Newton Programmer's Guide: Communications

First Edition

This first edition is an early release, published to enable Newton platform development. Every effort has been made to ensure the accuracy and completeness of this information, however it is subject to change.

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

If you discover physical defects in the manual or in the media on which a software product is distributed, APDA will replace the media or manual at no charge to you provided you return the item to be replaced with proof of purchase to APDA.

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

11/95

Contents

Figures and Tables

About This Book

This book, *Newton Programmer's Guide: Communications*, describes the Newton communications system software for version 2.0.

Note

This early release is published to enable Newton platform development. Every effort has been made to ensure the accuracy and completeness of this information, however it is subject to change. ◆

Audience

This guide is for anyone who wants to write NewtonScript programs for the Newton family of products, and specifically covers the communication interfaces in Newton system software.

Before using this guide, you should read *Newton Toolkit User's Guide* to learn how to install and use Newton Toolkit, which is the development environment for writing NewtonScript programs for Newton. You may also want to read *The NewtonScript Programming Language* either before or concurrently with this book. That book describes the NewtonScript language, which is used throughout the *Newton Programmer's Guide: Communications*. Additionally, this book is a companion volume to the other volumes in the set, *Newton Programmer's Guide: System Software*. You should refer to those books for details on NewtonScript programming on topics other than communications.

To make best use of this guide, you should already have a good understanding of object-oriented programming concepts and have had experience using a high-level programming language such as C or Pascal. It is helpful, but not necessary, to have some experience programming for a graphic user interface (like the Macintosh desktop or Windows). At the very least, you should already have extensive experience using one or more applications with a graphic user interface.

Related Books

This book is one in a set of books available for Newton programmers. You'll also need to refer to these other books in the set:

- *Newton Programmer's Guide: System Software*. This set of books is the definitive guide and reference for Newton programming topics other than communications.
- *Newton Toolkit User's Guide*. This book introduces the Newton development environment and shows how to develop Newton applications using Newton Toolkit. You should read this book first if you are a new Newton application developer.
- *The NewtonScript Programming Language*. This book describes the NewtonScript programming language.
- *Newton Book Maker User's Guide*. This book describes how to use Newton Book Maker and Newton Toolkit to make Newton digital books and to add online help to Newton applications. You have this book only if you purchased the Newton Toolkit package that includes Book Maker.
- *Newton 2.0 User Interface Guidelines*. This book contains guidelines to help you design Newton applications that optimize the interaction between people and Newton devices.

Sample Code

The Newton Toolkit product includes many sample code projects. You can examine these samples, learn from them, experiment with them, and use them as a starting point for your own applications. These sample code projects illustrate most of the topics covered in this book. They are an invaluable resource for understanding the topics discussed in this book and for making your journey into the world of Newton programming an easier one.

The Newton Developer Technical Support team continually revises the existing samples and creates new sample code. You can find the latest collection of sample code in the Newton developer area on AppleLink. You can gain access to the sample code by participating in the Newton developer support program. For information about how to contact Apple regarding the Newton developer support program, see the section "Developer Products and Support," on page xix.

Conventions Used in This Book

This book uses the following conventions to present various kinds of information.

Special Fonts

This book uses the following special fonts:

- **Boldface**. Key terms and concepts appear in boldface on first use. These terms are also defined in the Glossary.
- `Courier typeface`. Code listings, code snippets, and special identifiers in the text such as predefined system frame names,

slot names, function names, method names, symbols, and constants are shown in the Courier typeface to distinguish them from regular body text. If you are programming, items that appear in Courier should be typed exactly as shown.

- *Italic typeface*. Italic typeface is used in code to indicate replaceable items, such as the names of function parameters, which you must replace with your own names. The names of other books are also shown in italic type, and *rarely*, this style is used for emphasis.

Tap Versus Click

Throughout the Newton software system and in this book, the word "click" sometimes appears as part of the name of a method or variable, as in `ViewClickScript` or `ButtonClickScript`. This may lead you to believe that the text refers to mouse clicks. It does not. Wherever you see the word "click" used this way, it refers to a tap of the pen on the Newton screen (which is somewhat similar to the click of a mouse on a desktop computer).

Frame Code

If you are using the Newton Toolkit (NTK) development environment in conjunction with this book, you may notice that this book displays the code for a frame (such as a view) differently than NTK does.

In NTK, you can see the code for only a single frame slot at a time. In this book, the code for a frame is presented all at once, so you can see all of the slots in the frame, like this:

```
{  viewClass: clView,
   viewBounds: RelBounds( 20, 50, 94, 142 ),
   viewFlags: vNoFlags,
   viewFormat: vfFillWhite+vfFrameBlack+vfPen(1),
   viewJustify: vjCenterH,
```

```
ViewSetupDoneScript: func()
   :UpdateDisplay(),

UpdateDisplay: func()
   SetValue(display, 'text, value);
};
```

If while working in NTK, you want to create a frame that you see in the book, follow these steps:

1. On the NTK template palette, find the view class or proto shown in the book. Draw out a view using that template. If the frame shown in the book contains a `_proto` slot, use the corresponding proto from the NTK template palette. If the frame shown in the book contains a `viewClass` slot instead of a `_proto` slot, use the corresponding view class from the NTK template palette.
2. Edit the `viewBounds` slot to match the values shown in the book.
3. Add each of the other slots you see listed in the frame, setting their values to the values shown in the book. Slots that have values are attribute slots, and those that contain functions are method slots.

Developer Products and Support

APDA is Apple's worldwide source for hundreds of development tools, technical resources, training products, and information for anyone interested in developing applications for Apple computer platforms. Customers receive the *Apple Developer Catalog* featuring all current versions of Apple and the most popular third-party development tools. APDA offers convenient payment and shipping options, including site licensing.

To order product or to request a complimentary copy of the *Apple Developer Catalog*:

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
AppleLink	APDA
America Online	APDAorder
CompuServe	76666,2405
Internet	APDA@applelink.apple.com

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

Additionally, check out the Newton developer world-wide web page at: `http://dev.info.apple.com/newton`

Undocumented System Software Objects

When browsing in the NTK Inspector window, you may see functions, methods, and data objects that are not documented in this book. Undocumented functions, methods, and data objects are not supported, nor are they guaranteed to work in future Newton devices. Using them may produce undesirable effects on current and future Newton devices.

Overview

This chapter provides an overview of the communications facilities in Newton system software 2.0.

The Newton communications architecture is application-oriented, rather than protocol-oriented. This means that you can focus your programming efforts on what your application needs to do, rather than on communication protocol details. A simple high-level NewtonScript interface encapsulates all protocol details, which are handled in the same way regardless of which communication transport tool you are using.

The communication architecture is flexible, supporting complex communication needs. The architecture is also extensible, allowing new communication transport tools to be added dynamically and accessed through the same interface as existing transports. In this way, new communication hardware devices can be supported.

The Newton communications architecture is illustrated in Figure 1-1.

Figure 1-1 Communications architecture

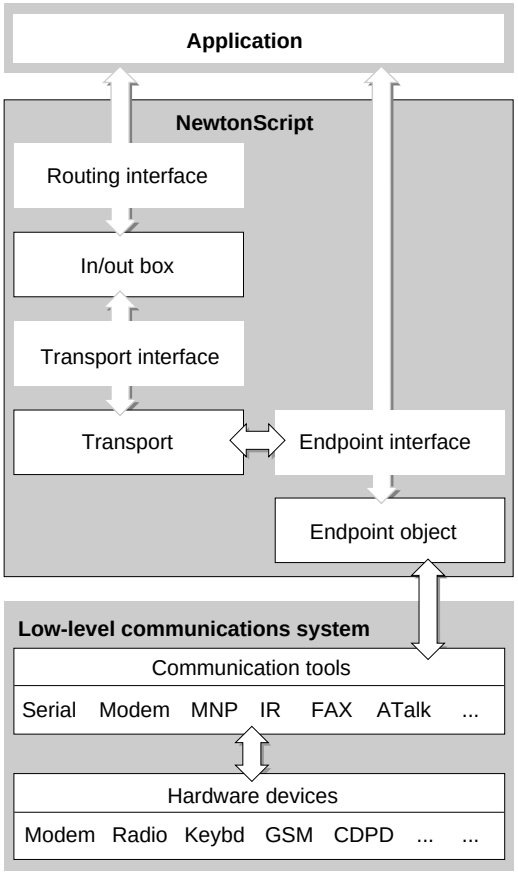


Figure 1-1 shows four unique communications interfaces available for you to use:

- routing interface
- endpoint interface
- transport interface
- communication tool interface

The first two, routing and endpoint interfaces, are available for NewtonScript applications to use directly.

The transport interface is a NewtonScript interface, but it isn't used directly by applications. A transport consists of a special kind of application of its own that is installed on a Newton device and that provides new communication services to the system.

The communication tool interface is a low-level C++ interface.

These interfaces are described in more detail in the following sections.

NewtonScript Application Communications

There are two basic types of NewtonScript communications an application can do. The most common type of communication that most applications do is routing through the In/Out Box. As an alternative, applications can use the endpoint interface to control endpoint objects.

Typically, an application uses only one of these types of communication, but sometimes both are needed.

These two types of communication are described in the following sections.

Routing Through the In/Out Box

The routing interface is the highest-level NewtonScript interface. The routing interface allows an application to communicate with the In/Out Box. The In/Out Box is an application that is visible to the Newton user as icons in the Newton Extras Drawer. The user can tap on either the In Box or the Out Box icon to open the In/Out Box to view and operate on the contents.

The routing interface is best suited for user-controlled messaging and transaction-based communications. For example, the Newton built-in applications use this interface for e-mail, beaming, printing, and faxing. Outgoing items can be stored in the Out Box until a physical connection

is available, when the user can choose to transmit the items, or they can be sent immediately. Incoming items are received in the In Box, where the user can get new mail and beamed items, for example.

For information on the routing interface, refer to Chapter 2, "Routing Interface."

The In/Out Box makes use of the transport and endpoint interfaces internally to perform its operations.

If you are writing an application that takes advantage of only the transports currently installed in the Newton system, you need to use only the routing interface. You need to use the transport or endpoint interfaces only when writing custom communication tools.

Endpoint Interface

The endpoint interface is a somewhat lower-level NewtonScript interface; it has no visible representation to the Newton user. The endpoint interface is suited for real-time communication needs such as database access and terminal emulation. It uses an asynchronous, state-driven communications model.

The endpoint interface is based on a single proto—`protoBasicEndpoint`—that provides a standard interface to all communication tools (serial, fax modem, infrared, AppleTalk, and so on). The endpoint object created from this proto encapsulates and maintains the details of the specific connection. This proto provides methods for

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

The basic endpoint interface is described in Chapter 4, "Endpoint Interface."

Low-Level Communications

There are two lower-level communication interfaces that are not used directly by applications. The transport and communication tool interfaces are typically used together (along with the endpoint interface) to provide a new communication service to the system.

These two interfaces are described in the following sections.

Transport Interface

If you are providing a new communication service through the use of endpoints and lower-level communication tools, you may need to use the transport interface. The transport interface allows your communication service to talk to the In/Out Box and to make itself available to users through the Action button (envelope icon) in most applications.

When the user taps the Action button in an application, the Action picker appears. Built-in transports available on the Action picker include printing, faxing, eWorld e-mailing, and beaming.

For more information, refer to Chapter 3, “Transport Interface.”

Communication Tool Interface

Underlying the NewtonScript interface is the low-level communications system. This system consists of a communications manager module and several code components known as communication tools, representing different kinds of transports. These communication tools interact directly with the communication hardware devices installed in the system. The communication tools are written in C++ and are not directly accessible from NewtonScript—they are accessed indirectly through an endpoint object.

For information about configuring the built-in communication tools through the endpoint interface, refer to Chapter 5, “Built-in Communication Tools.”

Note that the communications manager module, and each of the individual communication tools, runs as a separate operating system task. All NewtonScript code is in a different task, called the Application task.

The system is extensible—additional communication tools can be installed at run time. Installed tools are made available to NewtonScript client applications through the same endpoint interface as the built-in tools.

At some point, Apple Computer, Inc. may release the tools and interfaces that allow C++ communication tool development.

Routing Interface

This chapter describes the Routing interface in Newton system software. The Routing interface allows applications to send, receive, and perform other operations on data such as deleting or duplicating it. It provides a common user interface mechanism that all applications should use to provide routing services.

You should read this chapter if your application needs to provide routing services to the user. This chapter describes how to:

- route items through the Out Box using transport-supplied services
- route items using application-supplied services
- receive incoming items through the In Box
- support viewing items in the In/Out Box

About Routing

Routing is a term used to describe nearly any action taken on a piece of data. Some typical routing actions include printing, faxing, mailing, beaming,

deleting, and duplicating. In addition to system-provided routing services, applications can implement their own routing actions that operate on data.

Routing also describes the process of receiving data through the In Box.

