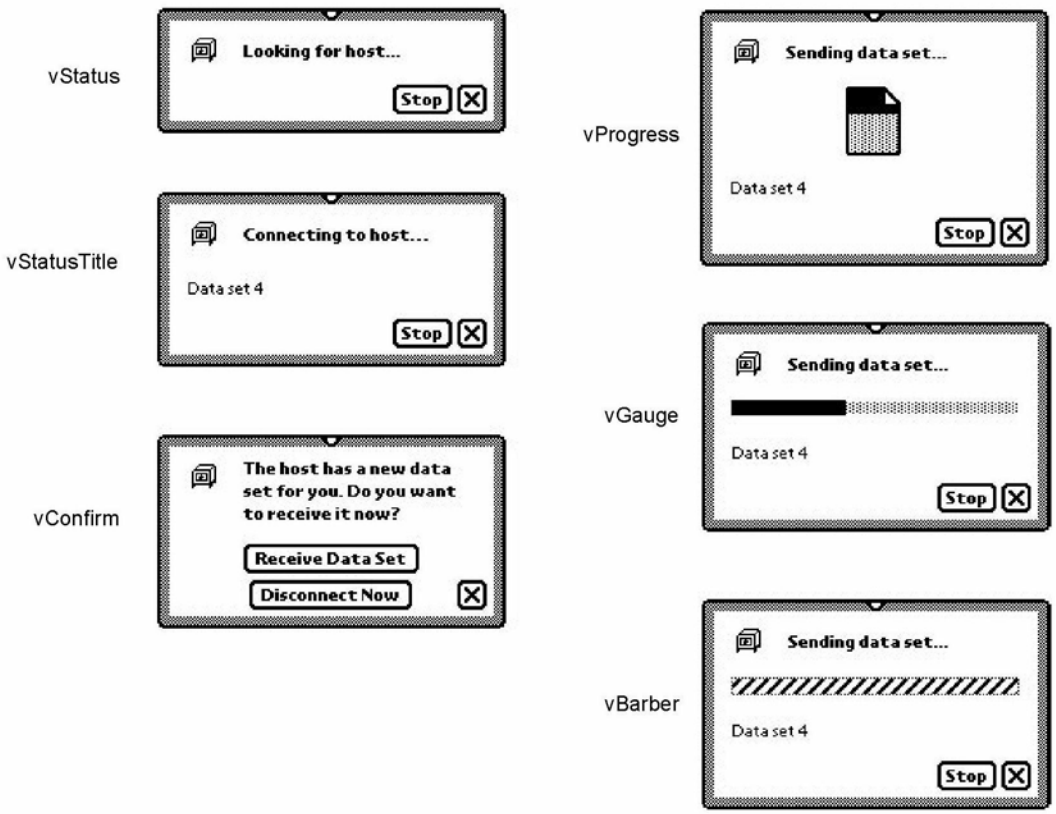
A status template for a transport is based on the proto `protoStatusTemplate`. The status template displays status information to the user. A transport should generally display a status view whenever it is sent the `ReceiveRequest` or `SendRequest` messages.

You probably won't need to create your own status template. The `protoTransport` is defined with a default status template named `statusTemplate` (based on `protoStatusTemplate`), which includes six predefined subtypes, described in Table 22-1 and shown in Figure 22-1. Each subtype consists of a set of child views that are added to the base status view. The base status view includes a transport icon and a close box, to which different child views are added, depending on the specified subtype name.

Table 22-1 Status view subtypes

Subtype name	Important values	Description
`vStatus`	`statusText` (top string)	A view that incorporates a status line. This is the default subview created by `SetStatusDialog`.
`vStatusTitle`	`statusText` (top string), `titleText` (lower string)	A view that incorporates a status line and a line for the item's title.
`vConfirm`	`statusText` (top string), `primary` (lower-button text and method: {`text:` *string*, `script:` *function*}), `secondary` (upper-button text and method: {`text:` *string*, `script:` *function*})	A view that has space for three lines of text, and two buttons. This view is suitable for situations where the user must choose between two options.
`vProgress`	`statusText` (top string), `titleText` (lower string), `progress` (integer, percentage completed)	A view that incorporates status and title lines, as well as a dog-eared page image that fills from top to bottom, based on the progress of the transfer.
`vGauge`	`statusText` (top string), `titleText` (lower string), `gauge` (integer, percentage completed)	A view that incorporates status and title lines, as well as a horizontal gauge that fills from left to right, based on the progress of the transfer.
`vBarber`	`statusText` (top string), `titleText` (lower string), `barber` (set to `true`)	A view that incorporates status and title lines, as well as a horizontal barber pole-like image that can be made to appear to move from left to right.

Figure 22-1 Status view subtypes



Each child view included in a subtype has one important value that controls the appearance of that child element. For example, the `vProgress` subtype consists of three child views that have these important values: `statusText` (the string displayed at the top of the view), `titleText` (the string displayed at the bottom of the view), and `progress` (an integer indicating the percentage of the page that should be shown filled with black). The important values for each subtype appear in Table 22-1. This information is necessary for use in the `SetStatusDialog` method.

A transport specifies the use of a subtype in the status view by passing the subtype name in the *name* parameter to the `SetStatusDialog` transport method. Transports can dynamically switch from one status subtype to another without closing the status view, and can easily update the contents of the status view as well (for example, updating the progress indicator).

Using this set of predefined status templates gives all transports a similar user interface and matches the use of other status views throughout the system.

For more detailed information on `protoStatusTemplate` and the predefined subtypes, refer to Chapter 17, "Additional System Services."

Controlling the Status View

Your transport should display a status view to the user whenever it is engaged in a lengthy activity such as sending or receiving data. In general, this means you must display a status view as part of the processing you do whenever you receive a `SendRequest` or `ReceiveRequest` message that results in the transmission of data.

To display a status view, use the transport method `SetStatusDialog`. If the `autoStatus` slot of the transport preferences frame is `true`, the status view opens automatically when you send the `SetStatusDialog` message with a status other than `'idle` as the first parameter. If the status view is already open, `SetStatusDialog` updates the status view with the new status information you pass to it. If `autoStatus` is `nil`, the status view does not open because the user has set a preference that it not be shown.

Here is an example of how to use the `SetStatusDialog` method:

```
:SetStatusDialog('Connecting, 'vStatus, "Looking for host...");
```

The `SetStatusDialog` method takes three parameters. The first is a symbol indicating what the new transport status is. This is typically one of the slots in the `dialogStatusMsgs` frame, such as `'Connecting`, or `'Idle`. The second parameter is the name of the status subtype you want to use. You can specify one of the built-in subtypes described in the previous section, or the name of a custom subtype that you have constructed. (You specify the value of the `name` slot in the subtype template.) For information on constructing custom `protoStatusTemplate` view subtypes, see Chapter 17, "Additional System Services."

The third parameter is typically a frame that contains one or more slots of values. Each slot corresponds to a single child view within the subtype you are using, and it sets the value of that child view. A slot name is the value of the `name` slot in the child view you are setting, and the value is whatever important value that type of view uses. The slot names and the expected values for the predefined status subtypes are listed in the "Important values" column in Table 22-1.

The following examples show how you'd use the `SetStatusDialog` method to set the different status subtypes to create the status views shown in Figure 22-1.

```
// vStatus subtype
:SetStatusDialog('Connecting, 'vStatus, "Looking for host...");

// vStatusTitle subtype
:SetStatusDialog('Connecting, 'vStatusTitle, {statusText: "Connecting
to host...", titleText: "Data set 4"});

// vConfirm subtype
:SetStatusDialog('Confirming, 'vConfirm, {statusText: "The host has a
new data set for you. Do you want to receive it now?",
secondary:{text:"Receive Data Set", script: func() ... },
primary:{text:"Disconnect Now", script: func() ... }});

// vProgress subtype
:SetStatusDialog('Sending, 'vProgress, {statusText: "Sending data
set...", titleText: "Data set 4", progress:40});

// vGauge subtype
:SetStatusDialog('Sending, 'vGauge, {statusText: "Sending data
set...", titleText: "Data set 4", gauge:40});

// vBarber subtype
:SetStatusDialog('Sending, 'vBarber, {statusText: "Sending data
set...", titleText:"Data set 4", barber:true});
```

Once the status view is open, each time you call `SetStatusDialog`, the system closes and reopens all its child views. This is fairly fast, but if you just want to update a progress indicator that is already visible in the subtypes `vProgress`, `vGauge`, or `vBarber`, you can use the alternate method `UpdateIndicator`. This `protoStatusTemplate` method updates the progress indicator child of the status view: the page image for the `vProgress` subtype, the horizontal bar for the `vGauge` subtype, and animation of the barber pole for the `vBarber` subtype.

For example, you'd use `UpdateIndicator` to update the `vGauge` subtype as follows:

```
statusDialog:UpdateIndicator({name:'vGauge, values:{gauge: 50,}});
```

Note that the frame of data you pass to `UpdateIndicator` consists of two slots, `name` and `values`, that hold the name of the subtype and the value(s) you want to set, respectively. The `values` slot is specified just like the *values* parameter to `SetStatusDialog`.

Also, note that `UpdateIndicator` is a method of `protoStatusTemplate`, and you need to send this message to the open status view. A reference to the open status view is stored in the `statusDialog` slot of the transport frame, so you can send the message to the value of that slot, as shown above.

The vBarber subtype shows a barber pole-like image, but it doesn't animate automatically. To make it appear to move, use the UpdateIndicator method in a ViewIdleScript method, as shown here:

```
// create the initial vBarber status view
:SetStatusDialog('Sending, 'vBarber, {statusText: "Sending data
set...", titleText:"Data set 1", barber:true});

...
// set up the status view data frame
statusDialog.barberValueFrame:={name:'vBarber,values:{barber:true}};
...
// set up the idle script
statusDialog.ViewIdleScript:= func()
   begin
   :UpdateIndicator(barberValueFrame); // spin the barber
   return 500; // idle for 0.5 seconds
   end;
...
// start the idler
statusDialog:Setupidle(500);
```

If the autoStatus slot of the transport preferences frame is true, the status view closes automatically when you send the SetStatusDialog message with 'idle as the first parameter.

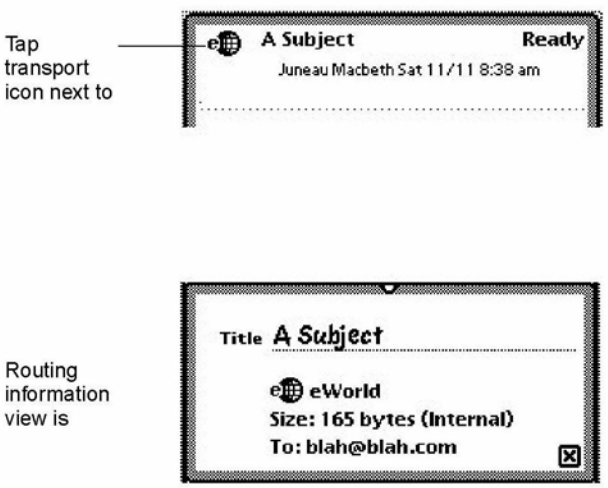
You can force the status view to close manually by sending the transport the message CloseStatusDialog. However, the next time you send the message SetStatusDialog with a state other than 'idle as the first parameter, the dialog reopens.

Providing a Routing Information Template

When viewing an item in the In/Out Box, the user can tap the transport icon to the left of the item title to display a view that gives routing information about the item. For example, for a fax item, the fax phone number is displayed; for a mail item, the e-mail header is shown. Figure 22-2 (page 22-26) shows an example of a routing information view.

You should create a template for a routing information view for your transport, using protoTransportHeader. If you don't specify a header view, your transport uses the default view, which displays the item title, the transport icon and name, and the item's size in the In/Out Box soup (the first three elements in the picture above).

Figure 22-2 Routing information view



In your transport object, store a reference to your routing information template in the `transportInfoForm` slot.

To add your own information to the routing information view, you can supply a `BuildText` method. From `BuildText`, call the `AddText` method for each additional line of text you want to add below the existing elements. Alternatively, you can add child views to the routing information view.

If you do add additional lines or views to the routing information view that cause it to increase in height, you must also set the `addedHeight` slot in the routing information view or in your `BuildText` method (or anywhere before the inherited `ViewSetupFormScript` method is called). In this slot, specify the number of pixels by which you are increasing the height of the view.

The header view may include editable fields. If the user changes something in an editable field, you probably want to know about it so that you can save the new information or perform other operations. The `InfoChanged` message is provided for this purpose. This message is sent to whatever object you designate when the header view is closed.

Providing a Routing Slip Template

A transport uses a routing slip when sending an item in order to get all the information necessary to transmit the item. Since the user interface for the routing slip is provided by the transport, the application does not need to know anything about what is required to send the item.

Store a reference to your routing slip template in the `routingSlip` slot in your transport object.

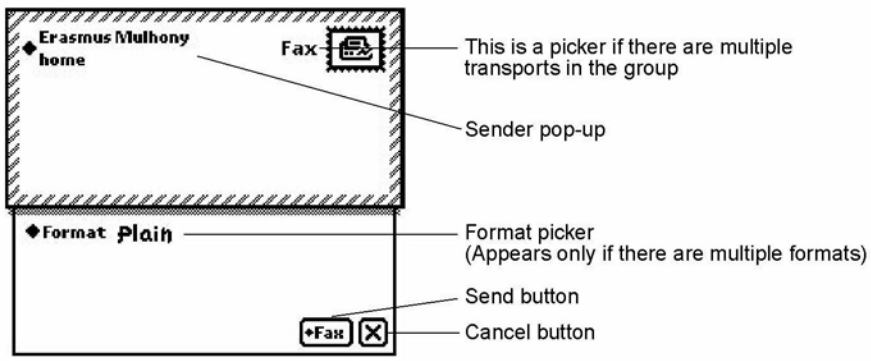
Use the `protoFullRouteSlip` template, described in the following section, to create a routing slip.

One additional proto for use in routing slips is described in the section "Using protoAddressPicker" (page 22-31).

Using protoFullRouteSlip

This routing slip proto already includes most of the elements required in a routing slip. Figure 22-3 shows an example of this proto. For a complete description of the slots and methods in this proto, see "protoFullRouteSlip" (page 19-38) in *Newton Programmer's Reference*.

Figure 22-3 `protoFullRouteSlip` view



The transport name and stamp icon in the upper-right corner of the routing slip are automatically supplied. They are based on the *transport*`.actionTitle` and *transport*`.icon` slots.

The format picker child in `protoFullRouteSlip` provides the picker list for choosing among multiple formats. The current format is initially displayed. The picker provides for opening an auxiliary view if one is associated with the current format. This child view uses the `currentFormat` slot in the item (the `fields.currentFormat` slot in the routing slip), a list of routing formats compatible with the item, and the `activeFormat` slot in the routing slip to set up the picker with the correct choices. These slots are set up by the system.

When the user picks another format, the `activeFormat` slot is updated, which changes the format choice shown next to the label. Additionally, the `SetDefaultFormat` message is sent to the application, and `currentFormat` in

the item is updated. The format picker also sends the `SetupItem` message to the format itself. If the format contains an `auxForm` slot, the view specified in the `auxForm` slot opens when the format is selected.

The sender pop-up child view allows the sender of the item to select a different owner persona or worksite from a picker, which might affect how the owner's name and address appear and how the item is sent. For example, if you choose a worksite location with a different area code from your company worksite, and send a fax to your company, the system automatically inserts a "1" and the company area code before the phone number, which it wouldn't do if you told the system you were at a location in that area code.

The default owner name (or persona as it is sometimes called) shown by this picker is the one corresponding to the last-used owner name for a routing operation. The default worksite for the owner is the one corresponding to the last worksite used for a routing operation, or the setting of the home location in the Time Zones application (whichever was done last). Note that additional owner names and worksites can be created by users in the Owner Info application.

The Send button child in `protoFullRouteSlip` provides the button that actually sends the item to the Out Box, and can also activate the transport. When tapped, the button may display a picker with the choices "Now" and "Later," or it may immediately send the item now or later. Its operation depends on the user preference setting of the `nowOrLater` slot in the preferences configuration frame described in Table 19-1 (page 19-7) in *Newton Programmer's Reference*, and on the return value of the transport `ConnectionDetect` method, which can force the button to send now or later without displaying a picker.

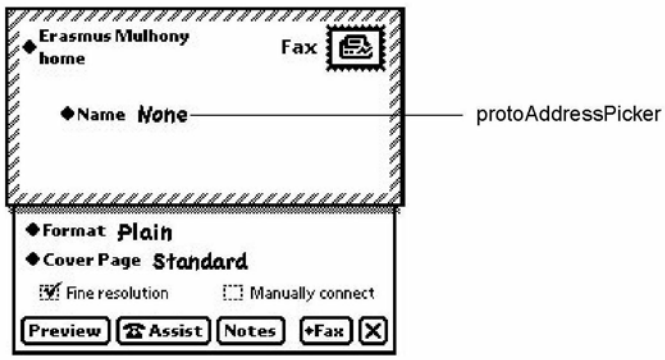
The Send button also handles submitting multiple items to the Out Box when the user has selected many entries from an overview. If the user has selected multiple items but the transport cannot handle cursors (the `allowBodyCursors` transport slot is `nil`), the system sends the transport the `VerifyRoutingInfo` method. This method allows the transport to modify the individual items, if necessary. When only a single item (not a multiple-item target object) is submitted to the Out Box, `VerifyRoutingInfo` is not called. In this case, if you need to modify the item before it is sent, you can do this in the routing slip method `PrepareToSend`.

The function of the Send button is to submit the contents of the `fields` slot in the routing slip to the Out Box. (The `fields` slot holds the item being routed and other information about it.) After the item is submitted, the Out Box sends the transport the `SendRequest` message to alert it that an item is waiting to be sent. If the `cause` slot in the *request* argument to `SendRequest` is set to `'submit`, this indicates the user chose to send the item later from the Send button. If the `cause` slot is `'item`, this indicates the user chose to send the item immediately. Additionally, the `connect` slot in the item contains a Boolean value indicating if the user chose to send the item now (`true`) or later (`nil`).

The name of the current transport appears in the upper-right corner of the `protoFullRouteSlip` view. If that transport belongs to a group, the transport name is actually a picker, from which the user can choose any of the other transports in the group. The picker is displayed only if there is more than one transport that belongs to the group. If the user changes the transport, the system closes and reopens the routing slip for the current target item, since the routing slip may be different for a different transport. Before the routing slip is closed, it is sent the `TransportChanged` message. This allows the routing slip to take such necessary action as alerting the user that addressing information might be lost as a result of changing transports. For more information on grouped transports, see the section "Grouping Transports" beginning on page 22-7.

Besides the supplied elements, your transport needs to add additional transport-specific elements to the routing slip view. For example, transports are responsible for adding the views that occupy the middle of the envelope area, to obtain routing or addressing information for the item. And transports typically add other elements to the area below the envelope. Figure 22-4 shows what a complete routing slip might look like, after you add transport-specific items.

Figure 22-4 Complete routing slip



The middle of the envelope portion of a routing slip template typically includes a view that gathers and displays recipient information for the item being sent. You'll probably want to use the `protoAddressPicker` to allow the user to choose recipients for the item. For details on how to use this proto, see the section "Using protoAddressPicker" beginning on page 22-31.

Positioning Elements in the Lower Portion of the Routing Slip

The height of the lower portion of the routing slip is controlled by the `bottomIndent` slot. Placing your own user interface elements in this portion of the routing slip is complicated by the fact that the format picker may or may not be inserted by the system. It is included only if there is more than one format for the item. Also, the system performs animation on the routing slip, changing the location of the bottom bounds.

Any user interface elements you add to this portion of the routing slip must be positioned relative to the bottom of the slip dynamically, at run time. You can determine the position of the bottom of the slip by calling the routing slip method `BottomOfSlip`. An alternative method of positioning elements dynamically is to make them sibling bottom-relative to the last child of the routing slip proto, which is the Send button.

Note that only the first child element you add needs to follow these rules. Additional elements can be positioned sibling-relative to it.