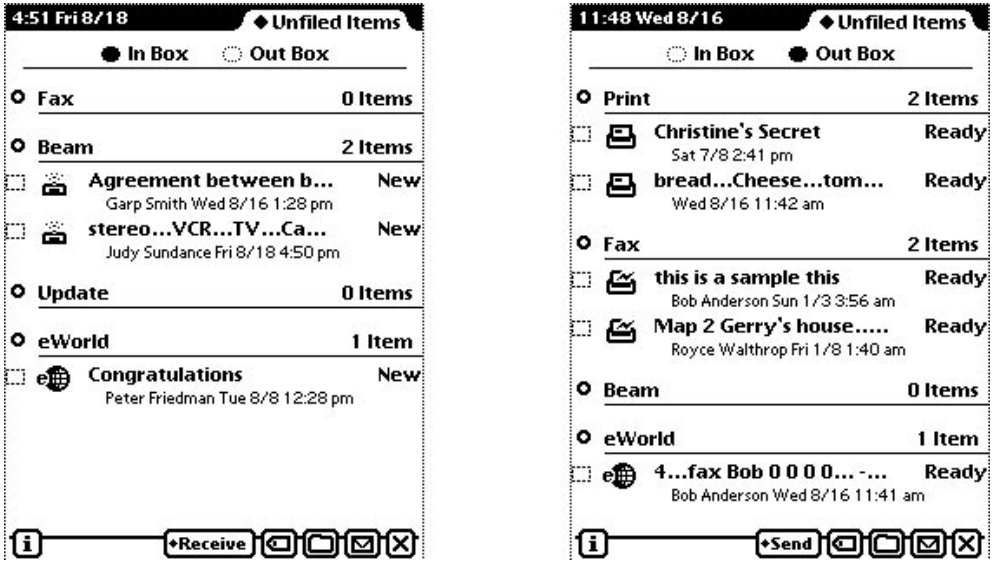
The Routing interface provides the link between an application and the In/Out Box for sending and receiving data using transports. The Routing interface also provides a standard mechanism for an application to make available its own internal routing actions such as deleting and duplicating, that do not use transports.

The In/Out Box

The In/Out Box is used as a central repository for all incoming and outgoing data handled by the Routing and Transport interfaces. The In/Out Box is an application that is accessed by the Newton user through the In Box and Out Box icons in the Extras Drawer. The user can tap either icon to open the In/Out Box to view and operate on its contents. Once it's open, the user can switch between the In Box and the Out Box by tapping a radio button in the application.

When open, the In/Out Box displays either the In Box, containing incoming items, or the Out Box, containing outgoing items. The user can choose to sort the items in both the In Box and the Out Box in various ways, such as by date, type of transport, or status. A transport is a type of communication service such as fax, e-mail, or beam. Figure 2-1 shows the In Box and Out Box overviews where the items are sorted by type of transport.

Figure 2-1 In Box and Out Box overviews



The In/Out Box makes use of the Transport interface internally to perform its operations.

The In Box

Incoming data items are received into the In Box and stored in the In Box soup. For example, the user may receive beamed items, e-mail, or fax messages. Many kinds of In Box items can be viewed in the In Box and they can be put away into one of the other applications residing on the Newton device. For example, the user may receive an e-mail message, read it in the In Box, and then put it away into the Notepad application. The act of putting away an item transfers it to the selected application and deletes it from the In Box soup.

The In Box also supports an automatic "put away" feature. An application can register to automatically receive items designated for it. In this case, as

soon as the In Box receives such an item, it is automatically transferred from the In Box soup to the application, without user intervention. For example, incoming stock quotes from a wireless modem could be automatically transferred to a stock tracking application.

The In Box itself also supports certain routing actions. Certain items in the In Box can be routed directly from there. For example, you can read incoming e-mail, and reply to it, print it, or fax it directly from within the In Box.

The Out Box

Outgoing data items are stored in the Out Box until a physical connection is available or until the user chooses to transmit the items. For example, the user may choose to fax and e-mail several items while aboard an airplane. These items are stored in the Out Box soup. When the user reaches her destination, she connects the Newton to a phone line and sends the items.

While stored in the Out Box, most items can be viewed, some can be edited, and the user can change routing or addressing information, for example, adding more recipients to an e-mail message or changing a fax number.

Individual transports can support automatic connection features. For example, if the Newton contains wireless communication capabilities, the Out Box can automatically transfer an item to a transport as soon as it is sent from an application, without the user having to open the Out Box and tap the Send button.

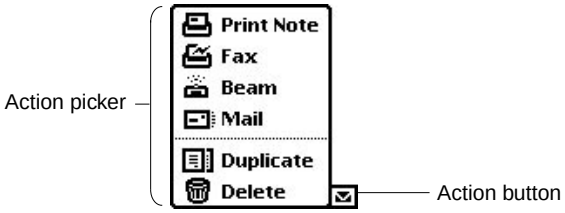
The Out Box itself also supports routing actions. Items in the Out Box can be sent through other transports directly from there. For example, if the user has queued a fax to send, the user can also print it from the Out Box.

Action Picker

Routing actions are accessed in an application from the Action button—the small envelope icon. When the user taps this button, a picker (pop-up menu) listing routing actions is displayed, as shown in Figure 2-2. These routing actions apply to the current target object. The **target** object typically consists of one or more selected items or the data in a view, such as the current note

on the Notepad. Usually this corresponds to a soup entry or to multiple soup entries.

Figure 2-2 Action picker



In the user interface of your application, the Action button should be positioned differently, depending on how your application displays individual items. If your application can have more than one selectable data view on the screen at a time (like the Notepad) then the Action button should be attached to the top of the view it will act on. For example, each note in the Notepad has its own Action button, which applies just to that note. If your application displays only one data view at a time, then the Action button should be on the button bar at the bottom of the screen.

You can add the Action button to the user interface of your application by adding a view based on the `protoActionButton` proto (page 2-51). This proto contains the functionality to create and pop up the picker.

The picker displayed when a user taps on the Action button lists the routing actions available for the particular type of data that is selected as the target. There are two kinds of routing actions that can appear on the Action picker:

- routing actions corresponding to transports installed in the Newton device
- application-defined actions such as delete and duplicate, that do not use the Out Box and a transport to perform the routing operation

Transport-based actions that support the type of data being routed are shown at the top of the Action picker. Application-defined routing actions appear at the bottom of the picker, below a separator line.

Note that the first action listed in the Action picker has the name of the target item appended to it (for example, “Print Note”). The system obtains the name of the item from the `appObject` slot. Most applications define this slot in their base view. It holds an array of two strings, the singular and plural forms of the name of the item (for example, `["Entry", "Entries"]`.

The system builds the list of routing actions dynamically, at the time the Action button is tapped. This allows all applications to take advantage of new transports that are added to the system at any time. Applications and transports need know nothing about each other; the Routing interface acts as the bridge, creating the picker at run time.

Applications don’t specify that particular transports be included on the picker. Instead, applications enable transports based on the type of data to route and the formats available for it: view data (for example, for printing and faxing), frame data (for example, for beaming), text data (for example, for e-mail), and so on.

Here’s a summary of the main steps that occur when the user taps the Action button:

1. The system obtains the target by sending the `GetTargetInfo` message to the Action button view, and then determines the class of the target.
2. Using the target class, the system builds a list of routing formats that can handle that class of data by looking them up in the view definition registry using the `GetViewDefs` function.
3. Using the list of formats, the system builds a list of transports that can handle at least one of the data types supported by any of the formats. The matching transports are shown on the Action picker. Application-defined actions such as delete or duplicate are also added to the picker.
4. If the user chooses a transport-based action from the picker, the system sends the `SetupItem` message to the current (last-used) format for that transport and the data type being routed. Then the routing slip is opened, where the user can supply addressing information and confirm the routing action. If the user switches formats from among those available, the `SetupItem` message is sent to the new format.

5. If the user chooses an application-defined action from the picker, the system sends the Action button view the message defined by the application for that action (in the `RouteScript` slot of the action frame).

The following section describes routing formats in more detail and explains how they're used to determine what transport-based routing actions appear on the Action picker. The steps in this summary are explained in much greater detail in the section "Providing Transport-Based Routing Actions" beginning on page 2-12.

Routing Formats

To implement the sending of data using the Routing interface and a transport, an application uses one or more routing formats that specify how data is to be formatted when it is routed. A routing format is a frame specifying items such as the title of the format, a unique identifying symbol, the type of data the format handles, and other information controlling how the data is handled. Some types of routing formats, such as print formats, are view templates that contain child views that lay out the data being printed. Other types of routing formats, such as frame formats, simply control how a frame of data is sent and have no visual representation.

Here is an example of a routing format frame:

```
{_proto: protoPrintFormat,// based on this proto
dataTypes: ['view], // unneeded, supplied by proto
title: "Two-column", // name of format
symbol: '|twoColumnFormat:SIG|, // format id
// construct child views that do the actual layout
ViewSetupChildrenScript: func() begin ... end,
// handle multiple pages
PrintNextPageScript: func() begin ... end,
...}
```

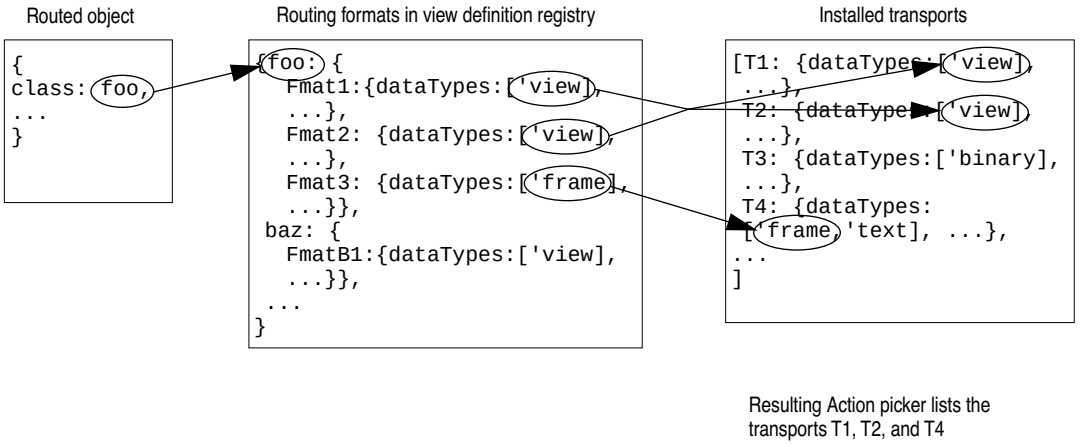
The `dataTypes` slot in the format frame indicates the types of data handled by the format. This slot and the class of the data object being routed are used to determine which transports show up in the Action picker. The system

builds a list of all formats registered under the symbol matching the class of the object being routed. This list contains all the formats that can be used with that class of object. Remember that the class of a frame object is simply the value of the `class` slot in the frame. So, to route a frame object, it must have a `class` slot that contains a value corresponding to one of the classes under which routing formats are registered. For more details about registering routing formats, see the section "Registering Routing Formats" beginning on page 2-20.

Each transport installed in the system contains a `dataTypes` array that indicates what data types it can handle. For the item being routed, the Action picker lists every transport whose `dataTypes` slot includes one of the types specified by the `dataTypes` slots of the routing formats associated with that item. This selection mechanism is illustrated in Figure 2-3.

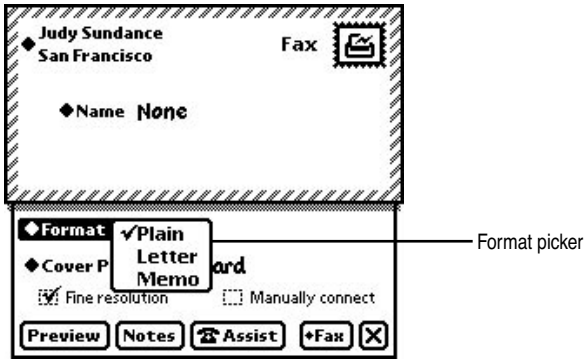
For more information about transports, see Chapter 3, "Transport Interface."

Figure 2-3 Transport selection mechanism for action picker



Once the user chooses a transport from the Action picker, the routing slip for that transport is displayed. All the routing formats that support the class of data being routed and that are handled by that transport are listed in the format picker in the routing slip, as shown in Figure 2-4. The last used format is set as the current format, or if no last format is found, the first format found is set as the current format.

Figure 2-4 Format picker in routing slip



The built-in applications and transports support routing of the basic data types listed in Table 2-1. Other data types may be defined by applications, but only those transports aware of them can use them. If you do create a custom data type, be sure to append your developer signature to make it unique.

Table 2-1 Routing data types

Data type	Description	Built-in transport support[1]
`'view`	Data is exported in page-by-page views for operations such as printing and faxing	print, fax
`'frame`	Data is exported as a NewtonScript frame	eWorld, beam
`'text`	Data is exported as a string	eWorld
`'binary`	Data is exported as a binary object	not applicable

[1] This column lists the built-in transports that support each of the routing data types. Note that this information is firmware-dependent. All Newton devices may not have all of these transports built in, and some devices may have additional transports not listed here.

Typically, an application defines multiple routing formats, to allow routing using different types of transports. For example, an application might define one `'frame` format, one `'text` format, and two `'view` formats.

An application may make use of built-in routing formats and other routing formats that have been registered in the system, if the application sends data of the class handled by those formats. But typically an application registers unique formats of its own that are custom designed for its own data.

You must register with the system all routing formats that you define, usually in your application part `InstallScript` function. Registration is discussed in the section “Registering Routing Formats” beginning on page 2-20.

Current Format

The routing system maintains a "current format," which is the last routing format used by your application for a specific transport, or the first routing format available otherwise. The current format is used to set the format picker in the routing slip the next time the user chooses to route an item using the same transport.

The system maintains the current format by adding a frame called `lastFormats` to your application base view. The `lastFormats` frame contains a slot for each transport that has been used from your application. Whenever a transport is selected, the system first checks your application for the `lastFormats` frame and for a slot within that frame named with the symbol of the transport being used. If the slot is found, it contains the symbol of the last format used by that transport. Then the system searches for a format whose `symbol` slot matches it.

If your application does not have a `lastFormats` slot, or if a matching format is not found (the format went away), the first format found becomes the current format.

Whenever the user changes the current format, it is saved in the `lastFormats` frame in your application's base view. It is your responsibility to save the `lastFormats` frame to a soup if you want to maintain it.

Routing Compatibility

The Routing interface described in this chapter is entirely new in system software 2.0. The previous Routing interface, documented in *Newton Programmer's Guide*, is obsolete, but still supported for compatibility with older applications. Do not use the old Routing interface, as it will not be supported in future system software versions.

Note that if a 1.x application that includes the routing capability is run under system software version 2.0, the names of routing actions in the Action picker may appear slightly differently than they do under version 1.x because of the way that the picker is constructed in version 2.0.

Print Formats

In the 1.x system, print formats have left and right default margins of 60 pixels. In Newton 2.0, the default margins are 0. If you are converting a print format for use in Newton 2.0, you shouldn't need to do anything special if you let the print format set up its own view bounds and the child views are positioned relative to the parent's bounds.

Using Routing

This section describes how to use the Routing interface to perform these specific tasks:

- provide transport-based routing actions
- provide application-specific routing actions
- send items programmatically
- receive items
- allow items to be viewed in the In/Out Box

Providing Transport-Based Routing Actions

Here's a summary of the minimum actions you need to do to support routing by the Action button in an application:

- Include the Action button in your application (or in individual views) by adding a view based on the `protoActionButton` proto (described on page 2-51).
- Supply a `GetTargetInfo` method in your application (or in individual views) or ensure that the `target` and `targetView` slots are set up correctly with the target object and target view, so the system can determine what is being routed.

- Ensure that the target data object has a meaningful class (for frame objects, this is the `class` slot). The data class is used to determine the appropriate formats, and thus transports, available to an item.
- Create one or more routing formats. Give your formats unique `symbol` and `title` slots, and supply the `SetupItem` method, if necessary (but call the inherited method also). View formats may need a `PrintNextPageScript` method if multiple pages could be involved, and may need a `FormatInitScript` method if much preparation must be done before printing or faxing. Text formats may need a `TextScript` method.
- Register your routing formats in the application `InstallScript` function and unregister them in the `RemoveScript` function.

To support routing through transports, your application uses one or more routing formats. These may be custom formats registered by your application or other formats built into the system or installed separately. For more information about routing formats, see the section "Routing Formats" beginning on page 2-7. There are some prototype formats built into the system that you can use to create your own formats:

- To create a format for routing a `'view` data type, you can use the `protoPrintFormat`, described on page 2-21.
- To create a format for routing a `'frame` or `'text` data type, you can use the `protoFrameFormat`, described on page 2-26.
- To create a new kind of format for data types other than `'view` or `'frame`, you can use the `protoRoutingFormat`, described on page 2-27.

The following sections describe the more detailed aspects of supporting transport-based routing.

Getting and Verifying the Target Object

When the user first taps on the Action button, but before a choice is made from the picker, the Routing interface sends the Action button view the `GetTargetInfo` message (page 2-70), passing the symbol `'routing` as a parameter. The purpose of this message is to get the target object to be routed and the target view in which it resides. Usually, these items are stored in slots

named `target` and `targetView` in your application. If you set up and use such slots in your views, you don't need to implement the `GetTargetInfo` method because this is a root view method that will be found by inheritance. The root view method simply looks for the slots `target` and `targetView`, starting from the receiver of the message, which is the Action button view. It returns these slots in a frame called the target information frame. If you don't use these slots in your views, you'll need to implement the `GetTargetInfo` method to return them.

You'll need to implement the `GetTargetInfo` method if the user has selected multiple items to route. In this case, you'll need to construct a single object that encapsulates the multiple items selected for routing, because the target must be a single object. In your `GetTargetInfo` method you can use the function `CreateTargetCursor` (page 2-67) to create a multiple item target object from the selected items.

Note

In most cases the target object is a frame. In some cases you might want to route a non-frame object such as a string or binary. The Routing interface supports non-frame target objects, however other system services such as Filing may require target objects that are frames, so you may not be able to use the same target with them. Note also that non-frame target objects must have a meaningful class. ◆

Once the user actually chooses a transport-based routing action from the Action picker, the system creates a new Out Box item frame containing some default slots and values for the target item. This is done by means of the transport method `NewItem`. One slot that is initialized by `NewItem` is the `appSymbol` slot of the item frame. The value for this slot is obtained from the `appSymbol` slot of the application doing the routing (through inheritance from the Action button view).

Then, just before the routing slip is opened, the Routing interface sends the message `VerifyRoutingInfo` (page 2-77) to the view identified by the `appSymbol` slot in the item frame. This is normally your application base view. However, if you are doing routing from a view created by `BuildContext`, for example, the `appSymbol` slot might be missing because

such views don't automatically include this slot. You must include an `appSymbol` slot in such a view, since it determines where the `VerifyRoutingInfo` message is sent.

The `VerifyRoutingInfo` method is passed two parameters, the target information frame obtained by `GetTargetInfo`, and the partially initialized item frame obtained from `NewItem`. The `VerifyRoutingInfo` method allows you a chance to verify or change the target item before the routing slip is opened. Normally you would return the same target frame that was passed in, possibly modified. To cancel the routing operation, you can return `nil`. If you don't care to implement this method, you don't need to.

If multiple items are being routed, the target object (constructed by `CreateTargeCursor`) encapsulates them all. In your `VerifyRoutingInfo` method, you can use the function `GetTargetCursor` (page 2-67) to return a cursor to navigate the items. Then you can iterate through the cursor using the cursor methods `Entry`, `Next`, and `Prev`, as described in Chapter 11, "Data Storage and Retrieval," in *Newton Programmer's Guide: System Software*.

Note

When an item is routed from within the In/Out Box, the In/Out Box implements its own `VerifyRoutingInfo` method, which adds one slot to the target information frame that it returns. This slot is named `targetEntry`, and it contains the actual In/Out Box entry being routed. This is provided in case you need access to the In/Out Box soup entry, since `target` contains only the `body` slot of the soup entry. ◆

Getting and Setting the Current Format

Next, the Routing interface sends your application base view the `GetDefaultFormat` message (page 2-64). The purpose of this message is to get the default format so that when the routing slip is opened, the format can be initially set to the default. Normally, the default format for a particular transport is simply the last format used with that transport from that

application. This information is stored in the `lastFormats` slots of your application base view. Unless you want to do something special, you don't need to implement the `GetDefaultFormat` method because this is a root view method that will be found by inheritance. The root view method simply gets the default format from the `lastFormats` slot.

The format can be changed by the user, or by the system (if no last format is found, the default is set to the first one that is found). When the format is changed, the Routing interface sends your application base view the `SetDefaultFormat` message (page 2-65). The purpose of this message is to store the default format for later use. Normally, this is stored in the `lastFormats` slot in the application base view. Unless you want to do something special, you don't need to implement the `SetDefaultFormat` method because this is a root view method that will be found by inheritance. The root view method simply sets the new format in the `lastFormats` slot of your application base view.

Supplying the Target Object

Next, the Routing interface sends the `SetupItem` message (page 2-56) to your format, assuming it is the one set as the current format. This message informs you that the format is selected and an item is being routed. The `SetupItem` method is passed two parameters: a partially initialized item frame, and a target information frame, as returned by `GetTargetInfo`. The item frame is obtained from the transport method `NewItem`, which creates a new Out Box item frame containing some default slots and values. This is the frame that is to be stored in the Out Box soup. It must be filled in with the data object being sent.

The target information frame contains two important slots, `target` and `targetView`, which define the data object to be routed and the view that contains it, respectively. The `SetupItem` method must set the `body` slot of the item frame to the value contained in the `target` slot of the target information frame. This fills in the item frame with the actual data to be sent.

You are not required to provide a `SetupItem` method in your routing format since this method is defined in the routing format protos. The `SetupItem` method defined in the proto simply assigns the `target` slot in

the target information frame to the `body` slot of the item frame. You can override this method if you want to perform additional operations and then call the inherited `SetupItem` method. Note that there's a potential problem with not copying the target object. If the object is viewable and editable and the user edits the object in the Out Box, that also changes the original object stored by the application, since there's actually not two separate objects—just two pointers to the same object.

If you want to modify the `body` slot of the item in some way, you should supply your own `SetupItem` method instead of calling the inherited version. Then in your own `SetupItem` method, clone the `target` slot of the target information frame into the `body` slot of the item frame.

When sending data to another Newton device (for example, by beaming) it's a good idea to ensure that the sent object contains a `version` slot that holds the current version of your application. This will help to avoid compatibility problems in future versions of your application. If the data format changes, your application can easily identify older and newer data formats.

Storing an Alias to the Target Object

When there is a single target object, if there is not enough storage space, or the target object is larger than a specified size, you can specify that an alias to the target object, rather than the target object itself, be stored in the `item.body` slot. You enable the storing of an alias by setting the `storeAlias` slot in the routing format frame to `true`. Additionally, you can specify a maximum size limit for target objects by setting the `sizeLimit` slot in the routing format frame. If any target object is larger than the size specified in this slot, and `storeAlias` is also `true`, an alias to the target object is stored in the `item.body` slot.

The default `SetupItem` method provided in the routing format protos reads the `storeAlias` slot and performs the appropriate operations if this slot is `true`; otherwise, it assigns the actual target object to the `item.body` slot, as usual. If an alias to the target object is stored in the `item.body` slot, the routing interface also sets the `item.needsResolve` slot to `true`, to signal that the `body` slot contains an alias that will need to be resolved.

If an alias to an item is stored, the item can still be viewed and operated upon in the In/Out Box, just like any other item.

Note that there's some potential problems if an alias to the target object is stored. If the target entry resides on a card store, and the card is removed before the item is actually sent from the Out Box, the alias cannot be resolved and the send operation will fail. No matter where the original object resides, even if it is simply deleted, the send operation will fail. Therefore, whenever an alias is stored, the user is warned by an alert slip explaining that the original item must be available when the routed item is actually sent. You can set the slot `showMessage` to `nil` in the format to prevent the warning message from being displayed.

Another problem with storing an alias is that the alias is just a pointer to the original data. For example, say the user faxes a note and chooses to send it later, and you store an alias to the note in the Out Box. Then the user opens the fax item in the Out Box and changes the note. This actually changes the original note in the Notepad application, since the alias is just a pointer to the original data. Similarly, if the user changed the original note before the fax was sent, then the fax text would be changed without the user being aware of it.

Most target objects are soup entries, for which the routing format protos can handle the operations of determining the object size, making an alias, and resolving the alias when needed. However, in some cases, you may want to route objects that are not soup entries. If you want to create aliases to such objects, you must override the routing format methods that handle the alias operations: `TargetSize` (page 2-58), `MakeBodyAlias` (page 2-58), and `ResolveBody` (page 2-59).

The `TargetSize` method must determine the size of the target object passed to it. The default method does this for soup entries, but you must override it and do it yourself for other kinds of objects. The size of the object is used to determine if the object is greater than the size specified by the sizeLimit slot in the routing format, or greater than the available space on the store. If either of these conditions is true, then an alias is created for the object.

Handling a Multi-Item Selection

If the target data consists of multiple items selected from an overview, you can specify that these items be stored individually in the Out Box or that a single alias cursor be stored in the Out Box. The `storeCursors` slot in the routing format frame controls this feature, along with the transport. This feature works only if the transport also supports it and is able to handle a cursor (the transport `allowBodyCursors` slot is also `true`). The built-in beam and eWorld transports do not support the storing of cursors for multiple items, so the `storeCursors` slot in the routing format will be ignored for those transports.

The default value of the `storeCursors` slot is `true`.

Set this slot to `true` to store a single alias cursor to the items in the Out Box. When the items are sent, the cursor is resolved into its component entries. Note that there's a potential problem with storing an alias cursor to the items. If the target entries reside on a card store, and the card is removed before one of the items is actually sent from the Out Box, the alias cannot be resolved and the send operation will fail.

Set the `storeCursors` slot to `nil` to store each of the selected items as a separate item in the Out Box. Each of the items can later be sent or operated on individually from the Out Box.

Displaying an Auxiliary View

When the user chooses a format in the format picker, you may need to get additional information from the user in the routing slip view. You can do this by means of an auxiliary view template that you specify in the `auxForm` slot of the routing format. If you specify a view template in this slot, when the format is selected, this auxiliary view template is instantiated with the function `BuildContext` and is sent an `Open` message to display itself.

If you need access to information about the item being routed, you can access the `fields` slot in the auxiliary view. The system sets this slot to the frame that will become the In/Out Box entry for the item being routed. For details on this frame, see the section "Item Frame" beginning on page 2-44. It is recommended that you do not change any slots in the `fields` frame.

Registering Routing Formats

All routing formats are specified as view definitions and are registered with the system by means of the global function `RegisterViewDef`, described on page 2-72. The formats that handle data types other than `'view` are not actually views, but they are registered as view definitions to take advantage of the central registration mechanism. Registering formats in this way makes them available to all applications in the system. Routing formats are specially identified in the view definition registry because the `type` slot of all routing formats is set to the symbol `'printFormat` (even non-view formats).