Using Owner Information

The `protoFullRouteSlip` view sends the `OwnerInfoChanged` callback method to itself if the user changes the selection of owner name or worksite location in the sender pop-up view. The `OwnerInfoChanged` method provides the chance to update any information in the routing slip that depends on data in the sender's current owner card or worksite. In addition, the `fromRef` slot in the item will probably need to be updated with new sender information. For more information about setting the `fromRef` slot, see the section "Obtaining an Item Frame" beginning on page 22-13.

In your `OwnerInfoChanged` method, you can obtain any changes by checking variables in which you are interested in the user configuration data, using the `GetUserConfig` function. For example, the area code at the user's location can be found by using this code:

```
GetUserConfig('currentAreaCode);
```

For a list of variables in the user configuration data, see "User Configuration Variables" (page 16-101) in *Newton Programmer's Reference*.

One issue to consider when saving items in the Out Box for later transmission is when to read the sender's owner card and worksite information. In general, data from the owner card should be obtained from the current persona at the time the item is queued by the user. Such information might include the sender's name, return address, credit card information, and so on.

However, if you use worksite information (for example, for addressing), you may want to wait until the item is actually transmitted to obtain the most current information based on the user's current worksite setting, and modify addressing

information at that time. For example, if a user queued several fax items from home but didn't send them until she got to work, the area code information for telephone numbers might need to be changed.

Using protoAddressPicker

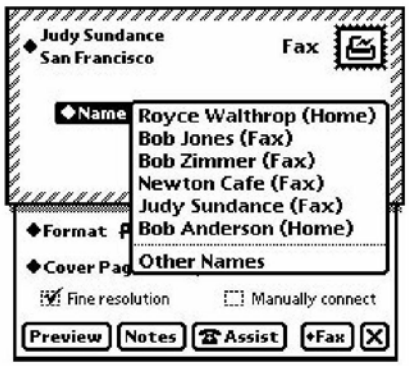
This proto consists of a labeled field that you can use in the routing slip to allow the user to choose the recipient(s) of the item being sent. The first time the user taps on the address picker, it opens a view that displays a list of names from the Names file, from which the user can choose one or more recipients (Figure 22-5).

Figure 22-5 `protoPeoplePicker` view

This view uses the `protoPeoplePicker` to provide the name picking facility. The address picker is customizable so that you can substitute a name picking service other than `protoPeoplePicker` by setting the `_picker` slot. For example, an e-mail transport might use this facility to provide an alternate directory service.

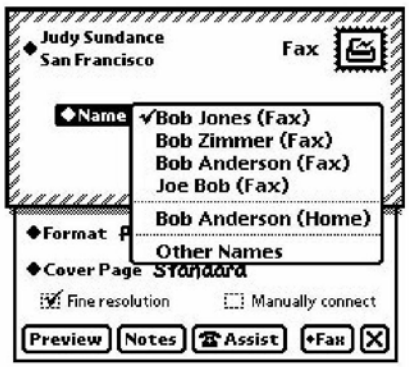
When the user picks a name, the information is saved, and the next time the address picker opens, it displays a small picker with the saved name and the choice "Other Names." The user can choose "Other Names" to reopen the `protoPeoplePicker` view and select from the comprehensive list of names. Each time a new name is selected, it is saved and added to the initial address picker list, giving the user a convenient way to select from recently used addresses, as shown in Figure 22-6. The address picker remembers the last eight names selected.

Figure 22-6 Address picker with remembered names



The Intelligent Assistant also interacts with the address picker. If the user invokes a routing action such as "fax Bob" with the Intelligent Assistant, the Intelligent Assistant sets up the address picker with a list of alternatives from the Names file, as shown in Figure 22-7.

Figure 22-7 Address picker set up by Intelligent Assistant



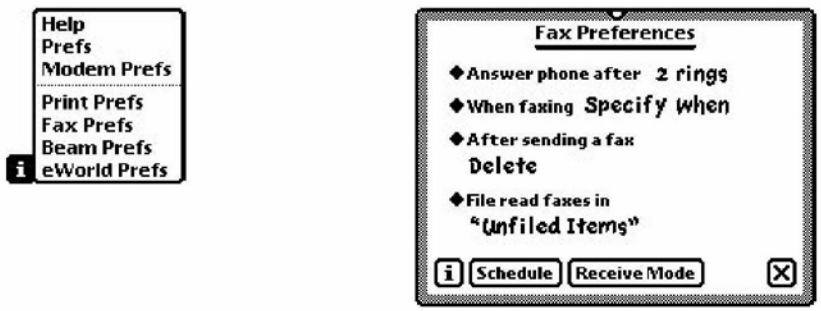
The `protoAddressPicker` uses name references to refer to individual names. A name reference is a frame that contains a soup entry or an alias to a soup entry, usually from the Names soup, hence the term name reference. The system includes built-in data definitions that can access name references and has associated view definitions that can display the information stored in or referenced by a name reference. The built-in data definitions and view definitions are registered under subclasses of the symbol `'nameRef`. For more information about name references, see "Name References" (page 5-1) in *Newton Programmer's Reference*.

Most transports can use the built-in name reference data and view definitions to handle and display name references. For example, one place you might need to use these is if you need to build a string representing the address or addresses chosen in the `protoAddressPicker`. The `selected` slot of the `protoAddressPicker` contains an array of name references for the names selected by the user in the picker. You can use the name reference data definition method `GetRoutingTitle` to return a string representing all the selected addresses, truncated to the length you specify. Alternately, you can use the transport method `GetNameText` to do the same thing.

Providing a Preferences Template

Transport preferences are accessed and changed from the information button in the In/Out Box. (The information button is the small button with an "i" in it.) Each transport with a preferences view is listed in the information picker, as shown in Figure 22-8.

Figure 22-8 Information picker and preferences view



Information picker Preferences view

To make a preferences view for a transport, create a template with a prototype of `protoTransportPrefs`. In your transport object, store a reference to your preferences view template in the `preferencesForm` slot. When the information picker is displayed, it automatically includes an item for each transport that has a preferences template registered in the transport's `preferencesForm` slot.

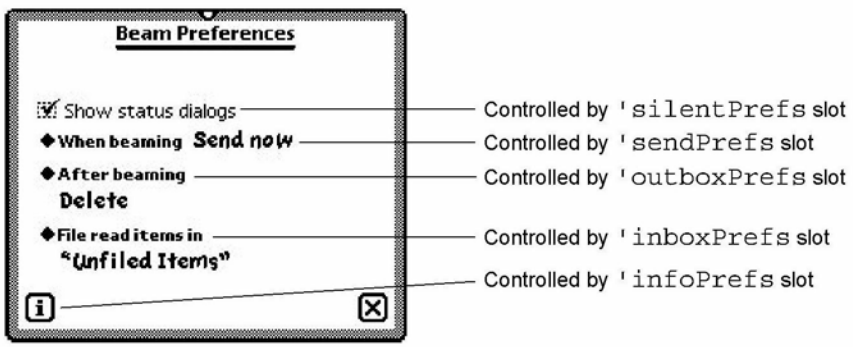
Each transport may add its own preferences view for configuring any options that apply to that transport. Some common options include

- enable/disable logging
- deferred/immediate send
- enable/disable listening

- default folders for new and read or sent items
- show/hide status and progress dialogs

The `protoTransportPrefs` proto provides a dialog containing the preferences items shown in Figure 22-9.

Figure 22-9 `protoTransportPrefs` view

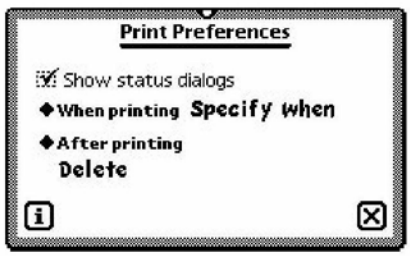


You can selectively remove any of the elements shown above by setting the corresponding slot to `nil` in the `protoTransportPrefs` view. To include additional items in your preferences view, add child views to the `protoTransportPrefs` view. The default child elements positioned in the center of the view are added from the bottom up and are justified relative to the bottom of the preferences view or to the top of their preceding sibling view. To add other child elements, increase the height of the view and add your elements above the existing ones, except for the title.

The `protoTransportPrefs` template also automatically checks your transport and displays or hides the In/Out Box preference elements. If your transport does not contain a `SendRequest` method, the Out Box preference element is not displayed; if your transport does not contain a `ReceiveRequest` method, the In Box preference element is not displayed. If the latter element is missing, the Out Box element is automatically drawn at the bottom of the preferences view.

For example, the built-in Print transport uses the `protoTransportPrefs` proto for its preferences view. Since the `ReceiveRequest` method does not exist in the Print transport, the In Box preference element is not displayed, as shown in Figure 22-10.

Figure 22-10 Print preferences



The Info button is included in the `protoTransportPrefs` template so you can give the user access to About and Help views for the transport. The button is built from the standard `protoInfoButton` proto. To include items on the Info picker, you must provide handler methods in the `infoPrefs` slot of your transport preferences view. The `protoTransportPrefs` template includes a handler for the "Help" item that displays the system help book, open to the routing section. You'll need to override this method if you want to provide your own help information.

You can add custom items to the Info picker by supplying `GenInfoAuxItems` and `DoInfoAux` methods in the `infoPrefs` frame. For more information about these methods and how the Info button works, see "protoInfoButton" (page 6-10) in *Newton Programmer's Reference*.

The `defaultConfiguration` slot in the `protoTransport` holds the initial preferences associated with the transport. This slot is set up by default with a frame holding an initial selection of preferences items. The child views of the `protoTransportPrefs` proto are designed to manipulate the slots in this frame.

If you want to override the default preferences frame, you need to construct an identical one with different values. You can't use a `_proto` slot in your default frame since the contents of the `defaultConfiguration` slot are stored in a soup and `_proto` slots can't be stored in soup entries.

Summary of the Transport Interface

Constants

```
ROM_RouteMailIcon // bitmap for mail group icon
ROM_RoutePrintIcon // bitmap for print group icon
ROM_RouteFaxIcon // bitmap for fax group icon
ROM_RouteBeamIcon // bitmap for beam group icon
ROM_RouteReply // bitmap for reply action icon
ROM_RouteForward // bitmap for forward action icon
ROM_RouteAddSender // bitmap for add sender to Names icon
ROM_RoutePasteText // bitmap for copy text to Notes icon
```

Protos

protoTransport

```
myTransport := {
_proto: protoTransport, // proto transport object
appSymbol: symbol, // transport symbol
title: string, // transport name
dataTypes: array, // symbols for routing types supported
actionTitle: string, // name of transport action
icon: bitmapFrame, // transport icon
group: symbol, // transport group symbol
groupTitle: string, // group name
groupIcon: bitmapFrame, // group icon
routingSlip: viewTemplate, // routing slip template
transportInfoForm: viewTemplate, // routing info template
preferencesForm: viewTemplate, // preferences template
statusTemplate: viewTemplate, // status template
statusDialog: view, // status view
modalStatus: Boolean, // modal status dialogs?
dialogStatusMsgs: frame, // status strings
status: symbol, // current status
addressingClass: symbol, // name reference symbol
addressSymbols: array, // don't translate e-mail classes
allowBodyCursors: Boolean, // allow cursors in body slot?
defaultConfiguration: frame, // user preferences defaults
AppClosed: function, // notifies transport of app closing
```

```
AppInFront: function, // notifies transport of change in
                          app frontmost status
AppOpened: function, // notifies transport of app opening
CancelRequest: function, // cancels in-progress operation
CanPutAway: function, // put away hook for transport
CheckOutbox: function, // invokes SendRequest operation
CloseStatusDialog: function, // closes status dialog
ConnectionDetect: function, // force send now or later
GetConfig: function, // returns a prefs value
GetDefaultOwnerStore: function, // returns default store
GetFolderName: function, // gets folder name for item
GetFromText: function, // hook to supply item sender
GetItemInfo: function, // returns item to or from info
GetItemStateString: function, // returns item status string
GetItemTime: function, // returns item time stamp info
GetItemTitle: function, // returns item title
GetNameText: function, // returns name string from namerefs
GetStatusString: function, // returns transport status
GetTitleInfoShape: function, // returns info shape
GetToText: function, // hook to supply item recipient(s)
GetTransportScripts: function,// extends In/Out Box actions
HandleError: function, // displays error alert
HandleThrow: function, // handles exceptions
IgnoreError: function, // screens errors
InstallScript: function, // notification of installation
IOBoxExtensions: function, // extends In/Out Box view defs
IsInItem: function, // is item in the In or Out Box?
IsLogItem: function, // has item been logged?
ItemCompleted: function, // processes an item
ItemDeleted: function, // called when item is deleted
ItemDuplicated: function, // called when item is duplicated
ItemPutAway: function, // called after item is put away
ItemRequest: function, // gets next queued item
MakeLogEntry: function, // makes log entry
MissingTarget: function, // notification of missing target
NewFromItem: function, // gets item frame for received data
NewItem: function, // gets new item frame
NormalizeAddress: function, // translates e-mail address
PowerOffCheck: function, // notification of power-off
QueueRequest: function, // queues item for later handling
ReceiveRequest: function, // receives data
SendRequest: function, // sends data
SetConfig: function, // sets a prefs value
SetStatusDialog: function, // opens/updates status dialog
```

```
TranslateError: function, // returns a string translation
VerifyRoutingInfo: function, // called on send of multiple
                      // item target that is being split
...
}
```

protoTransportHeader

```
aHeader := {
_proto: protoTransportHeader, // proto header object
transport: frame, // transport object
target: frame, // target object
addedHeight: integer, // height you're adding to header
context: view, // view to notify with InfoChanged msg
changed: Boolean, // user changed a field?
BuildText: function, // builds additional header lines
AddText: function, // adds lines to header
InfoChanged: function, // notifies view of changed field
...
}
```

protoFullRouteSlip

```
aFullRoutingSlip := {
_proto: protoFullRouteSlip, // proto full routing slip
viewJustify: integer, // viewJustify flags
envelopeHeight: integer, // height of envelope portion
envelopeWidth: integer, // width of envelope portion
bottomIndent: integer, // height of lower portion
fields: frame, // item frame
target: frame, // target object
activeFormat: frame, // currently selected format
transport: frame, // transport object
formatPicker: frame, // the format picker child view
sendButton: frame, // the send button child view
BottomOfSlip: function, // returns bottom of slip
FormatChanged: function, // notifies slip of new format
OwnerInfoChanged: function, // notifies slip of new sender
PrepareToSend: function, // notifies slip when item is sent
ContinueSend: function, // continues send process
TransportChanged: function, // notifies of transport change
...
}
```

protoAddressPicker

```
anAddressPicker := {
_proto: protoAddressPicker, // address picker
viewBounds: boundsFrame, // location and size
text: string, // picker label
otherText: string, // last item (pops up people picker)
selected: array, // name refs to be initially selected
alternatives: array, // name refs to be shown in picker
class: symbol, // name ref data def class
_picker: viewTemplate, // picker for other addresses
...
}
```

protoTransportPrefs

```
myTransportPrefs := {
_proto: protoTransportPrefs, // transport prefs proto
viewBounds: boundsFrame, // location and size
title: string, // transport name
appSymbol: symbol, // transport appSymbol
silentPrefs: frame, // controls checkbox element in prefs
sendPrefs: frame, // controls send element in prefs
outboxPrefs: frame, // controls out box prefs element
inboxPrefs: frame, // controls in box prefs element
infoPrefs: frame, // defines more info button choices
...
}
```

Functions and Methods

Utility Functions

RegTransport(*symbol*, *transport*)
UnRegTransport(*symbol*)
DeleteTransport(*symbol*)
GetCurrentFormat(*item*)
GetGroupTransport(*groupSymbol*)
QuietSendAll(*transportSym*) // platform file function
ownerApp:Refresh()
ownerApp:RemoveTempItems(*transportSym*)

CHAPTER 23

Endpoint Interface

This chapter describes the basic Endpoint interface in Newton system software. The Endpoint interface allows you to perform real-time communication using any of the communication tools available in the system. The Endpoint interface is well suited for communication needs such as database access and terminal emulation.

You should read this chapter if your application needs to perform real-time communications—that is, communication operations that do not use the Routing and Transport interfaces described in the previous chapters. This chapter describes how to

- set options to configure the underlying communication tool
- establish a connection
- send and receive data
- set up an input specification frame to control how data is received
- cancel communication operations

This chapter describes the general approach to using the Endpoint interface, but does not discuss details specific to using individual communication tools. For specific details on using particular built-in communication tools, see Chapter 24, "Built-in Communications Tools."

About the Endpoint Interface

The Endpoint interface is based on a single proto—`protoBasicEndpoint`—which provides a standard interface to all communication tools (serial, modem, infrared, AppleTalk, and so on). This proto provides methods for

- interacting with the underlying communication tool
- setting and getting endpoint options
- opening and closing connections
- sending and receiving data

The **endpoint** object created from this proto encapsulates and maintains the details of the specific connection. It allows you to control the underlying communication tool to perform your communication tasks.

The Endpoint interface uses an asynchronous, state-driven communications model. In asynchronous operation, communication requests are queued, and control is returned to your application after each request is made but before it is completed. Many endpoint methods can also be called synchronously. In synchronous operation, execution of your application is blocked until the request completes; that is, the endpoint method does not return until the communication operation is finished.

The Endpoint interface supports multiple simultaneous connections. That is, you can have more than one active endpoint at a time. Each endpoint object controls an underlying communication tool, and these tools run as separate operating system tasks. However, remember that the endpoint objects you create and control all execute within the single Application task.

The number of simultaneously active endpoints you can use is limited in practice by available system memory and processor speed. Each communication tool task requires significant memory and processor resources. Note that memory for the communication tools that underlie endpoints is allocated from the operating system domain, whereas memory for the endpoint objects is allocated from the NewtonScript heap.

Asynchronous Operation

Almost all endpoint methods can be called asynchronously. This means that calling the method queues a request for a particular operation with the underlying communication tool task, and then the method returns. When the operation completes, the communication tool sends a callback message to notify the endpoint that the request has been completed. The callback message is the `CompletionScript` message, and it is defined by your application in a frame called the callback specification, or **callback spec.** (For more details, see “Callback Spec Frame” (page 20-9) in *Newton Programmer’s Reference*.)

You define the callback spec frame in your application and pass it as an argument to each endpoint method you call asynchronously. The callback spec frame contains slots that control how the endpoint method executes, and it contains a `CompletionScript` method that is called when the endpoint operation completes. The `CompletionScript` method is passed a result code parameter that indicates if the operation completed successfully or with an error.

A special type of callback spec, called an **output spec**, is used with the `Output` method. An output spec contains a few additional slots that allow you to pass special protocol flags and to define how the data being sent is translated. Output specs are described in “Output Spec Frame” (page 20-10) in *Newton Programmer’s Reference*.

This kind of asynchronous operation lends itself nicely to creating state-machine based code, where each part of the communication process is a state that is invoked by calling an endpoint method. The `CompletionScript` method of each state invokes the next state, and the state machine automatically progresses from one state to the next in a predefined fashion.

Synchronous Operation

Many endpoint methods can be called synchronously as well as asynchronously. Synchronous operation means that invoking a method queues a request for a particular operation with the underlying communication tool task, and the method does not return until the operation is completed. This means that your application is blocked from execution until the synchronous method returns.

Only a few endpoint methods must be called synchronously. Most can be called either asynchronously or synchronously. For methods that can be called in either mode, it is recommended that you use the asynchronous mode whenever possible. If you call such a method synchronously, the communication system spawns a separate task associated with the method call, while putting your application task on hold. This results in higher system overhead and can reduce overall system performance if you use many synchronous method calls.

Input

In the Endpoint interface, you receive data by defining a frame called an input specification, or **input spec,** and then waiting for input. The input spec defines how incoming data should be formatted, termination conditions that control when the input should be stopped, data filtering options, and callback methods. The main callback method is the `InputScript` method, which is passed the received data when the input operation terminates normally. Receiving data with the Endpoint interface is *always asynchronous*.

Here is an overview of the way you can use input spec methods to obtain the received data:

- Let the termination conditions specified in the input spec be triggered by the received data, thus calling your `InputScript` method. For example, when a particular string is received, the `InputScript` method is called.
- Periodically sample incoming data by using the input spec `PartialScript` method, which is called periodically at intervals you specify in the input spec.
- Cause the system to send the `InputScript` callback method by using the `Input` method. This immediately returns the contents of the input buffer and clears it.
- Immediately return the input buffer contents without terminating the active input spec and without clearing the buffer by using the `Partial` method.