You register formats with the class of the object you want them to act on. Here is an example of registering a format for `'view` data types:

```
RegisterViewDef(myPrintFormat, '|myDataClass:SIG|);
```

This call registers the format `myPrintFormat` as working with data whose class is `'|myDataClass:SIG|`. If the class of any target data object is `'|myDataClass:SIG|`, then the format `myPrintFormat` will be available when that item is routed. The fact that this print format has been registered means that you will be able to print and fax that class of data items. This mechanism enables you to have separate routing formats (and thus routing actions) for individual views, rather than using the same formats (and routing actions) for all views in an application.

Typically, your application registers routing formats when it is installed, in its part's `InstallScript` function, and unregisters formats in its `RemoveScript` function. You use the function `UnRegisterViewDef`, described on page 2-74, to unregister routing formats.

In the application part `InstallScript` function, when you register your routing formats, you must not use the Newton Toolkit function `GetLayout` to obtain a reference to the routing format layout so that you can pass it to `RegisterViewDef`. Nor should you use `DefConst`, or any other method that directly references the routing format. This is because the entire `InstallScript` function is passed to `EnsureInternal` (for application parts). Your routing format layouts would all be copied into the NewtonScript heap, wasting precious memory.

Instead, you should use an indirect method to reference your routing format layouts. One way is to store a reference to your routing format layouts (by using `GetLayout`) in a slot in your application base view. Then in the `InstallScript` function, you can reference that slot through the expression `partFrame.theForm`. Because the reference to the layout is found at run time through an argument to `InstallScript`, it will not be copied into NewtonScript memory by `EnsureInternal` when your application is installed.

For example, first you could store the routing format layout in an application base view slot:

```
anAppSlot: {myLayout: GetLayout("ViewAndText")};
```

Then in the `InstallScript` function, you could use code like this to register the format:

```
InstallScript(partFrame)
begin
   local myApp := partFrame.theForm;
   ...
   RegisterViewDef(myApp.anAppSlot.myLayout,
                   kMyMainDataDefSym);
end;
```

For more information about view definitions and the functions that act on them, refer to Chapter 5, “Stationery,” in *Newton Programmer’s Guide: System Software*.

Creating a Print Format

You create a print format by using `protoPrintFormat`. This proto is required for routing formats with a `'view` data type, such as views that you would print or fax. This proto format is actually a view template, which displays the target object visually. The data to be displayed is laid out as child views of the `protoPrintFormat` view.

Here is an example of a format based on this proto:

```
// in NTK you create a new layout for view formats
MyPrintFormat := {
   _proto: protoPrintFormat,
   symbol: '|myPrintFormat:SIG|,
   title: "PrintIt",
   ViewSetupChildrenScript: func() begin
      // construct child views for first page here
      end,
   PrintNextPageScript: func() begin
      nil;
      // construct child views for next page here
      end,
};
```

For more information about the slots and methods provided by this proto, see the section "Routing Format Protos" beginning on page 2-53.

Topics unique to `protoPrintFormat` are discussed in the following subsections.

Page Layout

The view based on the `protoPrintFormat` proto is automatically sized (in the `ViewSetupFormScript` method) to fit the dimensions of a page on the output device to which it is being routed. You can control the margins used when the data is laid out on the page by setting the `margins` slot. Set this slot to a bounds rectangle frame, like this:

```
{left: 25, top: 20, right: 25, bottom: 30}
```

Each of the slots in this frame is interpreted as an inset from the edge of the printable area of the paper in pixels. You should specify only positive values, to make sure that you don't try to print off the page. The default value of the `margins` slot is `{left:0, top:0, right:0, bottom:0}`.

Also, you can control the orientation in which the data is placed on the paper by setting the `orientation` slot. Specify a symbol indicating if the paper should be used vertically in portrait mode (`'portrait`) or horizontally in landscape mode (`'landscape`). The default value of the `orientation` slot is `'portrait`. Your format should always use relative view justification and/or check the actual bounds of the print format by using the `LocalBox` view method. Then it should adjust the orientation accordingly, or you may want to provide an auxiliary view in which the user can choose the orientation. For information about using an auxiliary view, see the section "Displaying an Auxiliary View" on page 2-19. Note that you cannot change the orientation between a series of pages being printed by a single print format.

If multiple items are being routed (as from a multiple selection in an overview), you may want to print each item on a separate page or you may want to print the items one after another, placing multiple items on the same page before starting a new page. You can control this feature by setting the `usesCursors` slot. The default setting of this slot is `nil`.

If you want to lay out multiple items on a page, set the `usesCursors` slot of the format to `true`. In this case, the target object encapsulates all of the items being routed. Your format should call the `GetTargetCursor` method (page 2-67) to return a cursor, on which you can iterate over the individual items to be routed using the standard cursor methods `Entry`, `Next`, and `Prev`. You can use the `GetCursorFormat` method (see page 2-60) of the `protoPrintFormat` to find formats for the individual items.

If you want to lay out each item on a separate page, or if this format cannot handle a multiple item target object, set the `usesCursors` slot to `nil`. In this case, this format is invoked multiple times, once for each item being routed, and each item is put on a separate page.

Printing and Faxing

When an item in the In Box is actually printed or faxed using your print format, the view represented by the print format is instantiated and drawn to the output device. As when any view is instantiated, the system sends the print format view standard messages and also routing-specific messages. For

optimal printing performance, and to avoid timing out a fax connection, you need to be aware of the sequence of events and know which operations are time-critical.

Here is the sequence of events during a printing or faxing operation:

1. The system sends the print format the `FormatInitScript` message (page 2-60), to give you an opportunity to perform initialization operations. You must perform any lengthy initialization operations in this method, before the transport connection is made. You can store initialized data in the format frame (`self`).
2. For sending a fax only, the system sends the print format the `CountPages` message (page 2-61). If you can determine the number of pages in the fax ahead of time, you should override this method in your print format and have it return the number of pages (not including the cover page). If you don't override this message, the system opens the print format view in an offscreen window and performs steps 3, 4, and 6, below, to go through each page so it can count the number of pages. Then the print format view is closed. Note that the `ViewShowScript` and `ViewDrawScript` messages are not sent to the view. This is a lot of work for the system to do just to determine the number of pages, so if you can, it's a good idea to override the `CountPages` method with one of your own.
3. The system instantiates the print format view and sends it the `ViewSetupFormScript` message. Depending on certain factors, the transport connection might be made at the beginning of this step or in step 4. You can rely only on the connection being made sometime after step 2.
4. The system sends the `ViewSetupChildrenScript` message to the print format view, then the child views are instantiated (and they are sent the standard view messages), and then the system sends the `ViewSetupDoneScript` and `ViewShowScript` messages to the view.
5. The system draws the print format view and sends the `ViewDrawScript` message to the view. Note that each of the child views on the page is also drawn and sent the `ViewDrawScript` message, in hierarchical order. The page might be printed or faxed in "bands," so this step might be repeated several times for the page.

If you need to draw something in your `ViewDrawScript` method, you can call the view method `GetDrawBox` to determine the band that is currently being drawn. Then you can draw just those shapes that are necessary for the current band. The system will not draw any views or shapes outside the current band. Any shapes extending outside the current band are automatically clipped.

IMPORTANT

The `ViewDrawScript` message is sent at a time-critical point in a fax operation. It is imperative that you do as little work as possible in the `ViewDrawScript` method. ▲

6. The system sends the `PrintNextPageScript` message to the print format view (see page 2-59). If your print format handles more than a single page of data, you must define the `PrintNextPageScript` method in your print format. The system sends this message each time it reaches the end of a page, to allow you to construct the next page of data. While there is more data to route, this method should return a non-`nil` value; in that case, the printing process continues with the next page at step 4. When there is no more data to route, the `PrintNextPageScript` method should return `nil`; in that case the printing process ends and the connection is closed.

You set up the child views containing the data for the first page in the `ViewSetupChildrenScript` method of your print format. Typically you do this by setting the value of the `stepChildren` array. Don't forget to call the inherited method (`inherited:?ViewSetupChildrenScript`) so that the proto behavior is preserved.

The `PrintNextPageScript` method should construct the view for the next page of data so that the message `self:Dirty()` will show the view. Typically, you do this by keeping track of what data has been routed so far, and when the format receives this message, you select a new set of child views representing the next page of data to send. Then you call the view method `RedoChildren`, which closes and then reopens the child views. This method also causes the system to send your print format view the `ViewSetupChildrenScript` message again.

When faxing, it's best not to perform lengthy operations in the `PrintNextPageScript` method, since the connection stays open between pages. However, this is less time-critical than the `ViewDrawScript` method. If possible, execute lengthy operations in the `ViewSetupFormScript` method, which is called just once before the connection is opened.

If you need to create any custom shapes which are to be drawn on the page by the `ViewDrawScript` method, create the shapes in the `FormatInitScript` method. Alternatively, you can create shapes at compile time, if they are static. Because of fax connection time-out issues, minimize shape creation in the `ViewDrawScript` method, as shape creation takes too much time and it's possible that the connection might time out as a result.

Creating a Frame Format

You create a frame format by using `protoFrameFormat`. This is the standard format for routing objects with `'frame` or `'text` data types, such as for beaming and e-mail. To enable these types of transports for your data, you must register at least one format based on this proto. Here is an example of a format based on this proto:

```
MyFrameFormat := {
   _proto: protoFrameFormat,
   symbol: '|myFrameFormat:SIG|,
   title: "No comments",
   SetupItem: func(item, targetInfoFrame) begin
         // call inherited method
         inherited:SetupItem(item, targetInfoFrame);
         // remove comments from target
         RemoveSlot(item.body,'comments);
         end,
   TextScript: func(item,target) begin . . . end,
   ...
};
```

Note that one application can have multiple frame formats. You would simply supply a different `SetupItem` method for the different formats (as well as unique `symbol` and `title` slots), to construct the item frame differently.

If your frame format doesn't support the `'text` data type, you should override the `dataTypes` slot and set it to only `['frame]`.

For more information about the slots and methods provided by this proto, see the section "Routing Format Protos" beginning on page 2-53.

Creating a New Type of Format

You create a new type of routing format by using `protoRoutingFormat`. This is the base routing format, which serves as a proto for the other routing format protos. You would normally use this proto only if you want to create a new type of routing format.

Here is an example of a format based on this proto:

```
MyNewFormat := {
   _proto: protoRoutingformat,
   dataTypes: ['binary],
   symbol: '|myFormat:SIG|,
   title: "Custom",
   SetupItem: func(item, targetInfoFrame) begin ... end,
   ...
};
```

For more information about the slots and methods provided by this proto, see the section "Routing Format Protos" beginning on page 2-53.

Providing Application-Specific Routing Actions

First, to provide the Action button in the user interface of your application, you must include a view based on the `protoActionButton` proto, described on page 2-51.

Your application can provide internal application-defined actions, such as deleting and duplicating, that do not use the Out Box and a transport to perform the routing operation. These routing actions appear at the bottom of the Action picker.

You define these routing actions by providing a slot named `routeScripts` in your application. The Action button searches its own context for the first `routeScripts` slot that it finds. Usually you will define `routeScripts` in the base view of your application. That way, all child views can find it by inheritance. But if you want to have different routing actions active for different views, you can define a `routeScripts` slot in each child view definition, where it will override the one in the base view.

Alternatively, instead of defining an array of application-specific routing actions in the `routeScripts` slot, you may want to build the array dynamically. To do this, you can override the root view method `GetRouteScripts`, which is used by the Routing interface to obtain the `routeScripts` array from your application. The default version of this method simply returns the contents of the `routeScripts` slot to the Routing interface. In the `GetRouteScripts` method, simply build and return an array just like you would define in the `routeScripts` slot. The `GetRouteScripts` method is described on page 2-70.

If you provide a `routeScripts` slot, it must contain an array of frames, one for each routing action item, that look like this:

```
{title: "MyAction", // name of action
icon: GetPictAsBits("MyActionIcon",nil), // picker icon
RouteScript: 'MyActionFunc, // called if action selected
// other slots and methods you need
...}
```

To include a separator line in the Action picker's list of application-specific routing actions, include the symbol `'pickSeparator` in the `routeScripts` array between the two items you want to separate. Alternatively, you can include a `nil` value to include a separator line.

For delete and duplicate actions, there are bitmaps available in ROM that you can use as icons. For the `icon` slot of a delete action, you can specify the magic pointer constant `ROM_RouteDeleteIcon`. For the `icon` slot of a duplicate action, you can specify the magic pointer constant `ROM_RouteDuplicateIcon`.

If your application registers view definitions with the system, note that each view definition can define its own `routeScripts` array. The routing action items that apply to the individual view definition are added below those that apply to the whole application in the Action picker. See the following section for more information about specifying `routeScripts` in stationery.

There is also support for a system global `routeScripts` array. Any routing action items defined in this global array will show up in all application Action pickers, above any application-specific or view-specific items.

Performing the Routing Action

The key slot in each of the frames in the `routeScripts` array is the `RouteScript` slot. This slot contains a symbol identifying a method (defined by your application) that is called if the user chooses this action from the Action picker. This method is where you perform the routing action. The method you define is passed two parameters, *target* and *targetView*, which define the data object to be routed and the view that contains it, respectively.

The two values, *target* and *targetView*, are obtained from your application by the Routing interface. As soon as the Action button is first tapped, the Routing interface sends the Action button view the `GetTargetInfo` message to obtain these two values. The `GetTargetInfo` method, described on page 2-70, returns a frame containing these and other slots.

If you set up and use `target` and `targetView` slots in your views, you don't need to implement the `GetTargetInfo` method because this is a root view method that will be found by inheritance. The root view method simply looks for the slots `target` and `targetView`, starting from the receiver of the message, which is the Action button view. It returns these slots in a frame called the target information frame. If you don't use these

slots in your views, you'll need to implement the `GetTargetInfo` method to return them.

The `RouteScript` slot can contain either a symbol identifying a function or it can contain a function directly. If you are defining the `routeScripts` array in a view definition, the `RouteScript` slot must contain a function directly. Alternatively, if your view definition is used only within your application, you can specify an `appSymbol` slot in the `routeScripts` frame and specify a symbol for the `RouteScript` slot. The `appSymbol` slot tells the system in what application (in the root view) to find the method identified by the `RouteScript` slot. Using the latter alternative ties the view definition to a single application.

Here is an example of how you might define the method identified by the `RouteScript` slot shown in the example frame above:

```
MyActionFunction: func(target,targetView)
   begin
      print("this is my action");
   end,
```

Handling Multiple Items

The target item, as returned by `GetTargetInfo`, may actually be a multiple item target object that encapsulates several individual items to be routed. You can check if this is the case by using the function `TargetIsCursor` (page 2-68). If the target item is a multiple item target object, and you need to act separately on the individual items, you can obtain a cursor for the items by using the function `GetTargetCursor` (page 2-67). Then you can use the standard cursor methods `Entry`, `Next`, and `Prev` to iterate over the cursor and return individual items. For more information about using cursors, refer to Chapter 11, "Data Storage and Retrieval," in *Newton Programmer's Guide: System Software*.

Note that `GetTargetCursor` works with any kind of target data, whether or not it's a cursor. So you don't need to call `TargetIsCursor` to check before calling `GetTargetCursor`.

Here's an example of a `RouteScript` method that uses `GetTargetCursor` to operate on multiple items:

```
MyActionFunction := func(target,targetView)
   begin
      local curs := GetTargetCursor(target,nil);
      local e := curs:Entry();
      while e do begin
         :DoMyAction(e); // do the operation
         e := curs:Next();
      end;
      // update display here
   end;
```

Handling No Target Item

If no target item is selected or there is nothing to do when the Action button is pressed, the system displays a warning message to inform the user of that fact. To take advantage of this warning message feature, all application-specific routing actions must be disabled when there is no target. (You may want to include some actions even when there is no target; in this case, you can ignore this section.)

To disable application-specific routing actions when there is no target, you can do one of two things:

- Define a `GetTitle` method in the `routeScripts` frame for each action, instead of a `title` slot. Then return `nil` from the `GetTitle` method to prevent that action from showing up on the picker.
- Define a `GetRouteScripts` method in your application, instead of a `routeScripts` slot. Then return `nil` or an empty array from the `GetRouteScripts` method to prevent any actions from showing up on the picker.

Sending Items Programmatically

Your application can send an item programmatically, using a specific transport, without any user intervention. (The Action button is not used in this case.) This is done using the global function `Send`, described on page 2-62.

Here is an example of how to use the `Send` function:

```
myItem := {
   toRef: nameRefObject, // a fax name ref
   title: "The Subject", // title of item
   body: theFax, // fax data frame
   appSymbol: kAppSymbol,
   currentFormat: kOtherPrintFormatSym
   };
Send('fax, myItem);
```

You must construct an item frame containing the data and other slots that you want to set in the item. You then pass this item frame as an argument to the `Send` function.

Before calling the `Send` function, you may want to allow the user to choose a format for the item being sent. To do this, you'll need a list of formats that can handle the item. To get a list of appropriate formats, you can use the `GetRouteFormats` function, described on page 2-63. Using this list, you could display a picker from which the user can choose a format.

You may also want to allow the user to choose a transport for the item being sent. To do this, you'll need a list of transports that can handle specific formats. To get a list of appropriate transports, you can use the `GetFormatTransports` function, described on page 2-64.

In the `Send` function, the Routing interface obtains a default item frame from the selected transport by sending the `NewItem` message to the transport. The slots you specify in your item frame are copied into the default item frame obtained from the transport. Note that the default frame supplied by the transport may contain other slots used by the transport.

The slots you include in the item frame vary, depending on the transport. The In/Out Box and transports ignore slots that they don't care about. Applications can use this feature to communicate information to multiple transports with the same item frame. For a comprehensive list of slots and detailed descriptions, see the section "Item Frame" beginning on page 2-44.

Here's a summary of the slots you might need to include in the item frame:

```
itemFrame := {
appSymbol: symbol, // appSymbol of sender
destAppSymbol: symbol, // receiving app, if different
body: frame, // the data to send
title: string, // item title, e-mail subject
text: string, // text of msg, for eWorld
toRef: array, // array of name refs for recipients
cc: array, // array of name refs for copied recipients
bcc: array, // array of name refs for blind copies
currentFormat: symbol, // routing format to use
connect: Boolean, // try to connect immediately?
hidden: Boolean, // hide UI and hide in Out Box?
covert: Boolean, // not logged or saved in Out Box?
completionScript: Boolean, // notify app of state change?
needsResolve: Boolean, // body slot contains an alias?
printer: frame, // a printer frame; the printer to use
coverPage: Boolean, // use a cover page for fax?
faxResolution: symbol, // 'fine or 'normal fax resolution
phoneNumber: string, // phone number, for call transport
name: string, // name, for call transport
serviceProvider: symbol, // 'modem, 'speaker, or nil
saveAsLog: Boolean, // log call in Calls app?
}
```

Note that you can set any of the Boolean slots in the `SetupItem` method of the routing format.

Applications implementing their own custom sending functionality apart from the Action button may need to open the transport routing slip view for the user. If you need to do this, you can use the global function `OpenRoutingSlip`, described on page 2-65.

Creating a Name Reference

For the built-in eWorld, fax, and call transports, addressing information for an item to send is stored in the `toRef` slot of the item frame, and certain other slots such as `cc`, `bcc`, and so on. These slots contain arrays of one or more name reference objects. A name reference is simply a frame that serves as a wrapper for a soup entry (often from the Names soup, thus the term "name reference"). The name reference may contain an alias to a soup entry and even some of the slots from the soup entry. Note that you must use name references; you cannot specify soup entries directly.

To create a name reference object, you use name reference data definitions registered with the system in the data definition registry. There are built-in name reference data definitions for e-mail (`'|nameRef.email|`), fax (`'|nameRef.fax|`), and call (`'|nameRef.phone|`) information associated with names from the Names file. These data definitions contain a method, `MakeNameRef`, that creates and returns a name reference.

You can pass a Names soup entry directly to `MakeNameRef`, or you can construct your own simple frame of information that contains the appropriate slots. Fax and call name references should include the slots `name`, `phone`, and `country`. E-mail name references should include the slots `name`, `email`, and `country`. For more information about these slots, see the documentation of Names soup entries in Chapter 18, "Built-In Applications and System Data," in *Newton Programmer's Guide: System Software*.

Here's an example of how to create a name reference for a fax phone number or an e-mail address:

```
// use a Names file entry directly
local myData := aNamesFileEntry; // entry from Names soup

// or create your own fake entry frame based on other info
```

```
local myData := {
   name:{first:"Juneau", last:"Macbeth"},
   phone: "408-555-1234", // fax phone string
   email: "jmacbeth@acompany.com", // e-mail address string
   country: "US",
}

// then create the fax name reference
toRef := GetDataDefs('|nameRef.fax|):MakeNameRef(myData,
                                                 '|nameRef.fax|);
// or create the e-mail name reference
toRef := GetDataDefs('|nameRef.email|):MakeNameRef(myData,
                                                 '|nameRef.email|);
```

For more information about name references and the `MakeNameRef` method, see the documentation of `protoListPicker` in Chapter 6, "Pickers, Pop-up Views, and Overviews," in *Newton Programmer's Guide: System Software*.

Specifying a Printer

For print operations, the `printer` slot of the item frame specifies which printer to use. This slot must contain a printer frame. The only valid way of obtaining a printer frame is from the `currentPrinter` slot of the user configuration frame. That slot holds the printer selected by the user as the current printer. You can use this function to obtain the current printer for the item:

```
item.printer := GetUserConfig('currentPrinter);
```

If you want to provide a way for the user to select a different printer, you can use the printer chooser proto, `protoPrinterChooserButton`, which is described on page 2-52. This proto changes the setting of the current printer in the system; it actually changes the `currentPrinter` slot of the user configuration frame.

If you don't want to change the current printer in the user's system, but just want to let them select a printer for this one print job, then you'll need to do the following things:

1. Get and temporarily save the current value of the `currentPrinter` slot in the user configuration frame, using `GetUserConfig`.
2. Display the printer chooser button, allowing the user to select a printer for this print job. When they select one, the printer chooser proto will automatically change the `currentPrinter` slot to the chosen one.
3. Retrieve the new value of the `currentPrinter` slot, using `GetUserConfig`, and use that for the `printer` slot in the item frame.
4. Reset the user's original printer choice by resetting the `currentPrinter` slot in the user configuration frame to the value you saved in step 1. You can use the function `SetUserConfig`.

The functions `GetUserConfig` and `SetUserConfig` are documented in Chapter 18, "Built-In Applications and System Data," in *Newton Programmer's Guide: System Software*.

Supporting the Intelligent Assistant

Besides using the standard interface for routing (the Action button), the user can also invoke routing actions by using the Intelligent Assistant and writing the name of the action. In order to determine what item to route, the Intelligent Assistant sends the `GetActiveView` message (page 2-69) to your application. This method returns the view to which the `GetTargetInfo` message should be sent.

The `GetActiveView` method is implemented by default in the root view and simply returns `self`, the current receiver. If this return value is not appropriate for your application, then you must override this method in your application base view.

Receiving Data

Incoming data arrives first as an entry in the In Box soup. If there is a public view definition registered for the class of the entry, the item may then be viewed directly in the In Box.

IMPORTANT

Generally, every received item must have a meaningful class. (This is not strictly required if the item has an `appSymbol` slot.) For frame items, the `class` slot identifies its data class. Frame items received from other Newton devices generally have a `class` slot. For items received from other systems, the transport must assign a meaningful class to each item (use `SetClass`). ▲

An incoming item may be stored in the In Box soup until the user chooses to manually put away the item into an application, or an incoming item may be transferred automatically to an application as soon as the item arrives in the In Box. This is controlled by the applications present on the Newton.

These are the minimum steps that you need to take to support receiving items through the In/Out Box in your application:

- Supply a `PutAwayScript` method in your application base view. When a user chooses to put away an item to your application from the In/Out Box, the item is passed to this method.
- Register the data types that your application can accept by using the `RegAppClasses` function in the application's `InstallScript` function. Unregister using `UnRegTheseAppClasses` or `UnRegAppClasses` in the application's `RemoveScript` function.

Automatically Putting Away Items

The first thing the In Box does with an incoming item is to determine which applications might want to accept the item immediately. The In Box does this by checking the In Box application registry (see the section "Registering to Receive Foreign Data" on page 2-40) to see if any applications have registered to accept such items. If a matching application is found in the registry, the `appSymbol` slot of the item is set to the value of the `appSymbol`

slot in the matching application. If no matching applications are found in the registry, the item may have a pre-existing `appSymbol` slot, which determines the application to which it belongs. If no matching application is located in the registry and the item has no existing `appSymbol` slot, it cannot be put away automatically.

Next, the In Box checks for an `AutoPutAway` method in the base view of the application whose `appSymbol` slot matches that in the item. If the `AutoPutAway` method exists, then the In Box sends the `AutoPutAway` message to the application, passing the incoming item as a parameter. In this way, items can be automatically transferred to an application, with no user intervention.

If the `AutoPutAway` method returns `nil`, this signals that the item could not be put away and the In Box leaves the item in the In Box soup.

If the `AutoPutAway` method returns a non-`nil` value, it is assumed that the application handled the item. In this case, the In Box takes an action on the item depending on the particular transport used. The item may be deleted from the In Box soup, the item may be deleted and a log entry for the item may be created in the In Box soup, or the item may be saved in the In Box soup. The value of the `inboxLogging` slot in the transport controls what happens to the item in the In Box.

If your application implements the `AutoPutAway` method, it must inform the system of this fact when it is installed. In the application part `InstallScript` function, you must call the global function `AppInstalled` to let the system know that the application is present. The `AppInstalled` function prompts the In Box to send an `AutoPutAway` message to the application for each In Box item that may have arrived for the application before the application was installed. (For a description of the `AppInstalled` function, see page 2-69.)

This feature is useful in cases where the application resides on a card which is not always installed in the system. Messages are held in the In Box soup while the application is not installed, and then when it is installed, those received messages are sent to the application with the `AutoPutAway` message.

The item passed to your application's `AutoPutAway` method is the entry from the In Box soup. It has several slots in it that are used by the In Box or the transport. Usually, the data your application uses is contained in the `body` slot. The `body` slot contains a copy of the `target` frame sent by the sending application.

If the item was sent by a custom transport that sends multiple item target objects (such as those created by `CreateTargetCursor`), then you might need to check if the body slot contains such an object by using `TargetIsCursor`. If so, you can get a cursor for the object by using `GetTargetCursor`, and then iterate over the cursor to handle individual items.

Manually Putting Away Items

If an item is not put away automatically, it resides in the In Box until the user chooses to put it away manually by tapping the Put Away button. When the user taps the Put Away button, the In Box displays a slip showing to which application the item will be put away. This application is the one that matches the `appSymbol` slot in the item. The In Box sends the `PutAwayScript` message to the base view of that application. The item is passed as a parameter.