If the input operation terminates normally—that is, the `InputScript` method is called—the system automatically reposts the input spec for you to receive additional input. Of course, you can alter this process if you want to.

Data Forms

All NewtonScript data needs to be transformed whenever it is sent to or received from a foreign environment. That foreign environment may be a server or host computer at the other end of the connection, or it may even be the communication tool that's processing the configuration options you've passed to it. Typically, communication tools expect C-type option data.

Whether you're sending, receiving, or using data to set endpoint options, you can tag the data with a **data form.** A data form is a symbol that describes the transformations that need to take place when data is exchanged with other environments. When you send data or set endpoint options, the data form defines how to convert the data from its NewtonScript format. When you receive data or get endpoint options, the data form defines the type of data expected.

Data forms are used in output specs, input specs, and endpoint option frames. The data form is defined by a slot named `form` in these frames. If you don't define the data form in a particular case, a default data form is used, depending on the type of operation and the type of data being handled.

Note that when sending data, you can take advantage of the default data forms by not explicitly specifying a data form. Because NewtonScript objects have type information embedded in their values, the system can select appropriate default data forms for different kinds of data being sent. For example, if you send string data and don't specify the data form, the `'string` data form is used by default.

The symbols you use to indicate data forms are `'char`, `'number`, `'string`, `'bytes`, `'binary`, `'template`, and `'frame`. Each is best suited to certain data and operations.:

- For simple scalar values, use `'char` for characters and `'number` for integers.
- For semi-aggregate forms of these kinds of data, use `'string` for a concatenation of characters plus a terminating byte, and use `'bytes` for an array of bytes.
- For binary data, use `'binary`. This is the fastest option for sending and receiving, since the data processing is minimal.
- For more complex data, there are two aggregate data forms. You may want to use the `'template` form if you're communicating with a remote procedure call service that expects C-type data and that follows the exact same marshalling restrictions the Newton does. The `'frame` form is convenient if you're exchanging frames with another Newton.

The different types of data forms and the defaults are described in more detail in "Data Form Symbols" (Table 20-1 on page 20-2) in *Newton Programmer's Reference*.

Only a subset of data form values is applicable for any particular operation. Table 23-1 enumerates the data forms and their applicability to output specs, input specs, and endpoint option frames.

Table 23-1 Data form applicability

Data form	Output spec	Input spec	Option frame
`'char`	default for characters	OK	OK
`'number`	default for numbers	OK	OK
`'string`	default for strings	default	OK
`'bytes`	OK	OK	OK
`'binary`	default for binary objects; output spec can include optional `target` slot	OK; input spec must include `target` slot	OK
`'template`	OK	OK; input spec must include `target` slot	default
`'frame`	OK	OK	not applicable

Template Data Form

The `'template` data form enables you to pass data as if you were passing C structures, and is thus extremely useful in communicating with the lower level communication tools in getting and setting endpoint options.

When you set options or send data using the `'template` data form, the data is expected to be a frame containing two slots, `arglist` and `typelist`. The `arglist` slot is an array containing the data, the list of arguments. The `typelist` slot is a corresponding array containing the types that describe the data.

To get endpoint options, the data in the `data` slot must be a frame containing the `arglist` and `typelist` arrays. The `arglist` array should contain placeholder or default values. The system supplies the actual `arglist` values when the option list is returned.

In the same manner, to receive data, you must add a `target` slot to your input spec containing the `arglist` and `typelist` arrays. The `arglist` array contains

placeholder or default values, which the system fills in when the data is received. For more information, see the section "Specifying the Data Form and Target" beginning on page 23-13.

The data types that can be used in the `typelist` array are identified by these symbols: `'long`, `'ulong`, `'short`, `'byte`, `'char`, `'unicodechar`, `'boolean`, `'struct`, and `'array`. They are described in detail in "Data Type Symbols" (Table 20-2 on page 20-3) in *Newton Programmer's Reference*.

Note that the `'struct` and `'array` data types are not used alone, but in conjunction with other elements in a `typelist` array. They modify how the other elements are treated. The `'struct` data type defines the array as an aggregate structure of various data types that is padded to a long-word boundary (4 bytes in the Newton system). Note that the whole structure is padded, not each array element. You must specify the `'struct` data type in order to include more than one type of data in the array.

The `'array` data type defines the array as an aggregate array of one or more elements of a single data type. The `'array` data type is specified as a NewtonScript array of three items, like this:

`['array,` *dataTypeSymbol*`,` *integer*`]`

Replace the *dataTypeSymbol* with one of the other simple data types. And *integer* is an integer specifying the number of units of that data type to convert. To convert an entire string, including the terminator, specify zero for *integer*. A nonzero value specifies the exact number of units to be converted, independent of a termination character in the source string.

Here are some examples of how to use the `'array` data type to represent C strings and Unicode strings in NewtonScript. The first example shows how to convert between a NewtonScript string of undefined length and a C string (translated to/from Unicode):

```
['array, 'char, 0]
```

This example shows how to convert a four-character NewtonScript string to a C string:

```
['array, 'char, 4]
```

This example shows how to convert between a NewtonScript string and a Unicode string:

```
['array, 'unicodechar, 0]
```

The `'template` data form is intended primarily as a means of communicating with the lower level communication tools in the Newton system. You can use this data form to communicate with a remote system, however, you must be careful and know exactly what you are doing to use it for this purpose. Remember that the lengths of various data types and the byte order may be different in other systems and may change in future releases of the Newton operating system.

Endpoint Options

You configure the communication tool underlying an endpoint object by setting endpoint options. An endpoint option is specified in an **endpoint option frame** that is passed in an array as an argument to one of the endpoint methods. Options select the communication tool to use, control its configuration and operation, and return result code information from each endpoint method call. An alternative way to set options is to directly call the endpoint `Option` method.

There are three kinds of options you can set, each identified by a unique symbol:

- `'service` options, which specify the kind of communication service, or tool, to be controlled by the endpoint
- `'option` options, which control characteristics of the communication tool
- `'address` options, which specify address information used by the communication tool

For details on the particular options you can use with the built-in communication tools, see Chapter 24, “Built-in Communications Tools.”

Compatibility

The `protoBasicEndpoint` and `protoStreamingEndpoint` objects and all the utility functions described in this chapter are new in Newton system software version 2.0. The `protoEndpoint` interface used in system software version 1.x is obsolete, but still supported for compatibility with older applications. Do not use the `protoEndpoint` interface, as it will not be supported in future system software versions.

Specific enhancements introduced by the new endpoint protos in system software 2.0 include the following:

- **Data forms.** You can handle and identify many more types of data by tagging it using data forms specified in the `form` slot of an option frame.
- **Asynchronous behavior and callback specs.** Most endpoint methods can now be called asynchronously.
- **Flexible input specs.** Enhancements include support for time-outs and the ability to specify multiple termination sequences.
- **Better error handling.** Consistent with other system services, errors resulting from synchronous methods are signaled by throwing an exception.
- **Binary data handling.** The way binary (raw) data is handled has changed significantly. For input, you can now target a direct data input object, which results in significantly faster performance. For output, you can specify offsets and lengths, which allows you to send the data in chunks.

- **Multiple communication sessions.** The system now supports multiple simultaneous communication sessions. In other words, you can have more than one active endpoint at a time.

Using the Endpoint Interface

This section describes

- setting endpoint options
- initializing and terminating an endpoint
- establishing a connection
- sending data
- receiving data
- sending and receiving streamed data
- working with binary data
- canceling operations
- handling errors
- linking the endpoint with an application

Setting Endpoint Options

Endpoint options are specified in an endpoint option frame that is passed as an argument to an endpoint method. Typically you specify an array of option frames, setting several options at once. Note that you cannot nest an option array inside another one.

You must specify a single `'service` option, to select a communication tool. Then you usually specify one or more `'option` options to configure the communication tool—for example, to set the baud rate, flow control, and parity of the serial tool. Note that if you are using the modem communication tool, you can use the utility function `MakeModemOption` to return a modem dialing option for use with the built-in modem tool.

You may also need to specify an `'address` option, depending on the communication tool you are using. The only built-in tools that use an `'address` option are the modem and AppleTalk tools. Note that you should use the global functions `MakePhoneOption` and `MakeAppleTalkOption` to construct `'address` options for the modem and AppleTalk tools.

The slots in an endpoint option frame are described in detail in “Endpoint Option Frame” (page 20-7) in *Newton Programmer’s Reference*.

All option data you set gets packed together into one block of data. Each option within this block must be long-word aligned for the communication tools. So, when using the `'template` data form, you need to use the `'struct` type (at the beginning of the `typelist` array) to guarantee that the option is long-word aligned and padded. To set the serial input/output parameters, for instance, the option frame might look like this:

```
serialIOParms := {
   type: 'option,
   label: kCMOSerialIOParms,
   opCode: opSetNegotiate,
   data: {
      arglist: [
         kNoParity,  // parity
         k1StopBits, // stopBits
         k8DataBits, // dataBits
         k9600bps,   // bps
      ],
      typelist: [
         'struct,
         'uLong,
         'long,
         'long,
         'long
      ]
   }
};
```

To get the connection information, the option frame you construct might look like this:

```
connectInfoParms := {
   type: 'option,
   label: kCMOSerialIOParms,
   opCode: opGetCurrent,
   data: {
      arglist: [
         0, // parity placeholder
         0, // stopBits placeholder
         0, // dataBits placeholder
         0, // bps placeholder
      ],
      typelist: [
         'struct,
         'ulong,
         'long,
```

```
            'long,
            'long
        ]
    }
};
```

When you set endpoint options, the cloned option frame is returned to you so that you can check the result codes for individual options. If you set options with an asynchronous method call, the cloned option frame is returned as a parameter to the `CompletionScript` callback method. If you set options with a synchronous method call, the cloned option frame is returned as the value of the synchronous method itself.

The `result` slot in each option frame is always set for returned options. It can be set to any of the error codes listed in "Option Error Code Constants" (Table 20-5 on page 20-5) in *Newton Programmer's Reference*. If an option succeeds without errors, the `result` slot is set to `nil`.

Exceptions are not thrown when individual options fail. This allows a request to succeed if, for example, every specified option except one succeeds. If you need to determine whether a particular option succeeds or fails, you must check the `result` slot of the option in question.

Note that in one array of option frames, you can specify options that are of the same type, and that seem to conflict. Since options are processed one at a time, in order, the last option of a particular type is the one that is actually implemented by the communication tool.

Note
When instantiating an endpoint for use with the modem tool, you can have options specified by the *options* parameter to the `Instantiate` method, as well as options specified by a modem setup package (see Chapter 25, "Modem Setup Service."). Any options from a modem setup package are appended to those set by the `Instantiate` method. ◆

For details on the specific options you can set for the built-in communication tools, see Chapter 24, "Built-in Communications Tools."

Initialization and Termination

Before using an endpoint, you must instantiate it using the `Instantiate` method. This method allocates memory in the system and creates the endpoint object. Then, you must bind the endpoint object to the communication hardware by calling the `Bind` method. This allocates the communication tool resources for use by the endpoint.

When you are finished with an endpoint, you must unbind it using the `UnBind` method, then dispose of it using the `Dispose` method.

Establishing a Connection

After instantiating and binding an endpoint, you establish a connection.

There are two ways you can create a connection. One way is to call the `Connect` method. If the physical connection is serial, for instance, you don't even need to specify an address as an option. The `Connect` method immediately establishes communication with whatever is at the other end of the line.

Certain communication tools—for example, the modem and AppleTalk tools—require you to specify an option of type `'address` in order to make a connection. The modem tool requires a phone number as an `'address` option. You should use the global function `MakePhoneOption` to return a proper phone number `'address` option. The AppleTalk tool requires an AppleTalk Name Binding Protocol (NBP) `'address` option. You should use the global function `MakeAppleTalkOption` to return a proper NBP `'address` option.

To establish a connection where you expect someone else to initiate the connection, you need to call the `Listen` method. Once the connection is made by using `Listen`, you need to call the `Accept` method to accept the connection, or the `Disconnect` method to reject the connection and disconnect.

Sending Data

To send data, use the `Output` method. This method is intelligent enough to figure out the type of data you're sending and to convert it appropriately for transmission. This is because NewtonScript objects have type information embedded in their values, allowing the system to select appropriate default data forms for different kinds of data being sent.

You can specify output options and a callback method by defining an output spec, which you pass as a parameter to the `Output` method.

Certain communication tools may require or support the use of special flags indicating that particular protocols are in use. For example, the built-in infrared and AppleTalk tools expect framed (or packetized) data, and there are special flags to indicate that this kind of protocol is in use. If you are using such a communication tool to send data, you need to specify the `sendFlags` slot in the output spec frame. In this slot, you specify one or more flag constants added together.

To send packetized data, you set `sendFlags` to `kPacket+kMore` for each packet of data that is not the last packet. For the last packet, set `sendFlags` to `kPacket+kEOP`.

Receiving Data Using Input Specs

The most common way to receive data is to use input specs. An input spec is a frame that defines what kind of data you are looking for, termination conditions that control when the input should be stopped, and callback methods to notify you when input is stopped or other conditions occur.

An input spec consists of many pieces. It contains slots that define

- the type of data expected (`form` slot)
- the input target for template and binary data (`target` slot)
- the data termination conditions (`termination` slot)
- protocol flags for receiving data (`rcvFlags` slot)
- an inactivity time-out (`reqTimeout` slot)
- the data filter options (`filter` slot)
- the options associated with the receive request (`rcvOptions` slot)
- a method to be called when the termination conditions are met (`InputScript` method)
- a method to be called periodically to check input as it accumulates (`PartialScript` method, `partialFrequency` slot)
- a method to be called if the input spec terminates unexpectedly (`CompletionScript` method)

Table 23-2 summarizes the various input data forms and the input spec slots that are applicable to them. Input spec slots not included in the table apply to all data forms. For more details on the input spec frame, see “Input Spec Frame” (page 20-11) in *Newton Programmer’s Reference*.

After you’ve connected or accepted a connection, you set up your first input spec by calling `SetInputSpec`. When one input spec terminates, the system automatically posts another input spec for you when the `InputScript` method defined in the previous input spec returns. This new input spec duplicates the one that just terminated. If you don’t want this to happen, you can call the `SetInputSpec` method from within the `InputScript` method of your input spec to change the input spec or terminate the input. Pass `nil` to `SetInputSpec` to terminate the input.

You also use the `SetInputSpec` method if you need to set up an input spec at some other point. Note that if you want to terminate a current input spec to set up a new one, you must call the `Cancel` method before calling `SetInputSpec` with your new spec. (This applies inside an `InputScript` that is called as a result of calling the `Input` method.)

Table 23-2 Input spec slot applicability

Data form	`target` slot	`termination` slot	`discard After` slot*	`filter` slot	`partial Frequency` and `partial Script` slots†
`'char`	na (not applicable)	determined automatically	na	OK	na
`'number`	na	determined automatically	na	OK	na
`'string`	na	OK	OK	OK	OK
`'bytes`	na	OK	OK	OK	OK
`'binary`	`data` and `offset` slots only	all slots except `endSequence`	na	na	na
`'template`	`typelist` and `arglist` slots only	determined automatically	na	na	na
`'frame`	na	determined automatically	na	na	na

* `discardAfter` is written as one word, broken here because of space limitations.
† `partialFrequency` and `partialScript` are written as one word, broken here because of space limitations.

The following sections describe how to set the various slots in the input spec to accomplish specific tasks.

Specifying the Data Form and Target

You can choose how you want the received data formatted by setting the `form` slot in the input spec. In this slot, you specify one of the standard data forms described in "Data Form Symbols" (Table 20-1 on page 20-2) in *Newton Programmer's Reference*.

In preparation for receiving data, the system creates an input buffer. The buffer's size is based on the input spec slot `termination.byteCount`, on the slot `discardAfter`, or on the intrinsic size of the data. The system receives all the data in to this buffer, then translates the data into a newly created object whose type is specified by the input spec's `form` slot. It is this object that is passed back to the `InputScript` method.

If you specify the form `'template` or `'binary`, you also must specify a `target` slot in the input spec. The `target` slot is a frame used to define additional information pertaining to the data form.

If your input form is `'template`, then you must set the `arglist` and `typelist` slots in the `target` frame. The `arglist` array contains placeholder data, which is filled in with actual data when it is received, and the `typelist` array contains the template's array of types.

If your input form is `'binary`, data is written directly into the binary object that you specify in the `data` slot of the `target` frame. You can specify a binary object, virtual binary object, or string. Note that the binary object must be the same size as the received data; the system will not expand or shrink the object. For information on virtual binary objects, see Chapter 11, "Data Storage and Retrieval."

The `offset` slot in the `target` frame allows you to specify an offset in the binary object at which to write incoming data. For instance, if you want to write the received data in consecutive blocks in a binary object that already exists, you must set the `data` slot to the binary object, and set the `offset` slot to the byte offset at which you want the new data to be written for each block.

Specifying Data Termination Conditions

For `'string` and `'bytes` data forms, you must indicate when the input terminates by specifying a `termination` slot. You can terminate the input on these conditions:

- when a certain number of bytes has been received (set the `byteCount` slot)
- when a specific set of characters in the input stream has been found (set the `endSequence` slot)
- when the communication tool returns an end-of-packet indicator (set the `useEOP` slot)

Normally with the `'binary` data form, the input is terminated when the target object fills up. However, you can also use the `termination` slot with binary data to specify a byte count that causes the input to terminate after a certain number of bytes has been received. This feature is useful when you want to provide user feedback as a large binary object is being received. Set the `byteCount` slot in the `termination` frame, and, when the input terminates, repost the input spec with the `target.offset` slot offset by the value of the `termination.byteCount` slot.

If you want to receive data that ends with a particular sequence of data, define that sequence in the `endSequence` slot in the `termination` frame. The `endSequence` slot allows you to terminate input based on a particular sequence of incoming data called the termination sequence. You can specify a single

termination sequence, or an array of items, any one of which will cause the input to terminate. A termination sequence can be a single character, a string, a number, or an array of bytes. If you don't want to look for a termination sequence, don't define this slot.

For the `'binary` data form, you cannot use the `endSequence` slot to specify a termination condition.

Note
Note that the system executes byte-by-byte comparisons between the termination sequence and the input stream. To facilitate this process, the termination sequence (or elements within the `endSequence` array) is converted to a byte or binary format to speed the comparison. Internally, single characters are converted to single bytes using the translation table specified by the endpoint `encoding` slot. Numbers are converted to single bytes; strings are converted to binary objects. An array of bytes is also treated as a binary object. For large numbers, you must encode your number as an array of bytes if there are significant digits beyond the high order byte of the number. ◆

If you want to terminate input based on a transport-level end-of-packet (EOP) indicator, then you can set the `useEOP` slot in the `termination` frame. This slot holds a Boolean value specifying whether or not to look for EOP indicators. Specify this slot only if the input spec `rcvFlags` slot includes the `kPacket` flag. Moreover, if the `rcvFlags` slot includes the `kPacket` flag and you do not specify the `termination.useEOP` slot, the system effectively sets `useEOP` to the default value `true`. For more information, see the following section, "Specifying Flags for Receiving."

It is not appropriate to specify the `termination` slot for data forms other than `'string`, `'bytes`, and `'binary`. The `'char` and `'number` data forms automatically terminate after 1 and 4 bytes, respectively. The `'frame` data form is terminated automatically when a complete frame has been received, and the `'template` data form terminates when the number of bytes received matches the `typelist` specification in the `target` frame.

To limit the amount of accumulated data in the input buffer, you can define a `discardAfter` slot in the input spec. You can do this only when you have not specified a `termination.byteCount` slot for `'string` and `'bytes` data forms. The `discardAfter` slot sets the input buffer size. If the buffer overflows, older bytes are discarded in favor of more recently received bytes.

Specifying Flags for Receiving

For certain communication tools, it may be necessary to use special protocol flags when receiving data. You do this by specifying one or more flag constants in the `rcvFlags` slot in the input spec. You can use such flags only if the communication tool supports them.

For example, some of the built-in communication tools, such as the infrared and AppleTalk tools, support only framed receiving (packetized data). In order to use framed receiving, you must set the `rcvFlags` slot to the constant `kPacket`. With the infrared tool, if you do not specify a `rcvFlags` value of `kPacket`, the tool will behave unexpectedly.

Do not define the `rcvFlags` slot if the underlying communication tool does not support EOP indicators. If you do so, your input will terminate after each physical buffer of data is received. If you wish to terminate an input spec based on an EOP indicator, set the `useEOP` slot in the `termination` frame to `true`.

Of the built-in communication tools, only the infrared, AppleTalk, and framed asynchronous serial tools support framed packets and the `kPacket` flag.

If you set the `kPacket` flag and set the `useEOP` slot to `true`, you cannot also use the `byteCount` slot in the termination frame—if you do, `byteCount` will be ignored. In this case, only an EOP indicator will terminate input. If you do want to use the `byteCount` slot with the `kPacket` flag, set the `useEOP` slot to `nil`. In the latter case, the remote system should send an EOP indicator with every packet, though input won't terminate until the `byteCount` condition is met.

Specifying an Input Time-Out

You can specify a time-out for input in the `reqTimeout` slot of the input spec. In this slot, you specify the time, in milliseconds, of inactivity to allow during input. If there is no input for the specified interval, the time-out expires, the input is terminated, and the `CompletionScript` message is sent to the input spec frame. In this case, the result code passed with the `CompletionScript` message is –16005.

Don't specify a `reqTimeout` value less than 30 milliseconds.

Note that if a time-out expires for an asynchronous request such as receiving, that request and *all* outstanding requests are canceled.

Specifying Data Filter Options

As incoming data is received in the input buffer, the data can be processed, or filtered. This filtering can occur on all types of received data, except binary data (defined by the `'binary` data form). This filtering of data is defined by the `filter`

slot in the input spec. The `filter` slot is a frame containing two slots, `byteProxy` and `sevenBit`, which allow you to perform two different kinds of processing.

The `byteProxy` slot allows you to identify one or more characters or bytes in the input stream to be replaced by zero or one characters. You may, for instance, replace null characters (`0x00`) with spaces (`0x20`). Note that if your input data form is set to `'string`, you are encouraged to use this slot. Otherwise, null characters embedded in your string may prematurely terminate that string. (Remember, NewtonScript strings are null-terminated.)

The `byteProxy` slot contains an array of one or more frames. Each frame must have a `byte` slot, identifying the single-byte character or byte to be replaced, and a `proxy` slot, identifying the single-byte character or byte to be used instead. The `proxy` slot can also be `nil`, meaning that the original byte is to be removed completely from the input stream.

Note
Note that the system executes byte-by-byte comparisons and swaps between the bytes in the input stream and the replacements in the `proxy` slot. To facilitate this process, the values in the `byte` and `proxy` slots are converted to a byte format to speed the comparison and swap. Internally, single characters are converted to single bytes using the translation table specified in the endpoint `encoding` slot. Numbers are converted to single bytes. If a number has significant digits beyond the high-order byte, they will be dropped during the comparison and swap. ◆

You can also specify the `sevenBit` slot in the `filter` frame. Set this slot to `true` to specify that the high-order bit of every incoming byte be stripped (“zeroed out”). This is a convenient feature if you plan to communicate over links (particularly private European carriers) that spuriously set the high-order bit.

Specifying Receive Options

You can also set communication tool options associated with the receive request. To do this, specify an option frame or an array of option frames in the `rcvOptions` slot in the input spec. The options are set when the input spec is posted by the `SetInputSpec` method. The processed options are returned in the *options* parameter passed to the `InputScript` method.

Note that the options are used only once. If your `InputScript` method is called, for example, and it returns expecting the input spec to remain active, the options are not reposted. To explicitly reset the options in this example, you must call `SetInputSpec` within your `InputScript` method.

Handling Normal Termination of Input

The `InputScript` message is sent to the input spec frame when one of the termination conditions is met. You define the `InputScript` method in the input spec frame.

The received data is passed as a parameter to the `InputScript` method. Another parameter describes the specific condition that caused the input to terminate, in case you had specified more than one in the input spec.

When the `InputScript` method returns, the system automatically posts another receive request for you using the same input spec as the last one. You can prevent this by calling `SetInputSpec` within the `InputScript` method. In the `SetInputSpec` method, you can set a different input spec, or you can prevent a new input spec from being posted by setting the *inputSpec* parameter to `nil`. Note that while the input spec is `nil`, incoming data may be lost.

Periodically Sampling Incoming Data

You can sample the incoming data without meeting any of the termination conditions by specifying a `PartialScript` method in the input spec. The system sends the `PartialScript` message to the input spec frame periodically, at the frequency you define in the `partialFrequency` slot in the input spec, as long as there are one or more bytes of data in the input buffer. The system passes to the `PartialScript` method all of the data currently in the input buffer, but the data is not removed from the input buffer. If you want to remove this data from the input buffer, you can call the `FlushPartial` method.

Note that the sending of `PartialScript` messages is controlled by system idle events and is in no way triggered by receive request completions. The current input spec remains in effect after the `PartialScript` method returns.

You typically would use a `PartialScript` method to detect abnormal or out-of-band data not found by any of the usual input termination conditions.

You can specify `PartialScript` methods only for those input data forms that allow termination conditions —specifically, the `'string` and `'bytes` data forms.

To use the `PartialScript` method, you must also include the `partialFrequency` slot in the input spec. The `partialFrequency` slot specifies the frequency, in milliseconds, at which the input data buffer should be checked. If new data exists in the buffer, the `PartialScript` message is sent to the input spec frame.

Handling Unexpected Completion

The `CompletionScript` message is sent to the input spec frame when the input spec completes unexpectedly—for example, because of a time-out expiring or a `Cancel` message.

If you do not specify a `CompletionScript` method in your input spec frame, an exception is forwarded to the endpoint `ExceptionHandler` method.

Special Considerations

If you want to set up an input spec, but you never want to terminate the input, you can set up the input form to be either `'string` or `'bytes` data, and not define any of the data termination conditions. In this case, it is up to you to read and flush the input. You can do this by using a `PartialScript` method that calls the `FlushPartial` method at the appropriate times. Note that if the input exceeds the `discardAfter` size, the oldest data in the buffer is deleted to reduce the size of the input.

Alternatively, if you omit the `InputScript` method, yet define the input data form and termination conditions, the input continues to be terminated and flushed at the appropriate times. The only difference is that without an `InputScript` method, you'll never see the complete input.

Receiving Data Using Alternative Methods

The methods described in this section allow you to receive data in ways other than letting an input spec terminate normally. You may not need to use these methods; they're provided for flexibility in handling special situations.

You can force the system to send a pending input spec the `InputScript` message by calling the `Input` method. Note that this method is appropriate to use only when receiving data of the forms `'string` and `'bytes`. Also, in an `InputScript` method that is called as a result of calling `Input`, you cannot use `SetInputSpec` to change or terminate the input spec. Instead, you must first send the `Cancel` message to cancel the current input spec.

You can look at incoming data outside the scope of your `InputScript` or `PartialScript` method by calling the method `Partial`. This method returns data from the input buffer but doesn’t remove it from the buffer. You can use this method to sample incoming data without affecting the normal operation of your input spec and its callback methods. Note that this method is appropriate to use only when receiving data of the forms `'string` and `'bytes`.

IMPORTANT
Do not call the `Input` or `Partial` methods in a polling loop to look for incoming data. The Newton communications architecture requires a return to the main event loop in order to process incoming data from the endpoint's underlying communication tool. These methods are included as an alternate way of retrieving data from the incoming data buffer, not as a way to implement synchronous data receives. ▲

To flush data from the input buffer, you can use the methods `FlushInput` and `FlushPartial`. The `FlushInput` method discards all data in the input buffer, and `FlushPartial` discards all data read by the last call to the `Partial` method.

Streaming Data In and Out

Besides `protoBasicEndpoint`, there is another type of endpoint proto called `protoStreamingEndpoint`. The purpose of this streaming endpoint is to provide a way to send and receive large frames without having first to flatten or unflatten them.

Flattening refers to the process of converting a frame object into a stream of bytes. Unflattening refers to the process of converting those bytes back into a frame object.

With the streaming endpoint, frame data is flattened or unflattened in chunks as it is sent or received. This allows large objects to be sent and received without causing the NewtonScript heap to overflow as a result of having to convert an entire object at once.

The `protoStreamingEndpoint` proto is based on `protoBasicEndpoint` and includes a method, `StreamIn`, that allows you to receive streamed data. This method automatically unflattens received data into a frame object in memory, and can place embedded virtual binary objects directly on a store. Another method, `StreamOut`, allows you to send frame data as a byte stream. Note that these two methods are synchronous; that is, they don't return until the operation is complete. However, they do provide progress information during the operation by means of a periodic callback.

Working With Binary Data

For receiving binary data, the data is returned as a raw byte stream. The data is not converted and is block-moved directly into a binary object that you have preallocated and specified as the target for the input.

To create this target object, specify a `target` frame in your input spec. This frame contains a `data` slot and optionally an `offset` slot. The `data` slot contains the preallocated binary (or virtual binary) object, while the `offset` slot is the offset

within the binary object at which to stream data. For more information on receiving binary data and using the `target` frame, see the section "Specifying the Data Form and Target" beginning on page 23-13.

For sending data, the data is expected to be a binary object and is interpreted as a raw byte stream. That is, the data is not converted and is passed directly to the communication tool. This is the default data form for sending binary objects.

If you wish to send only a portion of your binary data at once, you can specify a `target` frame in the output spec. Within the `target` frame, the `offset` slot defines the offset from the beginning of the binary object at which to begin sending data, and the `length` slot defines the length of the data to send.

These binary capabilities are very useful if you wish to send and receive flattened frames "packetized" for a communication protocol. By using the global function `Translate`, you can flatten a frame. Then you can packetize the transmission by using the `target` frame in the output spec.

On the receiving end, you can preallocate a virtual binary object, and then assemble the packets using the `target` frame in the input spec. Once all binary data has been received, you can unflatten the frame using the `Translate` function again.

Canceling Operations

To stop endpoint operations, you can use the endpoint method `Cancel` or `Disconnect`. Endpoint operations can also be canceled indirectly as a result of a time-out expiring. Remember that you can set a time-out for a request in the callback spec that you pass to most endpoint methods, and you can set a time-out in an input spec.

Note that you cannot specify what is canceled. When you or the system cancel operations, all outstanding synchronous and asynchronous requests are canceled.

The cancellation process proceeds differently depending on whether you are canceling asynchronous or synchronous requests that you have previously queued. Following a cancellation, it is safe to proceed with other endpoint operations at different times, according to the following rules:

- If you use only asynchronous calls in your application, you can safely proceed after you receive the `CompletionScript` message resulting from the `Cancel` call (or from the method whose time-out expired).
- If you use only synchronous calls in your application, you can safely proceed after the cancelled synchronous call throws an exception as a result of the cancellation.

Mixing asynchronous and synchronous methods in your application is not recommended. However, if you do so, you should treat the cancellation process as if you had used all synchronous calls, and proceed only after an exception is thrown.

The cancellation itself can be invoked asynchronously or synchronously, and is handled differently in the system depending on how it's done. The details are explained in the following subsections.

Asynchronous Cancellation

Cancellation can be invoked asynchronously in the following ways:

- calling the `Cancel` method asynchronously, or calling the `Disconnect` method asynchronously with the *cancelPending* parameter set to `true`
- having a time-out expire for an asynchronous request

When cancellation is invoked asynchronously, the system first cancels all pending asynchronous requests. This means that the `CompletionScript` message is sent to the callback spec for each of these requests, and the `CompletionScript` *result* parameter is set to –16005.

Note
When calling `Cancel` asynchronously, it is possible that additional asynchronous requests might be queued (by a `CompletionScript` method) after the `Cancel` request is queued but before it is executed. These additional requests will fail with error –36003 since they will be processed after the cancel process begins. In fact, any endpoint request that is made while a cancel is in progress will fail with error –36003. ◆

Next, the cancel request itself completes by sending the `CompletionScript` message. This message is sent to the callback spec passed to the `Cancel` (or `Disconnect`) method. Or, if the cancellation was invoked as the result of a time-out expiration, the `CompletionScript` message is sent to the callback spec of whatever method timed out (or to the input spec, if input was in progress).

Finally, any pending synchronous request is canceled by throwing an exception that contains error code –16005.

Synchronous Cancellation

Cancellation can be invoked synchronously in the following ways:

- calling the `Cancel` method synchronously, or calling the `Disconnect` method synchronously with the *cancelPending* parameter set to `true`
- having a time-out expire for a synchronous request

When cancellation is invoked synchronously, the system first cancels any pending asynchronous requests. This means that the `CompletionScript` message is sent to the callback spec for each of these requests, and the `CompletionScript` *result* parameter is set to –16005.

Next, the `Cancel` (or `Disconnect`) method returns, and any pending synchronous request is canceled by throwing an exception that contains error code –16005. Or, if the cancellation was invoked as the result of a time-out expiration, then whatever method timed out throws an exception containing error code –16005.

Other Operations

The `Option` method allows you to get and set options apart from the *options* parameter to the `Bind`, `Connect`, `Listen`, `Accept`, and `Output` methods.

You can check the state of a connection by calling the `State` method.

Custom communication tools can return special events to the endpoint object through the `EventHandler` message. This message is sent to the endpoint whenever an event occurs that is not handled by one of the usual endpoint event handlers. A custom communication tool and an endpoint can use this mechanism to pass events from the communication tool up to the endpoint layer.

Error Handling

By specifying an `ExceptionHandler` method in your endpoint, you can handle exception conditions not caught by local `try...onexception` clauses, as well as exceptions not caught by `CompletionScript` methods.

When you call an endpoint method synchronously, and an error occurs in that method, the system throws an exception (usually of type `|evt.ex.comm|`). You can catch these exceptions in your application by using the `try...onexception` construct. It's a good idea to bracket every endpoint method call with this exception catching construct.

If an error occurs as a result of an asynchronous request, no exception is thrown, but the error is returned in the *result* parameter to the `CompletionScript` method associated with that request. If you did not define a `CompletionScript` method, or if the error is unsolicited, the error is forwarded to your `ExceptionHandler` method. If you did not define an `ExceptionHandler` method, then the communication system throws an exception. This exception is caught by the operating system, which displays a warning message to the user.

Constants for error codes generated by the Endpoint interface are defined in "Endpoint Error Code Constants" (Table 20-4 on page 20-4) in *Newton Programmer's Reference*.

When you use the `Option` method (or any method that takes options as a parameter), not only can the method itself fail, but a failure can occur in processing each of the individual option requests. If the latter happens, the `result` slot in the returned option frame is set to one of the option error codes listed in "Option Error Code Constants" (Table 20-5 on page 20-5) in *Newton Programmer's Reference*. If

an option succeeds without errors, the `result` slot is set to `nil`. For more general information on setting options, see the section "Endpoint Options" beginning on page 23-7.

Power-Off Handling

During send and receive operations, you may want to protect against the system powering off so that the connection is not broken. The system can power-off unexpectedly as a result of the user inadvertently turning off the power or as a result of a low battery. If you want to be notified before the system powers off, you can register a callback function that the system will call before the power is turned off. Depending on the value you return from your callback function, you can prevent, delay, or allow the power-off sequence to continue.

For details on registering power handling functions, see Chapter 17, "Additional System Services."

Linking the Endpoint With an Application

If your endpoint is going to be driven by an application, you'll have a reference to the endpoint frame in your application. Also, you'll probably want to have a reference to your application base view in the endpoint frame, so you can handle endpoint messages in your application through inheritance.

The easiest way to link the endpoint and application together is to create a slot in your application base view like this:

```
ViewSetupFormScript: func ()
   begin
   self.fEndPoint: {_proto: protoBasicEndpoint,
                    _parent: self};
   end
```

This creates an endpoint frame as a slot in the application base view at run time, and makes the application base view (`self` here) the parent of the endpoint frame, so it can receive endpoint messages through inheritance.

Summary of the Endpoint Interface

Constants and Symbols

Data Form Symbols

```
'char
'number
'string
'bytes
'binary
'template
'frame
```

Data Type Symbols

```
'long
'ulong
'short
'byte
'char
'unicodechar
'boolean
'struct
'array
```

Option Opcode Constants

`opSetNegotiate`	256
`opSetRequired`	512
`opGetDefault`	768
`opGetCurrent`	1024

Endpoint State Constants

kUninit	0
kUnbnd	1
kIdle	2
kOutCon	3
kInCon	4
kDataXfer	5
kOutRel	6
kInRel	7
kInFlux	8
kOutLstn	9

Other Endpoint Constants

kNoTimeout	0
kEOP	0
kMore	1
kPacket	2

Data Structures

Option Frame

```
myOption := {
type: symbol, // option type
label: string, // 4-char option identifier
opCode: integer, // an opCode constant
form: 'template, // default form for options
result: nil, // set by the system on return
data: {
   arglist: [], // array of data items
   typelist:[], // array of data types
   }
}
```

Callback Spec Frame

```
myCallbackSpec := {
async: Boolean, // asynch request?
reqTimeout: integer, // time-out period, or 0
CompletionScript: // called when request is done
   func(endpoint, options, result)...,
}
```

Output Spec Frame

```
myOutputSpec := {
async: Boolean, // asynch request?
reqTimeout: integer, // time-out period, in milliseconds
sendFlags: integer, // flag constant(s)
form: symbol, // data form identifier
target: { // used for 'binary data forms
   offset: integer, // offset to begin sending from
   length: integer // number of bytes to send
   },
CompletionScript: // called when request is done
   func(endpoint, options, result)...,
}
```

Input Spec Frame

```
myInputSpec := {
form: symbol, // data form identifier
target: { // used with 'template and 'binary data forms
   typelist: [], // array of data types
   arglist: [], // array of data items
   data: object, // binary object to receive data
   offset: integer // offset at which to write data
   },
termination: { // defines termination conditions
   byteCount: integer, // number of bytes to receive
   endSequence: object, // char,string,number,or byte array
   useEOP: Boolean // terminate on EOP indicator?
   },
discardAfter: integer, // buffer size
rcvFlags: integer, // receive flag constant(s)
reqTimeout: integer, // time-out period, in milliseconds
```

```
filter: { // used to filter incoming data
   byteProxy: [{ // an array of frames
      byte: char, // char or byte to replace
      proxy: char // replacement char or byte, or nil
      }, ...],
   sevenBit: Boolean // strip high-order bit
   },
rcvOptions: [], // array of options, or a single frame
partialFrequency: integer,// freq, in milliseconds, to call
                          // PartialScript
InputScript: // called when input is terminated
   func(endpoint, data, terminator, options)...,
PartialScript: // called at partialFrequency interval
   func(endpoint, data)...,
CompletionScript: // called on unexpected completion
   func(endpoint, options, result)...,
}
```

Protos

protoBasicEndpoint

```
myEndpoint := {
_proto: protoBasicEndpoint, // proto endpoint
encoding:integer,//encoding table,default=kMacRomanEncoding
Instantiate: // instantiates endpoint object
   func(endpoint, options) ...,
Bind: // binds endpoint to comm tool
   func(options, callbackSpec) ...,
UnBind: // unbinds endpoint from comm tool
   func(options, callbackSpec) ...,
Dispose: // disposes endpoint object
   func(options, callbackSpec) ...,
Connect: // establishes connection
   func(options, callbackSpec) ...,
Listen: // passively listens for connection
   func(options, callbackSpec) ...,
Accept: // accepts connection
   func(options, callbackSpec) ...,
Disconnect: // disconnects
   func(cancelPending, callbackSpec) ...,
Output: // sends data
   func(data, options, outputSpec) ...,
SetInputSpec: // sets input spec
   func(inputSpec)...,
```

```
Input: // returns data from input buffer and clears it
   func() ...,
Partial: // returns data from input buffer
   func() ...,
FlushInput: // flushes whole input buffer
   func() ...,
FlushPartial: // flushes input buffer previously read
   func() ...,
Cancel: // cancels operations
   func(callbackSpec) ...,
Option: // sets & gets options
   func(options, callbackSpec) ...,
ExceptionHandler: // called on exceptions
   func(error) ...,
EventHandler: // called on unhandled events
   func(event) ...,
State: // returns endpoint state
   func() ...,
...
}
```

protoStreamingEndpoint

```
myStreamEndpoint := {
_proto: protoStreamingEndpoint, // proto endpoint
StreamIn: // receives stream data
   func({   form: 'frame, // required
            reqTimeout: integer, // time-out in ms.
            rcvFlags: integer, // receive flag constant(s)
            target: {
               store: store}, // store for VBOs
            ProgressScript: // progress callback
               func(bytes, totalBytes)...
   }) ...,
StreamOut: // sends stream data
   func(data,
           {form: 'frame, // required
            reqTimeout: integer, // time-out in ms.
            sendFlags: integer, // send flag constant(s)
            ProgressScript: // progress callback
               func(bytes, totalBytes)...
   }) ...,
...
}
```

Functions and Methods

Utility Functions

`MakeAppleTalkOption(`*NBPaddressString*`)`
`MakeModemOption()`
`MakePhoneOption(`*phoneString*`)`
`Translate(`*data*`, `*translator*`, `*store*`, `*progressScript*`)`

CHAPTER 24

Built-in Communications Tools

This chapter describes the built-in communications tools provided in Newton system software 2.0. The following tools are built into the system:

- Serial
- Modem
- Infrared
- AppleTalk

These communications tools are accessed and used through the Endpoint interface. This chapter provides an introduction to each tool and the options that you use with each. For detailed descriptions of the options, see "Built-in Communications Tools Reference" (page 21-1) in *Newton Programmer's Reference*.