The item passed to your application's `PutAwayScript` method is the entry from the In Box soup. It has several slots in it that are used by the In Box or the transport. Usually, the data your application uses is contained in the `body` slot. The `body` slot contains a copy of the `target` frame sent by the sending application.

If the item was sent by a custom transport that sends multiple item target objects (such as those created by `CreateTargetCursor`), then you might need to check if the body slot contains such an object by using `TargetIsCursor`. If so, you can get a cursor for the object by using `GetTargetCursor`, and then iterate over the cursor to handle individual items.

If the `PutAwayScript` method returns `nil`, this signals that the item could not be put away and the In Box leaves the item in the In Box soup and an alert is displayed telling the user that the item could not be put away.

If the `PutAwayScript` method returns a non-`nil` value, it is assumed that the application handled the item. In this case, the In Box takes an action on the item depending on the particular transport used. The item may be deleted from the In Box soup, the item may be deleted and a log entry for the item may be created in the In Box soup, or the item may be saved in the In Box soup. The value of the `inboxLogging` slot in the transport controls what happens to the item in the In Box.

If multiple applications have registered to accept data of the item's class, in the Put Away slip, the system displays a picker listing those applications. The application that matches the `appSymbol` slot of the item is listed as the default choice. If there is no `appSymbol` slot, or the application is missing, then a different application is the default choice. The user can choose the application to which the data is to be sent, and the `PutAwayScript` message is sent to that application.

The registry used for this operation is called the application data class registry; note that it is different from the In Box application registry mentioned above. Applications can register to accept data of one or more classes by using the `RegAppClasses` function, described on page 2-71.

It is recommended that all applications wanting to receive items through the In Box register their capability to receive data of particular classes by calling the `RegAppClasses` function. If your application is no longer interested in data of these classes, or your application is being uninstalled, you can unregister to receive these data classes by using the `UnRegTheseAppClasses` function, described on page 2-75.

You can check which applications can accept data of a particular class by using the `ClassAppByClass` function, described on page 2-69.

Registering to Receive Foreign Data

To receive data from a different application or from a non-Newton source, your application must register its interest in such data with the In Box

application registry. To do this, you use the `RegInboxApp` function, described on page 2-72.

If your application is no longer interested in foreign data, or your application is being uninstalled, you can unregister to receive foreign data by using the `UnRegInboxApp` function, described on page 2-74.

Note that your application can register to receive data from a different application. If you register a test function with `RegInBoxApp`, and that test function returns `true` for a particular item, the Routing interface will change the value of the `appSymbol` slot in the item to be the value of the `appSymbol` slot in your application. Be careful when using this feature not to intercept incoming items that should be destined for other applications. The In Box application registry takes priority over all other mechanisms that attempt to find an owner application for an incoming item.

Filing Items That Are Put Away

When an item is put away by an application, by default it is filed in the same folder on the receiving Newton as it was in on the sending Newton. This often makes it difficult for users to find new items, since they may be put away in folders that are undefined. To alleviate this problem, it is recommended that all incoming items be put away unfiled, so that users can more easily find items and file them where they want to. Incoming items should be put away unfiled even if the recipient has a folder of the same name as the sender.

For most applications, you put away an item unfiled by setting the `body.labels` slot of the item to `nil`. However, filing techniques vary, so this may not work for all applications.

Viewing Items in the In/Out Box

When data is queued in the Out Box, or has been received in the In Box and not automatically put away, the user can view the data directly in the In/Out Box. When the user chooses to view an item in the In/Out Box by tapping on

the item, the system looks for a view definition of the type `'editor` or `'viewer` that is registered for the class of that item.

Your application should register such view definitions with the system if you want users to be able to view items from your application in the In/Out Box. If you do not provide a view definition, and there are no other view definitions available for that data class, the In/Out Box displays a generic blank view for the item. Items formatted with the `'view` data type do not need a separate view definition because the In/Out Box itself provides a page preview view for these items.

Of the view definitions registered by your application, you can identify which should be made available to the In/Out Box and which should be hidden from the In/Out Box, by setting the `protection` slot in the view definition. Set the `protection` slot to `'public` to make the view definition available to the In/Out Box. Set the `protection` slot to `'private` to hide the view definition from the In/Out Box.

Note also that application view definitions used in the In/Out Box must not expect data defined within the context of the application. If the view definition needs to access application data, it should access the application through the root view (`GetRoot().appSymbol`).

For more information about writing and registering view definitions, refer to Chapter 5, "Stationery," in *Newton Programmer's Guide: System Software*. Note that the In/Out Box does not need data definitions, only view definitions.

Changing View Definition Behavior

Some applications may wish to provide different behavior for views depending on whether they are being displayed in the application itself or in the In/Out Box. For example, you may want to prevent editing of a view in the In/Out Box when this same view is editable in the application. Or you may want to show particular interface elements in your application but not in the In/Out Box.

To implement this behavior, your view definition must determine within which application it is running and change its behavior accordingly.

View Definition Slots

View definitions to be used by the In/Out Box have other slots of interest besides the `target` slot.

One other slot of interest is named `fields`. When the view is open, the `fields` slot contains a reference to the In/Out Box entry. If the entry has a `body` slot and the `body` slot contains a frame with a `class` slot, then the In/Out Box sets `target` to the `body` slot of the entry. This allows view definitions written for your application to be used by the In/Out Box without modification. If you need to access addressing or other information in the entry besides the actual data being routed, look at the frame in the `fields` slot. However, use of the `fields` slot is for special cases only and is generally discouraged. This is because it only works in the In/Out Box, and so ties your stationery to it. If you need to use the `fields` slot in your stationery, you should always check for the existence of this slot before using it, and you must be able to handle the case if it is missing.

Also, view definitions to be used by the In/Out Box can have a `rollView` slot. This slot contains a Boolean value. If you set this slot to `true`, the view is treated as a paper roll-based view that specifies its height. In this case, the In/Out Box handles scrolling within the view for you. If the `rollView` slot is set to `nil`, then scrolling functionality must be provided by the view definition itself.

Advanced Alias Handling

For sending data, an application may register a routing format that stores a sent object as an alias in the In Box soup. In fact, you can set a slot, `storeAlias`, in the routing format that allows this to happen. When such an object is to be sent by the transport, the Routing interface automatically resolves the alias into the actual object which is sent.

However, in some circumstances, you might want to provide your own alias handling. For example, you might want to store an object in the In Box as a complex frame consisting of some directly stored data and some slots that contain aliases. In this case, you would override the routing format method `MakeBodyAlias` (page 2-58) to construct your own object.

When the system needs to access the item, such as when it is viewed in the In/Out Box, it sends the message `ResolveBody` (page 2-59) to the format. You must override this method and use it to resolve the alias you constructed in the `MakeBodyAlias` method.

Note that if the send operation fails, the Out Box continues to store the original unresolved entry.

Routing Reference

This section describes the routines and protos provided by the Routing interface, and the data structures used when interacting with the Routing interface.

Data Structures

This section describes the data structures that your application uses to interact with the Routing interface.

Item Frame

The item frame is the frame that encapsulates a routed (sent or received) item and that is stored in the In/Out Box soup. Some slots have meaning only to the application that created the item, other slots have meaning only to the In/Out Box itself, and other slots are for the transport. Note that there are additional slots used just by the Transport interface that are not documented here. For more information, see Chapter 3, "Transport Interface."

Slot descriptions

`appSymbol` Required. This slot contains a symbol representing the sending application.

`destAppSymbol` Optional. A symbol identifying the application to receive the item, if it is different from the sending application. The receiving transport will set the

	`appSymbol` slot in the received item to this value, and the original value of the `appSymbol` slot will be stored in the `fromAppSymbol` slot in the received item frame.
`body`	Required, except for the eWorld transport. This slot contains a NewtonScript object representing the data to send. For eWorld, this data should be accessed only by eWorld email recipients and will not be exported outside of eWorld. For fax and print transports, this object should be referenced by the print format which will draw the page. Print formats should access this data using the expression `target` (not `fields.body`). All application-specific data and information should be contained in the `body` slot. In general, do not add application-specific slots to the item frame. This slot is optional for the eWorld transport because information can be passed in the `title` or `text` slots.
`title`	Optional. A string to be shown in the Out Box's view as the item's title. Note that you should not make this string very long, so it doesn't wrap to the next line in the Out Box. (Current software wraps the string to the next line at about 44 characters). If you don't supply this slot, but there is a data definition for the class of data being sent, the system tries to obtain a title from the data definition. So, if you use a data definition, you may not want to supply this slot. Note that for e-mail, this string is also shown as the message subject when the mail is viewed.
`text`	Used for eWorld only. A string that is the text of the mail message. Specify this slot if you don't use the `body` slot for eWorld items.
`toRef`	Required for some transports (of the built-in transports eWorld, fax, and call use this slot). This slot contains an array of one or more name references holding recipient address information. The type of name reference information differs, depending on the transport. For mail transports, the name references contain names and e-mail addresses; for the fax and call transports, they

	contain names and telephone numbers. For more information about creating name references, see the section, "Creating a Name Reference" beginning on page 2-34.
`cc`	Used by some transports (of the built-in transports, eWorld uses this slot). This slot contains an array of one or more name references holding e-mail addresses of people who should receive copies of the mail (like the "cc:" field of a memo heading).
`bcc`	Used by some transports (of the built-in transports, eWorld uses this slot). This slot contains an array of one or more name references holding e-mail addresses of people who should receive blind copies of the mail. This means they receive copies but their names don't appear on the recipient list; they are hidden from the other recipients.
`fromRef`	Optional. A name reference or other information that identifies the sender. This information is usually extracted from the sender's current owner card, or persona. Note that you don't normally set this slot; it is normally set by the transport in its `NewItem` method (see the section "Obtaining an Item Frame" beginning on page 3-14). If the format needs to get the sender name, it can get it from this name reference. If you do specify this slot, it will override the one provided by the transport.
`currentFormat`	Optional. A symbol representing the routing format to be used to represent this item. If this slot is not set, the Routing interface uses the first format it can find that handles the class of the data being sent.
`connect`	Optional. This slot is a Boolean. If set to `true`, it suggests to the transport that an immediate connection is appropriate. However, an immediate connection cannot be guaranteed. For instance, the beaming transport might observe this slot and immediately try to

send the beam to another Newton. Some transports may disregard this slot and implement their own behavior.

`hidden` Optional. This slot is a Boolean. If set to `true`, the Out Box will hide the entry so it can't be seen, selected, or even deleted by the user.

IMPORTANT

All applications that set `hidden` to `true` must also set `completionScript` to `true` and must have an `ItemCompletionScript` method. This allows you to keep track of hidden items and delete them after they are sent (since the user can't). If you fail to supply an `ItemCompletionScript` method, the `hidden` slot will be removed from the item frame by the system. ▲

`covert` Optional. This slot is a Boolean. If set to `true`, the Out Box will not log or save this item after it is sent.

`completionScript`
Optional. This slot is a Boolean. If set to `true`, the application will be notified when the state of the item changes or when errors occur. This allows an application to track what happens to sent items. The application, identified by the `appSymbol` slot in the item frame, is sent the `ItemCompletionScript` message (page 2-76). This method must be defined in the application base view, if you want to be notified.

`needsResolve` Optional. This slot is a Boolean. Set it to `true` if the `body` slot contains an alias, rather than the actual data.

`printer` Optional. A printer frame used for printing only. This frame specifies the printer to use. If this slot is omitted, the last printer selected by the user will be used. This is obtained from the `currentPrinter` slot of the user configuration frame. For more information on how to specify a printer, see the section "Specifying a Printer" beginning on page 2-35.

Slot descriptions that apply to the built-in fax transport only

`coverPage` Optional. This slot is a Boolean. If set to `true`, a cover page will be printed. If `nil`, no cover page is printed. If this slot is omitted, the user preference setting will be observed.

`faxResolution` Optional. A symbol indicating the fax resolution to use. Specify either `'fine` or `'normal`. If this slot is omitted, the default resolution is `'fine`.

Slot descriptions that apply to the built-in call transport only

`phoneNumber` Optional. A string that is the phone number to dial. (This is required in addition to the `toRef` slot, if this transport is being used in conjunction with the Calls application.)

`name` Optional. A string that is the name of the person to call. (This is required in addition to the `toRef` slot, if this transport is being used in conjunction with the Calls application.)

`serviceProvider` Optional. A symbol identifying how the call should be placed. Specify `'modem` to dial it through the modem, or `'speaker` to dial it through the speaker, or `nil` to signify that the Newton device is not dialing the call at all (you're just logging a call that the user is dialing manually). If this slot is not specified, the current user preference setting is used.

`saveAsLog` Optional. This slot is a Boolean. If set to `true`, the Calls application is opened when the call is placed and an entry is made to log the call. If set to `nil`, no log entry is made and the Calls application is not opened. If this slot is not specified, the last user setting for the Log check box in the call routing slip is used.