For basic information on using communications endpoints, see "Endpoint Interface" (page 23-1).

Serial Tool

Three varieties of the serial tool are built into Newton system software:

- a standard asynchronous serial tool
- a standard asynchronous serial tool with Microcom Networking Protocol (MNP) compression
- a framed asynchronous serial tool

These serial tool varieties are described in the following three subsections.

Standard Asynchronous Serial Tool

You use the standard asynchronous serial communications tool to perform standard, asynchronous communications, including sending and receiving data.

The following is an example of how to create an endpoint that uses the standard asynchronous serial tool:

```
myAsyncEP := {_proto:protoBasicEndpoint};
myOptions := [
   { label:   kCMSAsyncSerial,
     type:    'service,
     opCode:  opSetRequired } ];
returnedOptions:= myAsyncEP:Instantiate(myAsyncEP,
   myOptions);
```

Table 24-1 summarizes the standard serial options. Each of these options is described in detail in "Options for the Standard Asynchronous Serial Tool" (page 21-2) in *Newton Programmer's Reference*.

Table 24-1 Summary of serial options

Label	Value	Use when	Description
`kCMOSerialHWChipLoc`	"schp"	Before or at binding	Sets which serial hardware to use.
`kCMOSerialChipSpec`	"sers"	Before or at binding	Sets which serial hardware to use and returns information about the serial hardware.
`kCMOSerialCircuitControl`	"sctl"	After connecting	Controls usage of the serial interface lines.
`kCMOSerialBuffers`	"sbuf"	Before or at binding	Sets the size of the input and output buffers.
`kCMOSerialIOParms`	"siop"	Any time	Sets the bps rate, stop bits, data bits, and parity options.
`kCMOSerialBitRate`	"sbps "	Any time	Changes the bps rate.
`kCMOOutputFlowControlParms`	"oflc"	Any time	Sets output flow control parameters.
`kCMOInputFlowControlParms`	"iflc"	Any time	Sets input flow control parameters.

continued

Table 24-1 Summary of serial options (continued)

Label	Value	Use when	Description
`kCMOSerialBreak`	"sbrk"	After connecting	Sends a break.
`kCMOSerialDiscard`	"sdsc"	After connecting	Discards data in input and/or output buffer.
`kCMOSerialEventEnables`	"sevt"	Any time	Configures the serial tool to complete an endpoint event on particular state changes.
`kCMOSerialBytesAvailable`	"sbav"	After connecting	Read-only option returns the number of bytes available in the input buffer.
`kCMOSerialIOStats`	"sios"	After connecting	Read-only option reports statistics from the current serial connection.
`kHMOSerExtClockDivide`	"cdiv"	After binding	Used only with an external clock to set the clock divide factor.

You can get or set most of the standard serial options in the endpoint method that established the state, as shown in Table 24-1. You set the endpoint options by passing an argument to the communications tool when calling one of the endpoint methods such as `Instantiate`, `Bind`, and `Connect`. For example, when you pass an option to the `Bind` method, the system sets the option and then does the binding.

Many of the communications options can only be used when the communications tool is in a certain state. For example, the first option in Table 24-1, `kCMOSerialHWChipLoc`, can only be used after the endpoint has been instantiated and before the binding is made. That means you could use it in the `Instantiate` and `Bind` methods, but not in the `Connect` method.

All of these options have default values, so you may not need to use an option if the default values provide the behavior you want. However, the default values do not apply partially. This means that if you do use an option, you must specify a value for each field within it.

Serial Tool with MNP Compression

The asynchronous serial communications tool with MNP compression works just like a standard asynchronous serial endpoint, except that it uses MNP data compression.

The following is an example that shows how to create an endpoint that uses the serial tool with MNP compression:

```
myMnpEP := {_proto:protoBasicEndpoint};
myOptions := [
   { label:   kCMSMNPID,
     type:    'service,
     opCode:  opSetRequired } ];
returnedOptions:= myMnpEP:Instantiate(myMnpEP,
      myOptions);
```

The serial tool with MNP endpoint uses all of the standard serial options, as well as two MNP options, which are summarized in Table 24-2. These options are described in detail in "Options for the Serial Tool with MNP Compression" (page 21-27) in *Newton Programmer's Reference*.

Table 24-2 Summary of serial tool with MNP options

Label	Value	Use when	Description
`kCMOMNPCompression`	"mnpc"	Before connecting	Sets the data compression type.
`kCMOMNPDataRate`	"eter"	Any time	Configures internal MNP timers.

Framed Asynchronous Serial Tool

The framed asynchronous serial communications tool is a superset of the standard asynchronous serial communications tool. This tool supports the sending and receiving of framed data. If you use this tool and do not specify framing for a send or receive operation, the framed asynchronous serial tool works exactly like the standard asynchronous serial tool.

When you use framing for input, the framed asynchronous serial tool discards characters until a start of frame sequence is detected and terminates input with an end-of-file (EOF) indication when the end-of-frame sequence is detected. The tool reports an error is if a CRC error is detected.

When you use framing for output, the data is prefixed with the start-of-frame sequence. The end-of frame-sequence and the calculated CRC are sent at the end of the data. The escape character is used for data transparency during framed operations.

An endpoint can include `kPacket`, `kEOP`, and `kMore` flags to control the sending and receiving of framed (packetized) data with the framed asynchronous serial tool. For more information on these flags, see "Sending Data" (page 23-11).

The following is an example that shows how to create an endpoint that uses the framed asynchronous serial tool:

```
myFramedEP := {_proto:protoBasicEndpoint};
myOptions := [
   { label:   kCMSFramedAsyncSerial,
     type:    'service,
     opCode:  opSetRequired } ];
returnedOptions:= myFramedEP:Instantiate(myFramedEP,
myOptions);
```

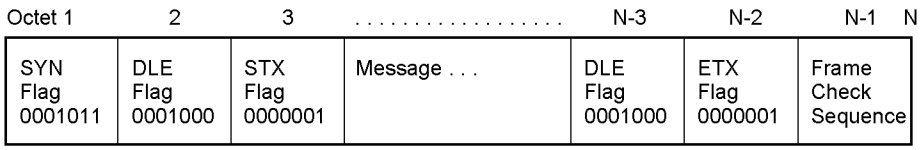
The framed asynchronous serial tool uses the standard asynchronous serial tool options, as well as two framing options, which are summarized in Table 24-3. These options are described in detail in "Options for the Framed Asynchronous Serial Tool" (page 21-29) in *Newton Programmer's Reference*.

Table 24-3 Summary of framed serial options

Label	Value	Use when	Description
`kCMOFramingParms`	"fram"	Any time	Configures data framing parameters.
`kCMOFramedAsyncStats`	"frst"	Any time	Read-only option returns the number of bytes discarded while looking for a valid header.

The default settings for the `kCMOFramingParms` option implement BSC framing, as shown in Figure 24-1.

Figure 24-1 Default serial framing



Each packet is framed at the beginning by the 3-character SYN-DLE-STX header. The packet data follows; if a DLE (escape character) occurs in the data stream,

both that character and an additional DLE character are sent; conversely, two consecutive DLE characters on input are turned into a single DLE data byte. The packet is framed at the end by the 2-character DLE-ETX trailer. Finally, a 2-character frame check sequence is appended. This frame check is initialized to zero at the beginning, and calculated on just the data bytes and the final ETX character, ignoring the header bytes, any inserted DLE characters, and the DLE character in the trailer.

The frame trailer is sent when an output is done that specifies end of frame. Conversely, on input, when a trailer is detected, the input is terminated with an end of frame indication; if a CRC error is detected, `kSerErr_CRCError` is returned instead.

Modem Tool

The modem communications tool includes built in support of V.42 and V.42bis. The alternate error-correcting protocol in V.42, also known as MNP, is supported (LAPM is not implemented). V.42bis data compression and MNP Class 5 data compression are supported.

The following is an example of how to create an endpoint that uses the built-in modem communications tool:

```
myModemEP := {_proto:protoBasicEndpoint};
myOptions := [
   { label:   kCMSModemID,
     type:    'service,
     opCode:  opSetRequired } ];
results := myModemEP:Instantiate(myModemEP, myOptions);
```

Table 24-4 summarizes the modem options you can use to configure the modem communications tool. These options are described in detail in “Options for the Modem Tool” (page 21-31) in *Newton Programmer’s Reference*.

Table 24-4 Summary of modem options

Label	Value	Use When	Description
kCMOModemPrefs	"mpre"	Any time	Configures the modem controller.
kCMOModemProfile	"mpro"	Any time	Override modem setup selected in preferences. Use when instatiating.
kCMOModemECType	"mecp"	Any time	Specifies the type of error control protocol to be used in the modem connection.
kCMOModemDialing	"mdo"	Any time	Controls the parameters associated with dialing.
kCMOModemConnectType	"mcto"	Any time	Configures the modem endpoint for the type of connection desired (voice, fax, data, or cellular data).
kCMOModemConnectSpeed	"mspd"	After connecting	Read-only option indicating modem-to-modem raw connection speed.
kCMOModemFaxCapabilities	"mfax"	After bind, before connecting	Read-only option indicating the fax service class capabilities and modem modulation capabilities.
kCMOModemFaxEnabledCaps	"mfec"	Any time	Determines or sets which fax and modem capabilities are enabled. This option is available only for System Software version 2.1 or later.
kCMOModemVoiceSupport	"mvso"	After bind, before connecting	Read-only option indicating if the modem supports line current sense (LCS).
kCMOMNPSpeedNegotiation	"mnpn"	Any time	Sets MNP data rate speed.
kCMOMNPCompression	"mnpc"	Before connecting	Sets the data compression type.
kCMOMNPStatistics	"mnps"	After connecting	Read-only option reporting performance statistics from the current MNP connection.

Infrared Tool

You use the infrared (IR) communications tool to perform half-duplex infrared communications. Since the infrared tool does not support full-duplex communications, you cannot activate an input specification and expect to output data.

The infrared tool supports packetized data, which means that an endpoint can include `kPacket`, `kEOP`, and `kMore` flags to control sending and receiving framed (packetized) data. For more information on these flags, see "Sending Data" (page 23-11).

The following is an example of how to create an endpoint that uses the infrared communications tool:

```
myIrEP := {_proto:protoBasicEndpoint};
myOptions := [
   { label:   kCMSSlowIR,
     type:    'service,
     opCode:  opSetRequired }
];


returnedoptions:= myIrEP:Instantiate(myIrEP, myOptions);
```

The infrared tool supports three options, which are summarized in Table 24-5. These options are described in detail in "Options for the Infrared Tool" (page 21-65) in *Newton Programmer's Reference*.

Table 24-5 Summary of Infrared Options

Label	Value	Use when	Description
`kCMOSlowIRConnect`	"irco"	When initiating, connecting, or listening	Controls how the connection is made
`kCMOSlowIRProtocolType`	"irpt"	After connecting or accepting	Read-only option returns the protocol and speed of the connection
`kCMOSlowIRStats`	"irst"	After connecting or accepting	Read-only option returns statistics about the data received and sent

The infrared tool uses the Sharp Infrared protocol. Because of the characteristics of this protocol, Apple recommends setting `sendFlags` to `kPacket` and to `kEOP` every time you send data. For more information on `sendFlags` see, "Sending Data" (page 23-11).

If you don't set `sendFlags` as recommended above, the tool only sends data after it queues 512 bytes of data, which means that input scripts do not terminate as you might expect. On the receiving side, the queuing means you terminate after every output if you set `useEOP` to `true`. If you are using `byteCount`, you should set `useEOP` to `nil` to trigger on `byteCount` instead of `EOP`. For more information on `useEOP` and `byteCount`, see "Specifying Data Termination Conditions" (page 23-14).

AppleTalk Tool

The AppleTalk tool enables access to the ADSP (Apple Data Stream Protocol component of the AppleTalk protocol stack.

The following is an example of how to create an AppleTalk endpoint:

```
myATalkEP := {_proto:protoBasicEndpoint};
myOptions := [
    { label:   kCMSAppleTalkID,
      type:    'service,
      opCode:  opSetRequired
    },
    { label:   kCMSAppleTalkID,
      type:    'option,
      opCode:  opSetRequired,
      data:    { arglist: ["adsp"],// or KCMOAppleTalkADSP
                 typelist:[
                    'struct
                    ['array, 'char, 4]
                    ]
               }
    },
    { label:   kCMOEndpointName,
      type:    'option,
      opCode:  opSetRequired,
      data:    { arglist: [kADSPEndpoint],
                 typelist:[
                    'struct
                    ['array, 'char, 0]
                    ]
```

```
                }
    } ];
results := myATalkEP:Instantiate(myATalkEP, myOptions);
```

The AppleTalk tool options are summarized in Table 24-6. These options are described in detail in "Options for the AppleTalk Tool" (page 21-71) in *Newton Programmer's Reference*.

Table 24-6 Summary of AppleTalk options

Label	Value	Use When	Description
`kCMARouteLabel`	"rout"	When connecting or listening	Sets an AppleTalk NBP address.
`kCMOAppleTalkBuffer`	"bsiz"	When connecting, listening, or accepting	Sets the size of the send, receive, and attention buffers.
`kCMOSerialBytesAvailable`	"sbav"	After connecting	Read-only option returns the number of bytes available in the receive buffer.
`kCMSAppleTalkID`	"atlk"	For instantiation	Specifies AppleTalk tool type.
`kCMOEndpointName`	"endp"	For instantiation	Specifies AppleTalk endpoint. Must be used as above.

Resource Arbitration Options

You can construct a communications tool to share its resources with other communications tools. For example, you might need to use a hardware port that other tools want to use. This section describes how you can implement resource sharing in your communications tool.

The communications tool base provides a default implementation of resource arbitration that uses two options to control the release of a tool's resources:

- The resource-passive claim option (`kCMOPassiveClaim`) has a Boolean value that specifies whether or not a communications tool is claiming its resources passively or actively. If this value is `true`, the communications tool is

claiming its resources passively and will allow another tool to claim it. If this value is `nil`, the communications tool is claiming its resources actively and will not allow another tool to claim it.

- The resource-passive state option (`kCMOPassiveState`) has a Boolean value that specifies whether or not the current state of the communications tool supports releasing resources. If this value is set, and `kCMOPassiveClaim` is `true`, your communications tool is willing to relinquish use of its passively claimed resources. If this value is `nil`, the communications tool is not willing to relinquish use of its passively claimed resources.

Table 24-7 shows the resource arbitration options. These options are described in detail in "Options for Resource Arbitration" (page 21-82) in *Newton Programmer's Reference*.

Table 24-7 Resource arbitration options

Label	Value	Use when	Description
`kCMOPassiveClaim`	"cpcm"	Before bind	Specifies whether your tool claims resources actively or passively
`kCMOPassiveState`	"cpst"	Typically on listen	Specifies whether your tool releases resources

The following example demonstrates how to instruct a communications tool to claim its resources passively. You must do this before binding the tool. By default all tools are claimed actively.

```
{
   label:  kCMOPassiveClaim,
     type:   'option,
   opCode: opSetRequired,
   data: {
       arglist: [
         true,   // passively claim modem
      ],
      typelist: [
         kStruct,
         kBoolean,
      ]
   }
}
```

The following example shows how to instruct a communications tool to allow its resources to be claimed by another tool. For instance, you might send this option with an `arglist` value of `true` if you are listening for an incoming connection. The default for all tools is to be in an active state.

```
{
   label:  kCMOPassiveState,
   type:   'option,
   opCode: opSetRequired,
   data: {
      arglist: [
         true,   // passively claim modem
      },
      typelist: [
         kStruct,
         kBoolean,
      ]

}
```

AppleTalk Functions

The Newton system software provides a number of global functions for obtaining the addresses of other devices on the network.

If you are using an endpoint with the AppleTalk tool, the AppleTalk drivers are opened automatically when you call the endpoint `Bind` method. The drivers are closed when you call the endpoint `UnBind` method.

To manually open the AppleTalk drivers, you need to call the `OpenAppleTalk` function. When you are done with AppleTalk, call the `CloseAppleTalk` function to close the drivers.

Note that you call the AppleTalk zone access functions without first calling `OpenAppleTalk`. Each of the AppleTalk zone access functions opens the drivers (if necessary), performs its operations, and closes the drivers (if necessary). If you are making multiple AppleTalk calls, it is more efficient for you to manually open the drivers, make your calls, and then close the drivers.

Table 24-8 summarizes the AppleTalk functions. These functions are described in detail in "AppleTalk Functions" (page 21-76) in *Newton Programmer's Reference*.

Table 24-8 AppleTalk functions

Function	Description
`OpenAppleTalk`	Opens the AppleTalk drivers.
`CloseAppleTalk`	Closes the AppleTalk drivers.
`AppleTalkOpenCount`	Returns the open count for the AppleTalk drivers.
`HaveZones`	Returns `true` if a connection exists and zones are available. Returns `nil` if there are no zones.
`GetMyZone`	Returns a string naming the current AppleTalk zone.
`GetZoneList`	Returns an array containing strings of all the existing zone names
`GetNames`	Returns the name for a network address or an array of names for an array of network addresses.
`GetZoneFromName`	Returns the zone name for a network address.
`NBPStart`	Begins a lookup of network entities.
`NBPGetCount`	Returns the number of entities the currently running NBP lookup has found.
`NBPGetNames`	Returns an array of names found by `NBPStart`.
`NBPStop`	Terminates a lookup started by `NBPStart`.

The Net Chooser

The Newton system provides a NetChooser as part of the root view. The Net Chooser is similar in operation to the Mac OS Chooser. You can use the function `GetRoot().NetChooser:OpenNetChooser` to display a list of network entities from which the user can make a selection. This function is declared as follows:

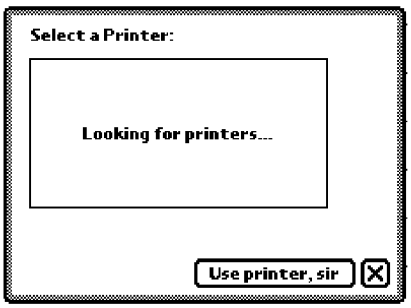
`NetChooser:OpenNetChooser(`*zone*`,` *lookupName*`,` *startSelection*`,`
who`,` *connText*`,` *headerText*`,` *lookforText*`)`

The `OpenNetChooser` method displays the NetChooser view on the user's screen. The following is an example that shows the use of this function:

```
GetRoot().NetChooser:openNetChooser(nil,"=:LaserWriter@",ni
l, self, "Use printer, sir", "Printer", "printers");
```

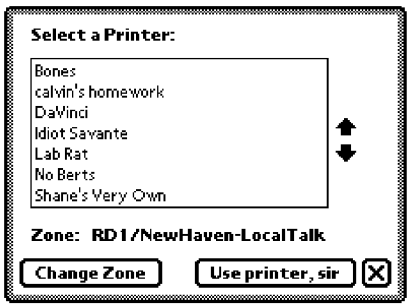
This example opens the NetChooser view and displays the *lookforText* string while the search is in progress, as shown in Figure 24-2 (page 24-14).

Figure 24-2 NetChooser view while searching



When the search has been completed, the NetChooser fills in the available choices and allows the user to make a selection, as shown in Figure 24-3.

Figure 24-3 NetChooser view displaying printers



After the user has made a selection, the system calls a method that you provide named `NetworkChooserDone`. The system fills in the parameters to this method with the name of the selection and zone chosen by the user. The `NetworkChooserDone` method must have the following format:

myChooser`:NetworkChooserDone(`*currentSelection*`,` *currentZone*`)`

The two parameters, *currentSelection* and *currentZone,* are filled in by the system after the user makes a choice.

The following is an example that shows the use of this function:

```
ChooserSample := {
      // open network connection
openNetworkScript: func()
begin
GetRoot().NetChooser:openNetChooser(nil,"=:LaserWriter@",ni
l, self, "Use printer, sir", "Printer", "printers");
end,

      // called when the user selects an item
networkChooserDone: func(currentSelection, currentZone)
begin
Print("Current Selection =" && currentSelection);
Print("Current Zone =" && currentZone);
end
};
```

The following is an example of running this code in the inspector:

```
ChooserSample:OpenNetworkScript()
#1A       TRUE

      // select the network entity, close the Chooser
"Current Selection = Idiot Savante"
"Current Zone = RD1/NewHaven-LocalTalk"
```

The NetChooser methods are described in detail in “NetChooser Methods” (page 21-81) in *Newton Programmer’s Reference*.

Summary

Built-in Communications Tool Service Option Labels

```
kCMSAsyncSerial                 "aser"
kCMSMNPID                       "mnps"
kCMSModemID                     "mods"
kCMSSlowIR                      "slir"
kCMSFramedAsyncSerial           "fser"
kCMSAppleTalkID                 "atlk"
```

Options

Asynchronous Serial Tool Options

```
kCMOSerialHWChipLoc             "schp"
kCMOSerialChipSpec              "sers"
kCMOSerialCircuitControl        "sctl"
kCMOSerialBuffers               "sbuf"
kCMOSerialIOParms               "siop"
kCMOSerialBitRate               "sbps "
kCMOOutputFlowControlParms      "oflc"
kCMOInputFlowControlParms       "iflc"
kCMOSerialBreak                 "sbrk"
kCMOSerialDiscard               "sdsc"
kCMOSerialEventEnables          "sevt"
kCMOSerialBytesAvailable        "sbav"
kCMOSerialIOStats               "sios"
kHMOSerExtClockDivide           "cdiv"
```

Serial with MNP Tool Options

```
kCMOMNPCompression              "mnpc"
kCMOMNPDataRate                 "eter"
```

Framed Serial Tool Options

kCMOFramingParms	"fram"
kCMOFramedAsyncStats	"frst"

Modem Options

kCMOModemPrefs	"mpre"
kCMOModemProfile	"mpro"
kCMOModemECType	"mecp"
kCMOModemDialing	"mdo"
kCMOModemConnectType	"mcto"
kCMOModemConnectSpeed	"mspd"
kCMOModemFaxCapabilities	"mfax"
kCMOModemFaxEnabledCaps	"mfec"
kCMOModemVoiceSupport	"mvso"
kCMOMNPSpeedNegotiation	"mnpn"
kCMOMNPCompression	"mnpc"
kCMOMNPStatistics	"mnps"

Infrared Tool Options

kCMOSlowIRConnect	"irco"
kCMOSlowIRProtocolType	"irpt"
kCMOSlowIRStats	"irst"

AppleTalk Tool Options

kCMARouteLabel	"rout"
kCMOAppleTalkBuffer	"bsiz"
kCMOSerialBytesAvailable	"sbav"
kCMSAppleTalkID	"atlk"
kCMOEndpointName	"endp"

Resource Arbitration Options

kCMOPassiveClaim	"cpcm"
kCMOPassiveState	"cpst"

Constants

Serial Chip Location Option Constants

```
kHWLocExternalSerial        "extr"
kHWLocBuiltInIR             "infr"
kHWLocBuiltInModem          "mdem"
kHWLocPCMCIASlot1           "slt1"
kHWLocPCMCIASlot2           "slt2"
```

Serial Chip Specification Option Constants

```
kSerCap_Parity_Space        0x00000001
kSerCap_Parity_Mark         0x00000002
kSerCap_Parity_Odd          0x00000004
kSerCap_Parity_Even         0x00000008

kSerCap_DataBits_5          0x00000001
kSerCap_DataBits_6          0x00000002
kSerCap_DataBits_7          0x00000004
kSerCap_DataBits_8          0x00000008
kSerCap_DataBits_All        0x0000000F

kSerCap_StopBits_1          0x00000010
kSerCap_StopBits_1_5        0x00000020
kSerCap_StopBits_2          0x00000040
kSerCap_DataBits_All        0x00000070

kSerialChip8250             0x00
kSerialChip16450            0x01
kSerialChip16550            0x02
kSerialChip8530             0x20
kSerialChip6850             0x21
kSerialChip6402             0x22
kSerialChipUnknown          0x00
```

Serial Circuit Control Option Constants

kSerOutDTR	0x01
kSerOutRTS	0x02
kSerInDSR	0x02
kSerInDCD	0x08
kSerInRI	0x10
kSerInCTS	0x20
kSerInBreak	0x80

Serial Configuration Option Constants

k1StopBits	0
k1pt5StopBits	1
k2StopBits	2
kNoParity	0
kOddParity	1
kEvenParity	2
k5DataBits	5
k6DataBits	6
k7DataBits	7
k8DataBits	8
kExternalClock	1
k300bps	300
k600bps	600
k1200bps	1200
k2400bps	2400
k4800bps	4800
k7200bps	7200
k9600bps	9600
k12000bps	12000
k14400bps	14400
k19200bps	19200
k38400bps	38400
k57600bps	57600
k115200bps	115200
k230400bps	230400

Serial Event Configuration Option Constants

kSerialEventBreakStartedMask	0x00000001
kSerialEventBreakEndedMask	0x00000002
kSerialEventDCDNegatedMask	0x00000004
kSerialEventDCDAssertedMask	0x00000008
kSerialEventHSKiNegatedMask	0x00000010
kSerialEventHSKiAssertedMask	0x00000020
kSerialEventExtClkDetectEnableMask	0x00000040

Serial External Clock Divide Option Constants

kSerClk_Default	0x00
kSerClk_DivideBy_1	0x80
kSerClk_DivideBy_16	0x81
kSerClk_DivideBy_32	0x82
kSerClk_DivideBy_64	0x83

Modem Error Control Type Option Constants

kModemECProtocolNone	0x00000001
kModemECProtocolMNP	0x00000002
kModemECProtocolExternal	0x00000008

Modem Fax Capabilities Option Constants

kModemFaxClass0	0x00000001
kModemFaxClass1	0x00000002
kModemFaxClass2	0x00000004
kModemFaxClass2_0	0x00000008
kV21Ch2Mod	0x00000001
kV27Ter24Mod	0x00000002
kV27Ter48Mod	0x00000004
kV29_72Mod	0x00000008
kV17_72Mod	0x00000010
kV17st_72Mod	0x00000020
kV29_96Mod	0x00000040
kV17_96Mod	0x00000080
kV17st_96Mod	0x00000100

kV17_12Mod	0x00000200
kV17st_12Mod	0x00000400
kV17_14Mod	0x00000800
kV17st_14Mod	0x00001000

MNP Compression Option Constants

kMNPCompressionNone	0x00000001
kMNPCompressionMNP5	0x00000002
kMNPCompressionV42bis	0x00000008

Infrared Protocol Type Option Constants

kUsingNegotiateIR	0
kUsingSharpIR	1
kUsingNewton1	2
kUsingNewton2	4
kUsing9600	1
kUsing19200	2
kUsing38400	4

Functions and Methods

AppleTalk Driver Functions

`OpenAppleTalk()`
`CloseAppleTalk()`
`AppleTalkOpenCount()`

AppleTalk Zone Information Methods

`HaveZones()`
`GetMyZone()`
`GetZoneList()`
`GetNames(`*fromWhat*`)`
`GetZoneFromName(`*fromWhat*`)`
`NBPStart(`*entity*`)`
`NBPGetCount(`*lookupID*`)`
`NBPGetNames(`*lookupID*`)`
`NBPStop(`*lookupID*`)`

NetChooser Function

`NetChooser:OpenNetChooser(`*zone*`,` *lookupName*`,` *startSelection*`,`
who`,` *connText*`,` *headerText*`,` *lookforText*`)`

CHAPTER 25

Modem Setup Service

This chapter contains information about the modem setup capability in Newton system software. You need to read this chapter if you want to define a modem setup package for your application. The built-in modem communications tool uses these packages for communicating with modems. For more information about the built-in modem communications tool, see "Built-in Communications Tools" (page 24-1).

This chapter describes:

- The modem setup service and how it works with modem setup packages.
- The user interface for modem setup.
- The modem characteristics required by the Newton modem tool.
- The constants you use in defining a modem setup. These constants are described in detail in "Modem Setup Service Reference" (page 22-1) in *Newton Programmer's Reference*.

About the Modem Setup Service

This section provides detailed conceptual information on the modem setup service. Specifically, it covers the following:

- a description of the modem setup user interface
- the programmatic process by which a modem is setup
- modem requirements

The modem setup service allows many different kinds of modems to be used with Newton devices. Each kind of modem can have an associated modem setup package, which can configure a modem endpoint to match the particular modem.

A modem setup package is installed on the Newton as an automatically loaded package. This means that when the package is loaded, the modem setup information is automatically stored in the system soup and then the package is removed. No icon appears for the modem setup in the Extras Drawer. Instead, modem setups are accessed through a picker in the Modem preferences view.

Modem setup packages can be supplied by modem manufacturers, or can be created by other developers.

A modem setup package can contain four parts:

- **General information.** The beginning of a modem setup package specifies general information about the modem corresponding to the package—for example, the modem's name and version number.
- **A modem tool preferences option.** The part of the package that contains specifications that configure the modem controller. For a description of this option, see "Modem Preferences Option" (page 21-34) in *Newton Programmer's Reference*.
- **A modem tool profile option.** This part of the package describes the characteristics of the modem—for example, whether the modem supports error correction protocols. For more information on this option, see the section "Modem Profile Option" (page 21-38) in *Newton Programmer's Reference*.
- **A fax profile option.** This part of the package describes the characteristics of the fax—for example, the speed at which faxes can be sent and received. This option is particularly useful to limit fax speeds over cellular connections.

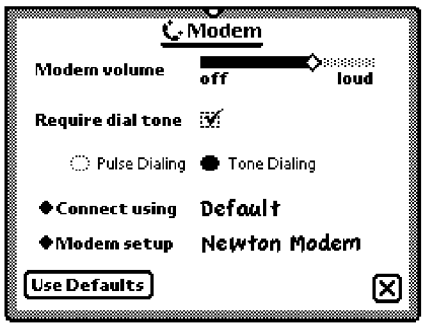
If a modem supports both cellular and landline operations and does not automatically configure itself, you need to create a separate modem profile or setup for each operation. If you want to give the user the option to limit fax speeds, which is a common practice with cellular connections, you may want a third profile that specifies the fax profile option.

Note
The constants and code shown in this chapter apply to the NTK project that is provided by Newton Technical Support. This project provides an easy way to create modem setups. ◆

The Modem Setup User Interface

The user chooses the current modem setup in the Modem preferences, as shown in Figure 25-1 (page 25-3). The Modem Setup item is a picker, which when tapped displays all of the modem setups installed in the system. The chosen modem setup is the default used by all applications.

Figure 25-1 Modem preferences view



The Modem Setup Process

All communication applications that use a modem endpoint make use of the modem setup service. The current modem setup is automatically invoked when an application calls the modem endpoint's `Instantiate` method.

Note
If the modem endpoint option list includes the modem profile option (`kCMOModemProfile`), the modem setup is not invoked. This allows modem applications to override the modem setup when configuring the modem for special purposes. ◆

Here is what happens in the `Instantiate` method when the modem setup is invoked:

1. The `kCMOModemPrefs` option is added to the endpoint configuration options, and the `fEnablePassThru` field is set to `true`. This enables the endpoint to operate in pass-through mode. In this mode, the modem endpoint is functionally equivalent to a serial endpoint for input and output.
2. The modem endpoint is instantiated and connected in pass-through mode.
3. The Newton system software sets the modem preferences (`kCMOModemPrefs`), modem profile (`kCMOModemProfile`), and fax profile (`kCMOModemFaxCapabilities`) options as defined in the modem setup.

Note
A modem setup method is executed only once—when the endpoint is instantiated—even if the endpoint is subsequently used for multiple connections. ◆

4. The modem endpoint is reconfigured with pass-through mode disabled, and control is returned to the client application, which can proceed with its `Bind` and `Connect` calls.

"Defining a Modem Setup" (page 25-5) describes how to define a modem setup.

Modem Communication Tool Requirements

The Newton modem communication tool expects certain characteristics from a modem. These characteristics are described here.

- The modem tool expects a PCMCIA modem to use a 16450 or 16550 UART chip.
- The modem tool expects hardware flow control in both serial and PCMCIA modems. In modems not supporting hardware flow control, direct connect support is required, and the modem profile constant `kDirectConnectOnly` must be set to `true`. This means that the modem tool and the modem must be running at the same bit rate, allowing for no compression or error correction protocols to be used by the modem. (When operating in direct connect mode, the data rate of the modem tool is automatically adjusted to the data rate stated in the "CONNECT SEXTETS" message.)
- The modem tool expects control signals to be used as follows:
  - The modem tool uses RTS to control data flow from the modem.
  - The modem uses CTS to control data flow from the modem tool.
  - Support of the DCD signal is optional. In general, the modem tool expects DCD to reflect the actual carrier state. The usage of this signal by the modem tool is governed by the `kUseHardwareCD` constant.
- The modem tool expects non-verbose textual responses from the modem.
- The modem tool expects no echo.
- The modem tool currently supports the Class 1 protocol for FAX connections; under some circumstance (see the note below), the modem tool supports the Class 2 protocol. The configuration string defined by the constant `kConfigStrNoEC` is used for sending and receiving FAX documents. Additionally, these other requirements apply to the FAX service:
  - Flow control is required. In modems not supporting hardware flow control (where `kDirectConnectOnly` = `true`), XON/XOFF software flow control must be enabled.
  - Buffering must be enabled.
  - The `kConfigSpeed` constant must be set to higher than the highest connect rate of which the modem is capable. For example, if the modem supports 14400, set `kConfigSpeed` to 19200; if the modem supports 28800, set `kConfigSpeed` to 54600.

CHAPTER 25

Modem Setup Service

Note
The modem tool has been upgraded to support the Class 2 and Class 2.0 FAX protocols in release 2.1 of the Newton System Software. This upgrade is also available in the German version of release 2.0 of the Newton System Software. To enable the use of these protocols, you must define the fax profile in your modem setup. ◆

Defining a Modem Setup

The parts of a modem setup are specified in a Newton Toolkit (NTK) text file, which is provided by Newton Technical Support. The modem preferences and profile options are specified by setting constants. The following sections describe each part of the modem setup.

Setting Up General Information

The beginning of a modem setup contains general information about the setup and the modem to which it corresponds. Here is an example:

```
constant kModemName      := "Speedy Fast XL";
constant kVersion        := 1;
constant kOrganization   := "Speedy Computer, Inc.";
```

The value of `kModemName` appears in the Modem preferences. It is usually the name of the modem. The constant `kVersion` identifies the (integer-only) version of the modem setup package. The constant `kOrganization` indicates the source of the modem setup package. For detailed descriptions of these constants, see "Modem Setup General Information Constants" (page 22-2) in *Newton Programmer's Reference*.

Setting the Modem Preferences Option

This modem option configures the modem controller. Here is an example:

```
constant kIdModem              :=  nil;
constant kUseHardwareCD        :=  true;
constant kUseConfigString      :=  true;
constant kUseDialOptions       :=  true;
constant kHangUpAtDisconnect   :=  true;
```

For detailed descriptions of these constants, see "Modem Setup Preference Constants" (page 22-3) in *Newton Programmer's Reference*. For more information

about the modem preferences option, see "Modem Preferences Option" (page 21-34) in *Newton Programmer's Reference*.

Setting the Modem Profile Option

This modem profile option describes the modem characteristics, to be used by the modem controller. Here is an example:

```
constant kSupportsEC        := true;
constant kSupportsLCS       := nil;
constant kDirectConnectOnly:= nil;
constant kConnectSpeeds     := '[300, 1200, 2400, 4800,
                               7200, 9600, 12000, 14400];
constant kConfigSpeed       := 38400;
constant kCommandTimeout    := 2000;
constant kMaxCharsPerLine   := 40;
constant kInterCmdDelay     := 25;
constant kModemIDString     := "unknown";
constant kConfigStrNoEC          :=
                          "ATE0&A0&B1&C1&H1&M0S12=12\n";
constant kConfigStrECOnly        :=
                          "ATE0&A0&B1&C1&H1&M5S12=12\n";
constant kConfigStrECAndFallbac  :=
                          "ATE0&A0&B1&C1&H1&M4S12=12\n";
constant kConfigStrDirectConnec  :=
                          "ATE0&A0&B0&C1&H0&M0S12=12\n";
```

For detailed descriptions of these constants, see "Modem Setup Profile Constants" (page 22-4) in *Newton Programmer's Reference*. For more information about the modem preferences option, see "Modem Profile Option" (page 21-38) in *Newton Programmer's Reference*.

When the modem tool establishes communication with a modem through an endpoint, the tool normally sends a configuration string to the modem (as long as `kUseConfigString` is `true`). Several configuration strings are defined in a typical modem profile; the one that is sent depends on the type of connection requested and other parameters set in the modem profile. Table 25-1 summarizes when each kind of configuration string is used:

Table 25-1 Summary of configuration string usage

Configuration string	When used
`kConfigStrNoEC`	The default configuration used for data connections when `kDirectConnectOnly` is `nil`. Also used for FAX connections. See "The No Error Control Configuration String" (page 22-7) in *Newton Programmer's Reference* for an example.
`kConfigStrECOnly`	Used for data connections that require error correction. This configuration string is used only if requested by an application. The constant `kSupportsEC` must be `true` for this configuration string to be used. See "The Error Control Configuration String" (page 22-8) in *Newton Programmer's Reference* for an example.
`kConfigStrECAndFallback`	Used for data connections that allow error correction, but that can fall back to non-error-corrected mode. This configuration string is used only if requested by an application. See "The Error Control with Fallback Configuration String" (page 22-9) in *Newton Programmer's Reference* for an example.
`kConfigStrDirectConnect`	The default configuration used for data connections when `kDirectConnectOnly` is `true`. See "The Direct Connect Configuration String" (page 22-9) in *Newton Programmer's Reference* for an example.

Setting the Fax Profile Option

The fax profile option describes the fax characteristics to be used by the fax tool. Here is an example:

```
constant kTransmitDataMod  :=
               kV21Ch2Mod + KV27Ter24Mod+ kV27Ter48Mod;
constant kReceiveDataMod:=
               kV21Ch2Mod + KV27Ter24Mod + kV27Ter48Mod;
constant kServiceClass  :=
               kModemFaxClass1 + kModemFaxClass2;
```

This example limits the faxing to 4800 bps for both send and receive messages. If neither of these constants is defined, then the fax send and receive speeds are not restricted.

Note
You can only set the service class (use the `kServiceClass` constant) for versions of the software that support the Class 2 fax protocol. Newton System Software version 2.1 and the German version of Newton System Software version 2.0 support the Class 2 fax protocol. ◆

For detailed descriptions of these constants, see "Fax Profile Constants" (page 22-10) in *Newton Programmer's Reference*.

The constants that you can use to specify speeds in defining your fax profile values are shown in Table 22-5 (page 22-11) in *Newton Programmer's Reference*.