RouteScripts Array

The `routeScripts` slot in an application contains an array of frames, where each frame corresponds to one application-specific routing action to be

displayed on the Action picker. Each of these `routeScripts` frames is defined as follows:

```
{
title: string,          // string name of picker item
icon: bitmap object,    // icon for picker item
RouteScript: symbol,    // func called if this action chosen
appSymbol: symbol,      // symbol for context of RouteScript
GetTitle: function      // supplied instead of title slot
...                     // other slots used by your app
}
```

Slot descriptions

`title`	Optional. A string that appears in the Action picker. If this slot is `nil` or missing, the `GetTitle` method is used to get the title for the picker.
`icon`	Optional. An icon that appears to the left of the item in the picker.
`RouteScript`	Required. A symbol identifying a function that is called if this routing action is selected from the picker. (Alternately, you can include the function directly in this slot.) The specified function is passed two arguments, the `target` and `targetView` slots as returned by the message `self:GetTargetInfo('routing)`. Note that `self` evaluates to the Action button view, where the lookup for these two slots begins.
`appSymbol`	Optional. A symbol identifying an application in the root view where the function identified by the `RouteScript` symbol can be found. This slot is used only if the `RouteScript` slot contains a symbol and this frame is defined in a view definition rather than in an application.
`GetTitle`	Optional. If the `title` slot is `nil` or missing, this method is used to obtain the title. This method takes one parameter, the `target` slot of the item being

routed. (This slot is obtained by the system sending the message `self:GetTargetInfo('routing)`.)

The `GetTitle` method must return a title string, or `nil`. If this method returns `nil`, then the action will not show up in the picker. The `GetTitle` method allows you to return different titles depending on the target item.

For more information on the `GetTargetInfo` method, see the section "GetTargetInfo" beginning on page 2-70. Note that your application can override this method to return custom data.

Format Frame

To support routing through transports, your application uses routing formats that specify what data types your application routes and how the data is to be formatted when it is routed. These routing formats are frames defined as follows:

```
{_proto: routing proto,  // proto for format
dataTypes: array,        // identifies supported data types
title: string,           // name of format
symbol: symbol,          // unique id - include signature
type: 'printFormat,      // identifies this format as
                         // used for routing
version: integer,        // version number of format
auxForm: viewTemplate,   // defines auxiliary view
storeAlias: Boolean,     // store alias?
showMessage: Boolean,    // warn user when aliasing?
sizeLimit: integer,      // maximum size without aliasing
storeCursors: Boolean,   // store multiple items as a cursor?
usesCursors: Boolean,    // format handles cursor for layout?
orientation: symbol,     // 'portrait or 'landscape
margins: boundsFrame,    // margin insets for layout
SetupItem: function,     // called if this format selected
TextScript: function,    // gets text data from item
```

```
TargetSize: function,   // determines target size
MakeBodyAlias: function, // makes an alias
ResolveBody: function,  // resolves alias body slot
PrintNextPageScript: function,   // constructs next page
GetCursorFormat: function,       // gets format for an item
FormatInitScript: function,      // initialization
CountPages: function,            // counts pages for a fax
}
```

Note that many of the slots shown above are optional, and some apply only to formats based on specific routing format protos.

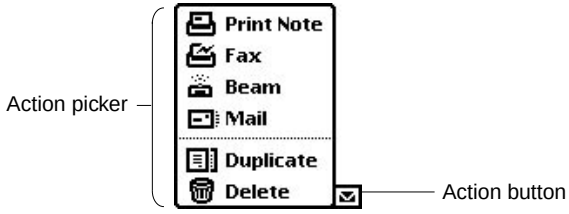
For descriptions of the slots, refer to the individual protos in the following sections.

Protos

This section describes protos used in the Routing interface.

protoActionButton

This proto is used to include the Action button in a view. When the user taps on the Action button, a picker is dynamically created and displayed. The picker lists those actions that the current application has implemented and that are supplied by transports that can handle the target data. When an item from the picker is selected, a routing slip may be displayed, and if confirmed, the target item selected in the application is routed. Here is an example of the Action button and picker:



Slot descriptions

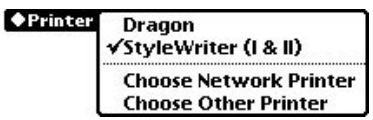
`viewBounds`	By default, the Action button is placed in the upper-right corner of its parent view. The default top-left coordinate is (–42, 2). Set this slot if you want to change the icon's location. It is recommended that you put the Action button with other buttons on a status bar, if you have one.
`viewJustify`	Optional. The default setting is `vjCenterH + vjCenterV + vjParentRightH`.
`viewFormat`	Optional. The default setting is `vfFillWhite + vfFrameBlack + vfPen(2) + vfRound(4)`.

The following additional methods are defined internally: `ViewClickScript`, `ButtonClickScript`, `PickActionScript`, and `PickCancelledScript`. If you need to use one of these methods, be sure to call the inherited method also (for example, `inherited:?ViewClickScript(unit)`), otherwise the proto may not work as expected.

The `protoActionButton` uses the `protoPictureButton` as its proto; and `protoPictureButton` is based on a view of the `clPictureView` class.

protoPrinterChooserButton

This proto is used to include the printer chooser button in a view. When the user taps on the button, a picker is displayed. The picker lists recent printers that the user has chosen, along with items that allow the user to choose another built-in printer or a network printer. If the user chooses to select a network printer and is connected to a network, a scrollable list of printers found on the network is displayed. Here is an example of the printer chooser button and picker:



Slot descriptions

`viewBounds`	Set to the location where you want the printer chooser button to appear.

The `protoPrinterChooserButton` uses the `protoLabelPicker` as its proto.

Routing Format Protos

The three routing format protos, `protoRoutingFormat`, `protoPrintFormat`, and `protoFrameFormat`, are described together in this section because they share many common slots and methods. In fact, `protoRoutingFormat` serves as a proto to the other two. The common information is labeled as such, and is followed by the information that applies to the individual protos.

Slot descriptions common to all proto routing formats

`type`	Required. This slot is set to `'printFormat`. You shouldn't change it.
`title`	Required. A string identifying this format. This string is displayed in the picker listing formats in the routing slip.
`symbol`	Required. A symbol that uniquely identifies this format from all others. This is used to save the current format. Be sure to append your developer signature (for example, `'|aFormat:mySIG|`).
`dataTypes`	Required. An array of symbols set to the data types that this format supports. The currently defined types in the system are listed in Table 2-1 on page 2-10. The default value of this slot in `protoRoutingFormat` and `protoFrameFormat` is `['frame,'text]`. The default value in `protoPrintFormat` is `['view]`.
`version`	Optional. An integer identifying the version of this format.
`auxForm`	Optional. A view template. This optional auxiliary view is used to gather extra information from the user in the routing slip view. If this slot is provided, the auxiliary view is opened when the format is selected.

`storeAlias`	Optional. If you set this slot to `true`, and the target is larger than `sizeLimit` or there is not enough storage space for it, an alias to the target object is assigned to the `body` slot of the item frame in the default `SetupItem` method (see page 2-56). The default value of this slot is `nil`. For more information, see the `SetupItem` method.
`showMessage`	Optional. When an alias to the target object is stored, the system warns the user that the original item must be available when the routed item is sent. The display of that message is controlled by this slot. When set to `true`, this slot enables the message and when set to `nil`, this slot suppresses the message. The default value of this slot is `true`.
`sizeLimit`	Optional. An integer specifying a number of bytes. If `storeAlias` is `true` and the target object exceeds this number of bytes (or there is not enough storage space for it), then an alias to the target object is assigned to the `body` slot of the item frame. The default value of this slot is `nil` (meaning there is no limit).
`storeCursors`	Optional. This slot controls how a selection of multiple items from an overview are handled. If you set this slot to `true`, and the transport also handles cursors, a selection of multiple items is stored in the Out Box as a multiple item target object (flattened cursor) that is later resolved into its component entries. If you set this slot to `nil`, a selection of multiple items is resolved into separate entries which are stored individually in the Out Box. The default value of this slot is `true`. Note that the transport slot `allowBodyCursors` must also be set to `true` for a cursor to be used. If this is not the case, then a cursor will not be used, even if `storeCursors` is set to `true`; each item will be stored separately in the Out Box. Of the built-in transports, only the print and fax transports handle cursors.

Slot descriptions for the `protoPrintFormat` variant

`usesCursors`	Optional. Set this slot to `true` if this format can handle laying out multiple items on the same page when multiple items are being routed. In this case, the format is passed a single cursor to the items being routed. If you want each item to be printed on a separate page or if this format cannot handle a cursor, set this slot to `nil`. In this case, the format is called multiple times, once for each item being routed. The default setting of this slot is `nil`.
`orientation`	Optional. A symbol indicating whether this format should use the paper in portrait mode (`'portrait`), or horizontally in landscape mode (`'landscape`). The default is `'portrait`.
`margins`	Optional. A bounds rectangle giving the margins to be used when laying out the items on the page. The value of each slot (`left`, `top`, `right`, `bottom`) in this frame is interpreted as an inset from the edge of the paper in pixels. The default is `{0,0,0,0}`.
`viewFlags`	Optional. The default setting is `vVisible + vReadOnly`.
`viewJustify`	Optional. The default setting is `vjParentFullH + vjParentFullV`.
`viewFont`	Optional. The default font is `userFont12`.
`pageWidth`	The `viewSetupChildrenScript` method of the proto sets this slot to the width, in pixels, of the view.
`pageHeight`	The `viewSetupChildrenScript` method of the proto sets this slot to the height, in pixels, of the view.

The methods that are of interest in these three routing format protos are described in the following subsections. The common methods are described first, followed by the methods that apply to the individual protos. The following methods apply to all of the routing format protos:

```
SetupItem
TextScript
TargetSize
MakeBodyAlias
ResolveBody
```

The following methods apply only to `protoPrintFormat`:

```
ViewSetupChildrenScript
PrintNextPageScript
GetCursorFormat
FormatInitScript
CountPages
```

Note also that the following methods are defined internally in `protoPrintFormat`: `ViewSetupFormScript` and `ViewSetupChildrenScript`. If you need to use one of these methods, be sure to call the inherited method also (for example, `inherited:?ViewSetupFormScript()`), otherwise the proto may not work as expected.

SetupItem

format`:SetupItem(`*item*`,` *targetInfoFrame*`)`

This method is called if this format is selected from the picker listing formats in the routing slip. This method must set the `body` slot of the *item* frame to the data to be routed. Additionally, you can use this method to initialize other slots in the *item* frame; however, do not put any application-specific data into other slots, as they are not guaranteed to be preserved. For instance, they won't be copied if the item is rerouted from the In/Out Box.

item	An item frame, as obtained from the transport method `NewItem` (page 3-68). For more information about the item frame, see the section "Item Frame" beginning on page 3-3.
targetInfoFrame	The target information frame returned by the method `GetTargetInfo` (page 2-70).

The routing format protos provide a default `SetupItem` method that simply assigns the `target` slot in *targetInfoFrame* to the `body` slot of *item*. You can override this method if you want to perform additional operations and then call the inherited `SetupItem` method. For more information on using this method, see the section "Supplying the Target Object" beginning on page 2-16.

The default `SetupItem` method returns *item*, after the `body` slot in it has been set. If it returns `nil`, the item won't be routed and the user is notified by the system that the item could not be sent.

IMPORTANT
The `SetupItem` method should not assume that the application associated with the item is open. The In/Out Box might be rerouting the item, separate from the application. In this case, the application gets a chance to modify the item in its `VerifyRoutingInfo` method (page 2-77), which the In/Out Box calls in the application that owns the item. ▲

You can use the `storeAlias` slot in the routing format frame to specify that an alias to the target soup entry is to be stored in the `body` slot. For more information on the `storeAlias` slot, see its description on page 2-54. The default `SetupItem` method also handles creating and storing an alias if the `storeAlias` slot is `true`, and handles the `sizeLimit` slot.

TextScript

format`:TextScript(`*item*`,` *target*`)`

Transports might call this method is called to obtain a textual representation of the data to be routed. This method is only called by transports that handle text type data. Of the built-in transports, only the eWorld transport calls this method.

item	The item frame.
target	The target object to be routed in the application.

The routing format protos provide a default `TextScript` method that attempts to get the textual representation of the data from the data definition

registered with the system. First it calls the `TextScript` method of the data definition, then it looks in the `description` slot of the data definition, and lastly it tries the `name` slot of the data definition. You can override this behavior if you want by providing your own `TextScript` method.

TargetSize

format`:TargetSize(`*targetInfoFrame*`)`

You must override this method of the routing format protos if you need to determine the size of a target object that is not a soup entry. This method must return an integer that is the size of the target object in bytes.

targetInfoFrame The target information frame passed to the method `SetupItem`.

If you can't determine the size of the target object, return `nil` from this method.

The proto provides a default `TargetSize` method that works for soup entries. It simply uses the `EntrySize` function to determine the size of the object.

MakeBodyAlias

format`:MakeBodyAlias(`*targetInfoFrame*`)`

You must override this method of the routing format protos if you need to make an alias for some special target object that is not a soup entry. In this method, you must make an alias object (in whatever way you want) and return it.

targetInfoFrame The target information frame passed to the method `SetupItem` (page 2-56).

The alias object that you return must have a `class` slot whose value is the class of the target object.

Note that if you provide this method, you must also provide a `ResolveBody` method that is able to resolve the alias.

ResolveBody

format:`ResolveBody(`*item*`)`

You must override this method of the routing format protos if you have provided a `MakeBodyAlias` method. `ResolveBody` must resolve and return the `body` slot of *item*. This method is called by the system whenever it needs to access the original target item.

item The item frame.

The default `ResolveBody` method returns the `body` slot of *item*, resolving an alias stored there, if necessary. Note that this method works whether or not the `body` slot of *item* is an alias.