Summary of the Modem Setup Service

Constants

Constants for Modem Setup General Information

```
kModemName
kVersion
kOrganization
```

Constants for Modem Setup Preferences

```
kIdModem
kUseHardwareCD
kUseConfigString
kUseDialOptions
kHangUpAtDisconnect
```

Constants for the Modem Setup Profile

```
kSupportsEC
kSupportsLCS
kDirectConnectOnly
kXonnectSpeeds
kXommandTimeout
kMaxCharsPerLine
kInterCmdDelay
kModemIDString
kConfigStrNoEC
kConfigStrECOnly
kConfigStrECAndFallback
kConfigStrDirectConnect
```

Constants for the fax profile

```
kTransmitDataMod
kReceiveDataMod
kServiceClass
```

Fax Speed Constants

```
kV21Ch2Mod
kv27Ter24Mod
kV27Ter48Mod
kV29_72Mod
kV17_72Mod
kV17st_72Mod
kV29_96Mod
kV17_96Mod
kV17st_96Mod
kV17_12Mod
kV17st_12Mod
kV17st_14Mod
```

Fax Class Constants

```
kModemFaxClass0
kModemFaxClass1
kModemFaxClass2
kModemFaxClass2_0
```

CHAPTER 26

Utility Functions

This chapter provides a listing of a number of utility functions documented in the "Utility Functions Reference" in the *Newton Programmer's Reference*. The following groups of functions are included:

- Object system
- String
- Bitwise
- Array and sorted array
- Integer Math
- Floating point math
- Control of floating point math
- Financial
- Exception handling
- Message sending and deferred message sending
- Data extraction
- Data stuffing
- Getting and Setting Global Variables
- Miscellaneous

Four of the functions described in the Object system section are designed to clone, or copy, objects. These functions each behave slightly differently. Table 26-1 summarizes their actions. The "Recurs" column indicates if references within the object are copied. The "Follows magic pointers" column indicates if objects referenced through magic pointers are copied. The "Ensures object is internal" column indicates if the function ensures that all parts of the object exist in internal RAM or ROM. The "Copies object" column indicates if the object is copied.

Table 26-1 Summary of copying functions

Function name	Recurs	Follows magic pointers	Ensures object is internal	Copies object
`Clone`	—	—	—	yes
`DeepClone`	yes	yes	—	yes
`EnsureInternal`	yes	—	yes	as needed
`TotalClone`	yes	—	yes	yes

Compatibility

This section describes the changes to the utility functions for Newton System Software 2.0.

New Functions

The following new functions have been added for this release.

New Object System Functions

The following new object system functions have been added.

`GetFunctionArgCount`
`IsCharacter`
`IsFunction`
`IsInteger`
`IsNumber`
`IsReadOnly` (existed in 1.0 but now documented)
`IsReal`
`IsString`
`IsSubclass` (existed in 1.0 but now documented)
`IsSymbol`
`MakeBinary`
`SetVariable`
`SymbolCompareLex`

New String Functions

The following new string functions have been added.

```
CharPos
LatitudeToString
LongitudeToString
StrExactCompare
StrFilled (existed in 1.0 but now documented)
StrTokenize
StyledStrTruncate
SubstituteChars
```

New Array Functions

The following new array functions have been added.

```
ArrayInsert
InsertionSort
LFetch
LSearch
NewWeakArray
StableSort
```

New Sorted Array Functions

The following new functions have been added that operate on sorted arrays. These functions are based on binary search algorithms, hence the "B" prefix to the function names.

```
BDelete
BDifference
BFetch
BFetchRight
BFind
BFindRight
BInsert
BInsertRight
BIntersect
BMerge
BSearchLeft
BSearchRight
```

New Integer Math Functions

The following new functions related to integer math have been added.

```
GetRandomState
SetRandomState
```

New Financial Functions

The following new functions that perform operations related to the currency exchange rate have been added.

```
GetExchangeRate
SetExchangeRate
GetUpdatedExchangeRates
```

New Exception Handling Functions

The following new exception handling function has been added.

```
RethrowWithUserMessage
```

New Message Sending Functions

The following new utility functions for sending immediate messages have been added.

```
IsHalting
PerformIfDefined
ProtoPerform
ProtoPerformIfDefined
```

New Deferred Message Sending Functions

The following new utility functions for delayed and deferred actions have been added.

```
AddDeferredCall
AddDelayedCall
AddProcrastinatedCall
AddDeferredSend
AddDelayedSend
AddProcrastinatedSend
```

These new functions replace `AddDelayedAction` and `AddDeferredAction` (although both remain in the ROM for compatibility with existing applications). These two older functions have several problems, and you should not use them—they will likely be removed in future versions of system software.

New Data Stuffing Functions

The following new data stuffing functions have been added.

```
StuffCString
StuffPString
```

New Functions to Get and Set Globals

The following new functions that get, set, and check for the existence of global variables and functions have been added.

```
GetGlobalFn
GetGlobalVar
GlobalFnExists
GlobalVarExists
DefGlobalFn
DefGlobalVar
UnDefGlobalFn
UnDefGlobalVar
```

New Debugging Functions

The following debugging functions have been added.

```
StrHexDump
TrueSize
ViewAutopsy
```

The following debugging functions have been changed.

```
StackTrace
BreakLoop
```

New Miscellaneous Functions

The following miscellaneous functions have been added.

```
AddMemoryItem
AddMemoryItemUnique
Backlight
BacklightStatus
BinEqual
Gestalt
GetAppName
GetAppPrefs
GetMemoryItems
```

GetMemorySlot
MakePhone
MakeDisplayPhone
ParsePhone
PowerOff
Translate

Enhanced Functions

The following string function has been enhanced in Newton 2.0.

ParamStr has been enhanced to support conditional substitution.

Obsolete Functions

Some utility functions previously documented in the *Newton Programmer's Guide* are obsolete, but are still supported for compatibility with older applications. Do not use the following utility functions, as they may not be supported in future system software versions:

AddDeferredAction (use AddDeferredCall instead)
AddDelayedAction (use AddDelayedCall instead)
AddPowerOffHandler (use RegPowerOff instead)
ArrayPos (use LSearch instead)
GetGlobals (use GetGlobalVar or GetGlobalFn instead)
RemovePowerOffHandler (use UnRegPowerOff instead)
SmartStart (use other string manipulation functions)
SmartConcat (use other string manipulation functions)
SmartStop (use other string manipulation functions)
StrTruncate (use StyledStrTruncate instead)
StrWidth (use StrFontWidth instead)

Summary of Functions and Methods

Object System Functions

`ClassOf(`*object*`)`
`Clone(`*object*`)`
`DeepClone(`*object*`)`
`EnsureInternal(`*obj*`)`
`GetFunctionArgCount(`*function*`)`
`GetSlot(`*frame*`, `*slotSymbol*`)`
`GetVariable(`*frame*`, `*slotSymbol*`)`
`HasSlot(`*frame*`, `*slotSymbol*`)`
`HasVariable(`*frame*`, `*slotSymbol*`)`
`Intern( `*string*` )`
`IsArray(`*obj*`)`
`IsBinary(`*obj*`)`
`IsCharacter(`*obj*`)`
`IsFrame(`*obj*`)`
`IsFunction(`*obj*`)`
`IsImmediate(`*obj*`)`
`IsInstance(`*obj*`, `*class*`)`
`IsInteger(`*obj*`)`
`IsNumber(`*obj*`)`
`IsReadOnly(`*obj*`)`
`IsReal(`*obj*`)`
`IsString(`*obj*`)`
`IsSubclass(`*sub*`, `*super*`)`
`IsSymbol(`*obj*`)`
`MakeBinary(`*length*`, `*class*`)`
`Map(`*obj*`, `*function*`)`
`PrimClassOf(`*obj*`)`
`RemoveSlot(`*obj*`, `*slot*`)`
`ReplaceObject(`*originalObject*`, `*targetObject*`)`
`SetClass(`*obj*`, `*classSymbol*`)`
`SetVariable(`*frame*`, `*slotSymbol*`, `*value*`)`
`SymbolCompareLex(`*symbol1*`, `*symbol2*`)`
`TotalClone(`*obj*`)`

String Functions

BeginsWith(*string*, *substr*)
Capitalize(*string*)
CapitalizeWords(*string*)
CharPos(*str*, *char*, *startpos*)
Downcase(*string*)
EndsWith(*string*, *substr*)
EvalStringer(*frame*, *array*)
FindStringInArray(*array*, *string*)
FindStringInFrame(*frame*, *stringArray*, *path*)
FormattedNumberStr(*number*, *formatString*)
IsAlphaNumeric(*char*)
IsWhiteSpace(*char*)
LatitudeToString(*latitude*)
LongitudeToString(*longitude*)
NumberStr(*number*)
ParamStr(*baseString*, *paramStrArray*)
SPrintObject(*obj*)
StrCompare(*a*, *b*)
StrConcat(*a*, *b*)
StrEqual(*a*, *b*)
StrExactCompare(*a*, *b*)
StrFilled(*string*)
StrFontWidth(*string*, *fontSpec*)
Stringer(*array*)
StringFilter(*str*, *filter*, *instruction*)
StringToNumber(*string*)
StrLen(*string*)
StrMunger(*dstString, dstStart, dstCount, srcString, srcStart, srcCount*)
StrPos(*string*, *substr*, *start*)
StrReplace(*string*, *substr*, *replacement*, *count*)
StrTokenize(*str*, *delimiters*)
StyledStrTruncate(*string*, *length*, *font*)
SubstituteChars(*targetStr*, *searchStr*, *replaceStr*)
SubStr(*string*, *start*, *count*)
TrimString(*string*)
Upcase(*string*)

CHAPTER 26

Utility Functions

Bitwise Functions

Band(*a*, *b*)
Bor(*a*, *b*)
Bxor(*a*, *b*)
Bnot(*a*)

Array Functions

AddArraySlot (*array*, *value*)
Array(*size*, *initialValue*)
ArrayInsert(*array*, *element*, *position*)
ArrayMunger(*dstArray*, *dstStart*, *dstCount*, *srcArray*, *srcStart*, *srcCount*)
ArrayRemoveCount(*array*, *startIndex*, *count*)
InsertionSort(*array*, *test*, *key*)
Length (*array*)
LFetch(*array*, *item*, *start*, *test*, *key*)
LSearch(*array*, *item*, *start*, *test*, *key*)
NewWeakArray(*length*)
SetAdd (*array*, *value*, *uniqueOnly*)
SetContains(*array*, *item*)
SetDifference(*array1*, *array2*)
SetLength (*array*, *length*)
SetOverlaps(*array1*, *array2*)
SetRemove (*array*, *value*)
SetUnion(*array1*, *array2*, *uniqueFlag*)
Sort(*array*, *test*, *key*)
StableSort(*array*, *test*, *key*)

Sorted Array Functions

BDelete(*array*, *item*, *test*, *key*, *count*)
BDifference(*array1*, *array2*, *test*, *key*)
BFetch(*array*, *item*, *test*, *key*)
BFetchRight(*array*, *item*, *test*, *key*)
BFind(*array*, *item*, *test*, *key*)
BFindRight(*array*, *item*, *test*, *key*)
BInsert(*array*, *element*, *test*, *key*, *uniqueOnly*)
BInsertRight(*array*, *element*, *test*, *key*, *uniqueOnly*)
BIntersect(*array1*, *array2*, *test*, *key*, *uniqueOnly*)

`BMerge(`*array1*`, `*array2*`, `*test*`, `*key*`, `*uniqueOnly*`)`
`BSearchLeft(`*array*`, `*item*`, `*test*`, `*key*`)`
`BSearchRight(`*array*`, `*item*`, `*test*`, `*key*`)`

Integer Math Functions

`Abs(`*x*`)`
`Ceiling(`*x*`)`
`Floor(`*x*`)`
`GetRandomState()`
`Max( `*a*`, `*b*` )`
`Min( `*a*`, `*b*` )`
`Real(`*x*`)`
`Random (`*low*`, `*high*`)`
`SetRandomSeed (`*seedNumber*`)`
`SetRandomState(`*randomState*`)`

Floating Point Math Functions

`Acos(`*x*`)`
`Acosh(`*x*`)`
`Asin(`*x*`)`
`Asinh(`*x*`)`
`Atan(`*x*`)`
`Atan2(`*x*`,`*y*`)`
`Atanh(`*x*`)`
`CopySign(`*x*`,`*y*`)`
`Cos(`*x*`)`
`Cosh(`*x*`)`
`Erf(`*x*`)`
`Erfc(`*x*`)`
`Exp(`*x*`)`
`Expm1(`*x*`)`
`Fabs(`*x*`)`
`FDim(`*x*`,`*y*`)`
`FMax(`*x*`,`*y*`)`
`FMin(`*x*`,`*y*`)`
`Fmod(`*x*`,`*y*`)`
`Gamma(`*x*`)`
`Hypot(`*x*`,`*y*`)`

IsFinite(*x*)
IsNaN(*x*)
IsNormal(*x*)
LessEqualOrGreater(*x*, *y*)
LessOrGreater(*x*, *y*)
LGamma(*x*)
Log(*x*)
Logb(*x*)
Log1p(*x*)
Log10(*x*)
NearbyInt(*x*)
NextAfterD(*x*,*y*)
Pow(*x*,*y*)
RandomX(*x*)
Remainder(*x*,*y*)
RemQuo(*x*,*y*)
Rint(*x*)
RintToL(*x*)
Round(*x*)
Scalb(*x*, *k*)
SignBit(*x*)
Signum(*x*)
Sin(*x*)
Sinh(*x*)
Sqrt(*x*)
Tan(*x*)
Tanh(*x*)
Trunc(*x*)
Unordered(*x*, *y*)
UnorderedGreaterOrEqual(*x*, *y*)
UnorderedLessOrEqual(*x*, *y*)
UnorderedOrEqual(*x*, *y*)
UnorderedOrGreater(*x*, *y*)
UnorderedOrLess(*x*, *y*)
FeClearExcept(*excepts*)
FeGetEnv()
FeGetExcept(*excepts*)
FeHoldExcept()
FeRaiseExcept(*excepts*)
FeSetEnv(*envObj*)
FeSetExcept(*flagObj*, *excepts*)

`FeTestExcept(`*excepts*`)`
`FeUpdateEnv(`*envObj*`)`

Financial Functions

`Annuity(`*r*`, `*n*`)`
`Compound(`*r*`, `*n*`)`
`GetExchangeRate(`*country1*`, `*country2*`)`
`SetExchangeRate(`*country1*`, `*country2*`, `*rate*`)`
`GetUpdatedExchangeRates()`

Exception Functions

`Throw(`*name*`, `*data*`)`
`Rethrow()`
`CurrentException()`
`RethrowWithUserMessage(`*userTitle*`, `*userMessage*`, `*override*`)`

Message Sending Functions

`Apply(`*function*`, `*parameterArray*`)`
`IsHalting(`*functionObject*`, `*args*`)`
`Perform(`*frame*`, `*message*`, `*parameterArray*`)`
`PerformIfDefined(`*receiver*`, `*message*`, `*paramArray*`)`
`ProtoPerform(`*receiver*`, `*message*`, `*paramArray*`)`
`ProtoPerformIfDefined(`*receiver*`, `*message*`, `*paramArray*`)`

Deferred Message Sending Functions

`AddDeferredCall(`*functionObject*`, `*paramArray*`)`
`AddDelayedCall(`*functionObject*`, `*paramArray*`, `*delay*`)`
`AddDeferredSend(`*receiver*`, `*message*`, `*paramArray*`)`
`AddDelayedSend(`*receiver*`, `*message*`, `*paramArray*`, `*delay*`)`
`AddProcrastinatedCall(`*funcSymbol*`, `*functionObject*`, `*paramArray*`, `*delay*`)`
`AddProcrastinatedSend(`*msgSymbol*`, `*receiver*`, `*message*`, `*paramArray*`, `*delay*`)`

Data Extraction Functions

`ExtractByte(`*data*`, `*offset*`)`
`ExtractBytes(`*data*`, `*offset*`, `*length*`, `*class*`)`
`ExtractChar(`*data*`, `*offset*`)`
`ExtractLong(`*data*`, `*offset*`)`
`ExtractXLong(`*data*`, `*offset*`)`
`ExtractWord(`*data*`, `*offset*`)`
`ExtractCString(`*data*`, `*offset*`)`
`ExtractPString(`*data*`, `*offset*`)`
`ExtractUniChar(`*data*`, `*offset*`)`

Data Stuffing Functions

`StuffByte(`*obj*`, `*offset*`, `*toInsert*`)`
`StuffChar(`*obj*`, `*offset*`, `*toInsert*`)`
`StuffCString(`*obj*`, `*offset*`, `*aString*`)`
`StuffLong(`*obj*`, `*offset*`, `*toInsert*`)`
`StuffPString(`*obj*`, `*offset*`, `*aString*`)`
`StuffUniChar(`*obj*`, `*offset*`, `*toInsert*`)`
`StuffWord(`*obj*`, `*offset*`, `*toInsert*`)`

Getting and Setting Global Variables and Functions

`GetGlobalFn(`*symbol*`)`
`GetGlobalVar(`*symbol*`)`
`GlobalFnExists(`*symbol*`)`
`GlobalVarExists(`*symbol*`)`
`DefGlobalFn(`*symbol*`, `*function*`)`
`DefGlobalVar(`*symbol*`, `*value*`)`
`UnDefGlobalFn(`*symbol*`)`
`UnDefGlobalVar(`*symbol*`)`

Debugging Functions

`BreakLoop()`
`DV(`*view*`)`
`GC()`
`ExitBreakLoop()`
`StackTrace()`

`Stats()`
`StrHexDump(`*object*`,` *spaceInterval*`)`
`TrueSize(`*object*`,` *filter*`)`
`ViewAutopsy(`*functionSpec*`)`

Miscellaneous Functions

`AddMemoryItem(`*memSymbol*`,` *value*`)`
`AddMemoryItemUnique(`*memorySlot*`,` *value*`,` *testFunc*`)`
`Backlight()`
`BacklightStatus(`*state*`)`
`BinEqual(`*a*`,` *b*`)`
`BinaryMunger(` *dst*`,` *dstStart*`,` *dstCount*`,` *src*`,` *srcStart*`,` *srcCount* `)`
`Chr(`*integer*`)`
`Compile(`*string*`)`
`Gestalt(`*selector*`)`
`GetAppName(`*appSymbol*`)`
`GetAppParams( )`
`GetAppPrefs(`*appSymbol*`,` *defaultFrame*`)`
`GetMemoryItems(`*memSymbol*`)`
`GetMemorySlot(`*memorySlot*`,` *op*`)`
`GetPrinterName(`*printerFrame*`) //platform file function`
`MakePhone (`*phoneFrame*`)`
`MakeDisplayPhone(`*phoneStr*`)`
rootView`:MungePhone(`*inNum*`,` *country*`)`
`ParsePhone(`*phoneStr*`)`
`PowerOff(` *reason* `)`
`Ord (`*char*`)`
`RegEmailSystem(`*classSymbol*`,` *name*`,` *internet*`)`
`RegPagerType(`*classSymbol*`,` *name*`)`
`RegPhoneType with (`*classSymbol*`,` *name*`,` *number*`)`
`ShowManual()`
`Sleep(`*ticks*`)`
rootView`:SysBeep()`
`Translate(`*data*`,` *translator*`,` *store*`,` *callback*`)`
`UnRegEmailSystem(`*classSymbol*`)`
`UnregPagerType(`*classSymbol*`)`
`UnregPhoneTypeFunc(`*classSymbol*`)`

A P P E N D I X

The Inside Story on Declare

This appendix describes the technical details of the declare mechanism. Knowing these technical details is not necessary to understanding what declaring a view means; they are provided primarily for completeness and to help you when you are debugging. You shouldn't write code that depends on these details.

For a basic discussion of the declare mechanism, see the section "View Instantiation" beginning on page 3-26. You should be familiar with that material before reading this appendix.

To serve as an example here, imagine a calculator application whose base view is named "Calculator." It has (among others) a child view named "Display." The Display view is declared in the Calculator view. See Figure A-1 for an illustration of this example.

In the following sections, we'll explain what happens at compile time and at run time as a result of the declare operation. A number of slots are created, which you may see in the Newton Toolkit (NTK) Inspector if you are examining the view templates.

Compile-Time Results

As a result of the declare operation, at compile time, NTK creates a slot in the place where the view is declared—that is, in the Calculator template. The name of the slot is the name of the declared view, `Display`. This slot's value is initially set to `nil`.

Another slot, called `stepAllocateContext`, is also created in the Calculator template. This slot holds an array of values (two for each view declared there). The first value in each pair is a symbol used by the system at run time to identify the name of the slot in the Calculator view that holds a reference to the declared view. This symbol is simply the name of the declared view, `Display`.

The second value in each pair is a reference to the template for the declared view. At run time, the system will preallocate a view memory object for the declared view from this template.

Note
Protos built into the system use an analogous slot called `allocateContext`, that holds the same thing as `stepAllocateContext`. The `allocateContext` slot is for declared children from the `viewChildren` array and the `stepAllocateContext` slot is for declared children from the `stepChildren` array. ◆

Also, as a result of the declare operation, NTK creates a slot in the Display template called `preallocatedContext`. This slot holds a symbol that is the name of the template, in this case `'Display`. This symbol will be used by the system when the view is instantiated to find the preallocated view memory object for the Display view.

Run-Time Results

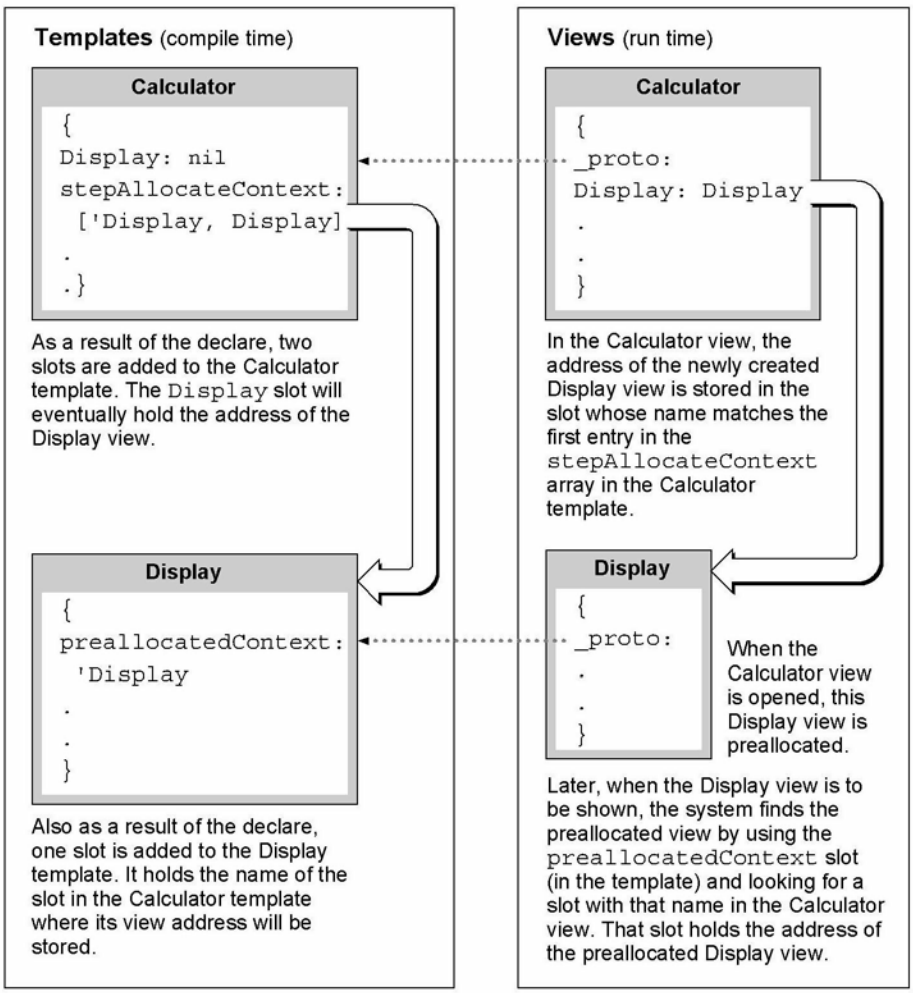
When the Calculator view is opened (even before its `ViewSetupFormScript` method is executed), a view memory object is preallocated for each view declared in Calculator. (The information required to do this is obtained from the `allocateContext` and `stepAllocateContext` slots.) In our example, a view memory object is created for the Display view.

The `Display` slot in the Calculator view is updated so that it points to the newly allocated Display view object.

Later in the instantiation process for the Calculator view, its child views are created and shown, including the Display view. At this time, the view system looks at the template for the Display view, sees the `preallocatedContext` slot, and knows that a view memory object has been preallocated for this view. Using this slot, the system can find the preallocated view.

The value of the `preallocatedContext` slot is the name of another slot in the Calculator view. The system locates this slot in the Calculator view, and finds there a reference to the preallocated view object. Instead of creating a new view object for the Display view, the system uses the preallocated view.

Figure A-1 Declare example



Glossary

Action button The small envelope button used in applications to invoke routing functions. When tapped, it displays a picker listing routing actions available for the current item.

alias An object that consists of a reference to another object. An alias saves space, since the alias object is small, and can be used to reference very large objects. Resolving an alias refers to retrieving the object that the alias references. See also **entry alias.**

application base view The topmost parent view in an application. The application base view typically encloses all other views that make up the application.

arc A portion of the circumference of an oval bounded by a pair of radii joining at the oval's center. Contrast a **wedge**, which includes part of the oval's interior. Arcs and wedges are defined by the bounding rectangle that encloses the oval, along with a pair of angles marking the positions of the bounding radii.

array A sequence of numerically indexed slots (also known as the array elements) that contain objects. The first element is indexed by zero. Like other nonimmediate objects, an array can have a user-specified class, and can have its length changed dynamically.

away city The **emporium** that's displayed as a counterpoint to your **home city.** It defines such information as dialing area, time zone, and so on. Sometimes it is called the "I'm here" city.

binary object A sequence of bytes that can represent any kind of data, can be adjusted dynamically in size, and can have a user-specified class. Examples of binary objects include strings, real numbers, sounds, and bitmaps.

Boolean A special kind of immediate value. In NewtonScript, there is only one Boolean, called `true`. Functions and control structures use `nil` to represent false. When testing for a true/false value, `nil` represents false, and any other value is equivalent to `true`.

button host An application that receives buttons from other applications (**button providers**).

button provider An application that adds a button to another application (the **button host**).

callback spec A frame passed as an argument to an endpoint method. The callback spec frame contains slots that control how the endpoint method executes, along with a completion method that is called when the endpoint operation completes. See also **output spec**.

card Short for a **PCMCIA** card. Also, a view of information about an entry in the Names soup, formatted as a business card.

child A frame that references another frame (its parent) from a `_parent` slot. With regard to views, a child view is enclosed by its parent view.

class A symbol that describes the data referenced by an object. Arrays, frames, and binary objects can have user-defined classes.

constant A value that does not change. In NewtonScript the value of the constant is substituted wherever the constant is used in code.

cursor An object returned by the `Query` method. The cursor contains methods that iterate over a set of soup entries meeting the criteria specified in the query. The addition or deletion of entries matching the query specification is automatically reflected in the set of entries referenced by the cursor, even if the changes occur after the original query was made.

data definition A frame containing slots that define a particular type of data and the methods that operate on it. The entries defined are used by an application and stored in its soup. A data definition is registered with the system. The shortened term dataDef is sometimes used. See also **view definition**.

data form A symbol that describes the transformations that must occur when data is exchanged with other environments. When you send data or set endpoint options, the data form defines how to convert the data from its NewtonScript format. When you receive data or get endpoint options, the data form defines the type of data expected.

declaring a template Registering a template in another view (usually its parent) so that the template's view is preallocated when the other view is opened. This allows access to methods and slots in the declared view.

deferred recognition The process of recognizing an ink word that was drawn by the user at an earlier time. Deferred recognition is usually initiated when the user double-taps on an ink word. See also **ink** and **ink word.**

desktop computer Either a Mac OS or Windows-based computer. Sometimes called simply "desktop."

emporium The permanent internal descriptions of places the user works with the Newton PDA. (Home and Office are obvious examples, but so might be "Tokyo Office" if the user travels a lot.) Choosing an emporium sets up information such as local area code, dialing prefixes, time zone, and so on. This term is sometimes called "locale." The plural is "emporia."

endpoint An object created from `protoBasicEndpoint`, or one of its derivative protos, that controls a real-time communication session. This object encapsulates and maintains the details of the specific connection, and allows you to control the underlying communication tool.

endpoint option An endpoint option is specified in a frame passed in an array as an argument to one of the endpoint methods. Endpoint options select the communication tool to use, control its configuration and operation, and return result code information from each endpoint method call.

entry A frame stored in a soup and accessed through a cursor. An entry frame contains special slots that identify it as belonging to a soup.

entry alias An object that provides a standard way to save a reference to a soup entry. Entry aliases themselves may be stored in soups.

enumerated dictionary A list of words that can be recognized when this dictionary is enabled. See also **lexical dictionary.**

EOP End of packet indicator.

evaluate slot A slot that's evaluated when NTK (Newton Toolkit) compiles the application.

event An entry in the Dates application for a day, but not a particular time during that day.

field An area in a view where a user can write information.

finder A frame containing methods and/or objects that enumerate data items found to match criteria specified via the Find slip.

flag A value that is set either on or off to enable a feature. Typically, flag values are single bits, though they can be groups of bits or a whole byte.

font spec A structure used to store information about a font, including the font family, style, and point size.

frame An unordered collection of slots, each of which consists of a name and value pair. The value of a slot can be any type of object, and slots can be added or removed from frames dynamically. A frame can have a user-specified class. Frames can be used like records in Pascal and structs in C, and also as objects that respond to messages.

free-form entry field A field of a `protoCharEdit` view that accepts any characters as user input.

function object A frame containing executable code. Function objects are created by the function constructor:

`func(`*args*`)` *funcBody*

An executable function object includes values for its lexical and message environment, as well as code. This information is captured when the function constructor is evaluated at run time.

gesture A handwritten mark that is recognized as having a special meaning in the Newton system, such as tap, scrub, caret, and so on.

global A variable or function that is accessible from any NewtonScript code.

grammar A set of rules defining the format of an entity to be recognized, such as a date, time, phone number, or currency value. Lexical dictionaries are composed of sets of grammars. See also **lexical dictionary.**

home city The **emporium** the system uses to modify dialing information, time zone, and so on. It is usually the user's home, but the user may set it to another city when traveling.

immediate A value that is stored directly rather than through an indirect reference to a heap object. Immediates are characters, integers, or Booleans. See also **reference.**

implementor The frame in which a method is defined. See also **receiver.**

In/Out Box The application that serves as a central repository for all incoming and outgoing data handled by the Routing and Transport interfaces.

inheritance The mechanism by which attributes (slots or data) and behaviors (methods) are made available to objects. Parent inheritance allows views of dissimilar types to share slots containing data or methods. Prototype inheritance allows a template to base its definition on that of another template or prototype.

ink The raw data for input drawn by the user with the stylus. Also known as raw ink or sketch ink.

ink word The grouping of ink data created by the recognition system, based on the timing and spacing of the user's handwriting. Ink words are created when the user has selected "Ink Text" in the Recognition Preferences slip. Ink words can subsequently be recognized with **deferred recognition.**

input spec A frame used in receiving endpoint data that defines how incoming data should be formatted; termination conditions that control when the input should be stopped; data filtering options; and callback methods.

instantiate To make a run-time object in the NewtonScript heap from a template. Usually this term refers to the process of creating a view from a template.

item frame The frame that encapsulates a routed (sent or received) object and that is stored in the In/Out Box soup.

lexical dictionary A list of valid grammars, each specifying the format of an entity to be recognized, such as a date, time, phone number or currency value. See also **enumerated dictionary** and **grammar.**

line A shape defined by two points: the current x and y location of the graphics pen and the x and y location of its destination.

local A variable whose scope is the function within which it is defined. You use the `local` keyword to explicitly create a local variable within a function.

magic pointer A constant that represents a special kind of reference to an object in the Newton ROM. Magic pointer references are resolved at run time by the operating system, which substitutes the actual address of the ROM object for the magic pointer reference.

meeting An entry in the Dates application for a specific time during the day. People can be invited and the meeting can be scheduled for a particular location.

message A symbol with a set of arguments. A message is sent using the message send syntax *frame*`:`*messageName*`()`, where the message *messageName* is sent to the receiver *frame*.

method A function object in a frame slot that is invoked in response to a message.

name reference A frame that contains a soup entry or an alias to a soup entry, often, though not necessarily, from the Names soup. The frame may also contain some of the individual slots from the soup entry.

NewtonScript heap An area of RAM used by the system for dynamically allocated objects, including NewtonScript objects.

`nil` A value that indicates nothing, none, no, or anything negative or empty. It is similar to `(void*)0` in C. The value `nil` represents "false" in Boolean expressions; any other value represents "true."

object A typed piece of data that can be an immediate, array, frame, or binary object. In NewtonScript, only frame objects can hold methods and receive messages.

option frame A frame passed as a parameter to an endpoint method that selects the communication tool to use; controls its configuration and operation; and returns result code information from the endpoint method.

origin The coordinates of the top-left corner of a view, usually (0, 0). The origin can be shifted, for example, to scroll the contents of a view.

output spec A special type of **callback spec** used with an endpoint method. An output spec contains a few additional slots that allow you to pass special protocol flags and to define how the data being sent is translated.

oval A circular or elliptical shape defined by the bounding rectangle that encloses it.

package The unit in which software can be installed on and removed from the Newton. A package consists of a header containing the package name and other information, and one or more **parts** containing the software.

package file A file that contains downloadable Newton software.

package store See **store part.**

parent A frame referenced through the `_parent` slot of another frame. With regard to views, a parent view encloses its child views.

part A unit of software—either code or data—held in a part frame. The format of the part is identified by a four-character identifier called its type or its part code.

part frame The top-level frame that holds an application, book, or auto part.

PCMCIA Personal Computer Memory Card International Association. This acronym is used to describe the memory cards used by the Newton PDA. Newton memory cards follow the PCMCIA standards.

persona The permanent internal description of an individual person that uses a particular Newton PDA, or a particular public image of the Newton owner. The owner is the obvious example, but there can be many others. Choosing a persona sets up information such as name, title, birthday, phone numbers, e-mail addresses, and so on. The plural is "personae."

picker A type of Newton view that pops up and contains a list of items. The user can select an item by tapping it. This type of view closes when the user taps an item or taps outside the list without making a selection.

picture A saved sequence of drawing operations that can be played back later.

polygon A shape defined by a sequence of points representing the polygon's vertices, connected by straight lines from one point to the next.

pop-up See **picker.**

project The collected files and specifications that NTK uses to build a package that can be downloaded and executed on the Newton.

proto A frame referenced through another frame's `_proto` slot. With regard to views, a proto is not intended to be directly instantiated—you reference the proto from a template. The system supplies several view protos, which an application can use to implement user interface elements such as buttons, input fields, and so on.

protocol An agreed-upon set of conventions for communications between two computers, such as the protocol used to communicate between a desktop computer and a Newton device.

raw ink See **ink.**

receiver The frame that was sent a message. The receiver for the invocation of a function object is accessible through the pseudo-variable `self`. See also **implementor.**

recognized text Ink words processed by the recognition system. Ink drawn by the user is converted into recognized text when the user has selected "Text" in the Recognition Preferences slip or after deferred recognition takes place. See also **ink word**.

rectangle A shape defined by two points—its top-left and its bottom-right corners—or by four boundaries—its upper, left, bottom, and right sides.

reference A value that indirectly refers to an array, frame, or binary object. See also **immediate.**

region An arbitrary area or set of areas on the coordinate plane. The outline of a region should be one or more closed loops.

resource Raw data—usually bitmaps or sounds—stored on the development system and incorporated into a Newton application during the project build.

restore To replace all the information in a Newton with information from a file on the desktop.

restricted entry field A field of a `protoCharEdit` view that accepts as user input only the values specified in the view's

template slot. For example, a field for entering phone numbers might restrict acceptable user input to numerals.

rich string A string object that contains imbedded ink words. Rich strings create a compact representation for strings that contain ink words and can be used with most of the string-processing functions provided in the system software. See also **rich string format.**

rich string format The internal representation used for rich strings. Each ink word is represented by a special placeholder character (`kInkChar`) in the string. The data for each ink word is stored after the string terminator character. The final 32 bits in a rich string encode information about the rich string.

root view The topmost parent view in the view hierarchy. All other views descend from the root view.

rounded rectangle A rectangle with rounded corners. The shape is defined by the rectangle itself, along with the diameter of the circles forming the corners (called the diameter of curvature).

routing format A frame that describes how to format an object that is to be sent (routed). Examples include print routing formats, which describe how to visually format data, and frame routing formats, which describe the internal structure of a frame.

routing slip A view that looks like an envelope. The transport displays this view after the user selects a transport-based action from the Action picker. This view is used by a transport to collect information needed to send the item.

script icon An icon that executes a function object when tapped.

self A pseudo-variable that is set to the current receiver.

shape A data structure used by the drawing system to draw an image on the screen.

siblings Child frames that have the same parent frame.

sketch ink See **ink.**

slot An element of a frame or array that can hold an immediate or reference.

soup A persistently stored object that contains a series of frames called entries. Like a database, a soup has indexes you can use to access entries in a sorted order.

soupervisor mechanism The system service that presents the user with information about a soup when the user taps its icon in the Extras Drawer. It allows for filing or moving all soup entries.

soup icon An icon that represents one or more soups, usually in the Storage folder of the Extras Drawer.

stationery Refers to the capability of having different kinds of data within a single application (such as plain notes and outlines in the Notepad) and/or to the capability of having different ways of viewing the same data (such as the Card and All Info views in the Names file). Implementing stationery involves writing data definitions and view definitions. See also **data definition** and **view definition.**

store A physical repository that can contain soups and packages. A store is like a volume on a disk on a personal computer.

store part A part that encapsulates a read-only store. This store may contain one or more soup objects. Store parts permit soup-like access to read-only data residing in a package. Store parts are sometimes referred to as package stores.

target The object being acted upon. Sometimes the target consists of multiple items, for example, when multiple items are selected from an overview for sending.

template A frame that contains the data description of an object (usually a view). A template is intended to be instantiated at run time. See also **proto.**

text run A sequence of characters that are all displayed with the same font specification. Text is represented in paragraph views as a series of text runs with corresponding style (font spec) information. See also **font spec.**

tick A sixtieth of a second.

transport A NewtonScript object that provides a communication service to the Newton In/Out Box. It interfaces between the In/Out Box and an endpoint. Examples include the print, fax, beam, and mail transports. See also **endpoint.**

transport A special type of Newton application used to send and/or receive data. Transports communicate with the In/Out Box on one end and typically to an endpoint object on the other end. Examples include the built-in transports such as print, fax, and beam. See also **endpoint.**

user proto A proto defined by an application developer, not supplied by the system.

view The object instantiated at run time from a template. A view is a frame that represents a visual object on the screen. The `_proto` slot of a view references its template, which defines its characteristics.

view class A primitive building block on which a view is based. All view protos are based directly or indirectly (through another proto) on a view class. The view class of a view is specified in the `viewClass` slot of its template or proto.

view definition A view template that defines how to display data from a particular data definition. A view definition is registered with the system under the name of the data definition to which it applies. The shortened term viewDef is sometimes used. See also **data definition**.

wedge A pie-shaped segment of an oval, bounded by a pair of radii joining at the oval's center. Contrast with **arc**.

Index

A

B

C

D

E

F

G

H

I, J

K

L

M

N

O

P

Q

R

S

T

U

V

W, X, Y, Z

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter Pro 630 printer. Final page negatives were output directly from the text and graphics files. Line art was created using Adobe™ Illustrator. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®.

LEAD WRITER
Christopher Bey

WRITERS
Bob Anders, Christopher Bey, Cheryl Chambers, Gary Hillerson, John Perry, Jonathan Simonoff, Yvonne Tornatta, Dirk van Nouhuys, Adrian Yacub

PROJECT LEADER
Christopher Bey

ILLUSTRATOR
Peggy Kunz

EDITORS
Linda Ackerman, David Schneider, Anne Szabla

PRODUCTION EDITOR
Rex Wolf

PROJECT MANAGER
Gerry Kane

Special thanks to J. Christopher Bell, Gregory Christie, Bob Ebert, Mike Engber, Dan Peterson, Maurice Sharp, and Fred Tou.

NEWTON PROGRAMMER'S REFERENCE CD TEAM
Gary Hillerson, Gerry Kane, Christopher Bey