ViewSetupChildrenScript

format:`ViewSetupChildrenScript()`

Typically, you use this method to set up the child views containing the data to be routed. When this method is called initially, you should set up the child views for the first page to be routed, typically by setting the value of the `stepChildren` array. If you follow the guidelines for the `PrintNextPageScript` method by using the view method `RedoChildren`, as a result, the `ViewSetupChildrenScript` method will be called for each subsequent page as well. Don't forget to call the inherited method (`inherited:?ViewSetupChildrenScript`) so that the proto behavior is preserved.

PrintNextPageScript

format:`PrintNextPageScript()`

You must define this method of the `protoPrintFormat` if your print format handles more than a single page of data. The system calls this method each time it reaches the end of a page to allow you to construct the next page of data. This method should construct the view for the next page of data so that the message `self:Dirty()` will show the view.

Typically, you do this by keeping track of what data has been routed so far. Then when the format receives this message, you set up child views

representing the next page of data to send, and send the `RedoChildren` message (which sends the `ViewSetupChildrenScript` message) to create the new child views representing the next page of data to route. For information on `RedoChildren` and other view methods, refer to Chapter 3, "Views," in *Newton Programmer's Guide: System Software*.

While there is more data to route, this method should return a non-`nil` value. When there is no more data to route, this method should return `nil`.

Note that some transports (for example, fax) might call this method before the data is actually printed to determine the page count.

For more information on using this method, see the section "Printing and Faxing" beginning on page 2-23.

GetCursorFormat

format`:GetCursorFormat(`*target*`)`

You can call this method of `protoPrintFormat` to return a format for a given target object.

This method is useful for getting formats for the individual items described by the cursor as you iterate through them.

target The target object to be routed in the application.

This method looks for a format registered as a view definition for the data class of the target object whose `symbol` slot matches the `symbol` slot of the view format in which this method is called. If no matching format is found, this function returns the first format registered for the data class of the target object that is for the `'view` data type and whose `usesCursors` slot is `nil`.

If no format is found, `nil` is returned.

FormatInitScript

format`:FormatInitScript(`*item*`,` *target*`)`

If the print format is to be used for faxing, you can supply this method in your print format to perform any lengthy initialization operations that you

want to do before a fax connection is made. This method is guaranteed to be called before a connection is made.

item	The In Box entry.
target	The data object to be faxed. This is usually the contents of the *item*`.body` slot.

For more information on using this method and faxing, see the section "Printing and Faxing" beginning on page 2-23.

CountPages

format`:CountPages(`*item*`,` *target*`)`

You should override this method of `protoPrintFormat` to return the number of pages in the fax (not including the cover page, if present), if you can determine it.

item	The In Box entry.
target	The data object to be faxed. This is usually the contents of the *item*`.body` slot.

The default `CountPages` method of the `protoPrintFormat` opens the print format view in an offscreen view and causes each page to be constructed in turn so it can count the number of pages (not including the cover page). The `PrintNextPageScript` message is sent to the print format after each page is done. Then the print format view is closed. This is a lot of work for the system to do just to determine the number of pages, so if you can, it's a good idea to override the `CountPages` method with one of your own.

For more information on using this method and faxing, see the section "Printing and Faxing" beginning on page 2-23.

Functions and Methods

This section describes send-related and utility functions and methods for the Routing interface.

Send-Related Functions and Methods

This section describes functions and methods used when an application sends an item programmatically.

Send

`Send(`*transportSym*`,` *item*`)`

Stores an item in the Out Box and routes it to the indicated transport.

transportSym A symbol representing the transport (or transport group) to which the item should be routed. You must specify an installed transport that supports sending, or a transport group symbol (built-in groups include `'fax`, `'beam`, `'mail`, and `'print`). If you specify a group symbol, the last used transport from that group is used to send the item. To obtain a list of valid transports for the item you are sending, you can use the functions `GetRouteFormats` and then `GetFormatTransports`.

item A frame containing slots that you want added to the item frame which is posted to the Out Box. This must include routing information and data to be sent. For a discussion on how to construct this frame and detailed descriptions of the slots, see the section "Sending Items Programmatically" beginning on page 2-32. The slots are described briefly here.

If successful, this function returns the item stored in the Out Box soup, otherwise it returns `nil`.

Here's a summary of the slots you can include in the item frame:

```
itemFrame := {
appSymbol: symbol, // appSymbol of sender
destAppSymbol: symbol, // receiving app, if different
body: frame, // the data to send
title: string, // item title, e-mail subject
text: string, // text of msg, for eWorld
```

```
toRef: array, // array of name refs for recipients
cc: array, // array of name refs for copied recipients
bcc: array, // array of name refs for blind copies
currentFormat: symbol, // routing format to use
connect: Boolean, // try to connect immediately?
hidden: Boolean, // hide UI and hide in Out Box?
covert: Boolean, // not logged or saved in Out Box?
completionScript: Boolean, // notify app of state change?
needsResolve: Boolean, // body slot contains an alias?
printer: frame, // a printer frame; the printer to use
coverPage: Boolean, // use a cover page for fax?
faxResolution: symbol, // 'fine or 'normal fax resolution
phoneNumber: string, // phone number, for call transport
name: string, // name, for call transport
serviceProvider: symbol, // 'modem, 'speaker, or nil
saveAsLog: Boolean, // log call in Calls app?
}
```

GetRouteFormats

`GetRouteFormats(`*item*`)`

Returns an array of routing formats registered in the system that can handle the class of the specified item. If no formats are found that can handle the item, `nil` is returned.

item — The item to be routed. The item is used only to obtain a class symbol.

Note that this function returns an array of actual format frames, not just symbols identifying formats.

You can pass the return value from this function to the `GetFormatTransports` function to get a list of transports that can send an item.

GetFormatTransports

`GetFormatTransports(`*formatArray*`, `*target*`)`

Returns an array of transports that can send data using the specified formats. If no transports are found that can handle the specified formats, an empty array is returned.

formatArray	An array of routing format frames. You can obtain this array from the `GetRouteFormats` function.
target	A frame, which is the `target` slot from the target information frame returned by the `GetTargetInfo` function (page 2-70).

Note that this function returns an array of actual transport frames, not just symbols identifying transports.

GetDefaultFormat

view`:GetDefaultFormat(`*transport*`, `*target*`)`

Gets the default format for a given transport (and target item).

transport	A symbol identifying a transport.
target	A frame, which is the `target` slot from the target information frame returned by the `GetTargetInfo` function (page 2-70) and verified by the `VerifyRoutingInfo` method (page 2-77).

This method is used to get the default format for a transport and target item. It should return a format frame or `nil`, if none are found or appropriate.

You do not need to implement this method because there is a default method implemented in the root view. The default method looks in the `lastFormats` slot of `self` (the application base view) to find a transport matching *transport*. If the transport is found, it returns the format stored in that slot, which is the last format used with that transport.

The `GetDefaultFormat` method is called only if a routed item's `appSymbol` slot is appropriately set and the application is present.

If you implement this method, you can use the *target* parameter to base the format you return on the target item in addition to the transport. The *target* parameter is ignored by the default method. Also, note that the variable `self` will evaluate to the application base view of the application that sent the item.

SetDefaultFormat

view`:SetDefaultFormat(`*transport*`,` *target*`,` *format*`)`

Sets the default format for a given transport (and target item).

transport	A symbol identifying a transport.
target	A frame, which is the `target` slot from the target information frame returned by the `GetTargetInfo` function (page 2-70) and verified by the `VerifyRoutingInfo` method (page 2-77).
format	A routing format frame.

This method is used to set the default format for a transport and target item.

The `SetDefaultFormat` method is called only if a routed item's `appSymbol` slot is appropriately set and the application is present.

You do not need to implement this method because there is a default method implemented in the root view. The default method stores *format* in this slot in `self` (the application base view):

`lastFormat.`*transport*

If you implement this method, you can use the *target* parameter in addition to the transport to do something different. The *target* parameter is ignored by the default method. Also, note that the variable `self` will evaluate to the application base view of the application that sent the item.

OpenRoutingSlip

`OpenRoutingSlip(`*item*`,` *targetInfo*`)`

Opens the routing slip for a transport.

item	An item frame as returned by the transport `NewItem` method, described on page 3-68.
targetInfo	This parameter must be a frame, containing `target` and `targetView` slots, as returned by the `GetTargetInfo` function (page 2-70).

If successful, this function returns the routing slip view, otherwise it returns `nil`. The routing slip view is returned to you so that you can close it if you need to, for example, if your application is closed.

In certain error conditions, this function can also return the symbol `'skipErrorMessage`. This return value means that the routing slip did not open because of an error, but the user has already been alerted by a warning message, so you don't need to display another message.

Your application can call this function to open the transport routing slip directly, bypassing the Action button, which would normally be used to open it automatically.

This function does much of the work in the Routing interface, performing initialization operations and calling the `SetupItem` method defined in the routing format.

Note that if the *item*`.state` slot is non-`nil`, `OpenRoutingSlip` does no initialization operations, nor does it call the `SetupItem` method. In this case, the assumption is that since the state of the item is non-`nil`, it has already been initialized.

Cursor-Related Functions

This section describes functions related to creating and testing for cursor objects.

CreateTargetCursor

`CreateTargetCursor(`*class*`, `*dataArray*`)`

Creates and returns a frame that encapsulates an array holding multiple target items.

class	A symbol identifying the data class to be used for the returned object. Only routing formats registered under this class symbol will be able to route this object.
dataArray	An array of items for which a multiple item target object is to be created. These can be soup entries, soup aliases, or any kind of NewtonScript object. The array items can be of mixed types.

The object returned is a frame that encapsulates multiple target objects and can be stored in a soup. If you want to navigate the individual target items with a cursor, you can get a cursor by calling `GetTargetCursor` and passing it the object returned by `CreateTargetCursor`. For example:

```
multiItemTarget := CreateTargetCursor('|myClass:SIG|, anArray);
aCursor := GetTargetCursor(multiItemTarget, nil);
```

If there is a data definition registered for the data class identified by the *class* parameter, then the `CreateTargetCursor` function sends the `CreateCursor` message to the data definition to let it create the multiple item target object. If there is no data definition, or if the `CreateCursor` method is not implemented, then `CreateTargetCursor` creates its own multiple item target object and gives it the class *class*.

Note that the built-in routing format protos are designed to handle multiple item target objects by finding a format for each item. You can override this behavior if you design your own format to handle multiple item target objects.

GetTargetCursor

`GetTargetCursor(`*target*`, `*param2*`)`

Returns a cursor for the *target* object.

target	The target object to be routed.
param2	Reserved for future use. Always set this parameter to `nil`.

Note that this function always returns a cursor, regardless of whether the *target* parameter is multiple item target object, a single target object, or `nil`. Of course, in the latter case, the cursor object will not point to any objects.

Note that the object returned by the cursor method `Entry` may not always be a soup entry—it can be any NewtonScript object. If a cursor entry is a soup alias, it will automatically be resolved when you access it by using one of the cursor methods. If the alias cannot be resolved, the cursor method might return the symbol `'deleted` (if the item is removed while you are iterating over the cursor), but usually it will just skip over the unresolved item. Subsequent calls to the cursor methods `Next` and `Prev` will skip over the unresolved item.

The cursor object returned by this function is not exactly the same as a standard soup cursor returned by the soup `Query` method. However the cursor object returned by this function can be used like a standard cursor in that it responds to the following cursor methods: `Entry`, `Next`, and `Prev`. You can find these cursor methods described in Chapter 11, "Data Storage and Retrieval," in *Newton Programmer's Guide: System Software*.

TargetIsCursor

`TargetIsCursor(`*target*`)`

Returns `true` if the target object to be routed consists of a cursor (to multiple objects), or returns `nil` if it's not a cursor.

target	The target object to be routed.

If the target is a cursor, you can use the function `GetTargetCursor` to obtain the cursor.

The `TargetIsCursor` method returns true for the multiple item target objects created by `CreateTargetCursor`, since these objects represent flattened cursors.

Utility Functions and Methods

This section describes utility functions and methods used in the Routing interface, in alphabetical order.

AppInstalled

`AppInstalled(`*appSymbol*`)`

Informs the system that your application implements the `AutoPutAway` method (page 2-75). If your application uses an `AutoPutAway` method, you must call the `AppInstalled` function from your application `InstallScript` function so that the system knows about it.

appSymbol A symbol identifying your application.

ClassAppByClass

`ClassAppByClass(`*dataClass*`)`

Returns an array of application symbols corresponding to applications that are registered to accept items of the specified data class. If no applications are found with the specified data class, `nil` is returned.

dataClass A symbol identifying a data class.

GetActiveView

view`:GetActiveView()`

Returns the current receiver (`self`), which is the view to which this message is sent.

The Intelligent Assistant sends this message when the user initiates a routing action through it. The message is sent to the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot (not including floating views, as indicated by the `vFloating` flag). The purpose of this method is to return the view to which the `GetTargetInfo` message should be sent by the Intelligent Assistant, so that it can determine what object to route.

If the default return value of `self` is not appropriate for your application, then you should override this view method in your application base view. This method should return the view to which the `GetTargetInfo` message should be sent by the Intelligent Assistant, so that it can determine what object to route.

GetItemTransport

`GetItemTransport(`*item*`)`

Returns the transport used for an item being sent or received.

item The item for which you want to get the transport.

GetRouteScripts

view`:GetRouteScripts()`

Returns the value of the `routeScripts` slot, using full proto and parent inheritance and starting in the context of `self`. If the `routeScripts` slot is not found, or if it contains an empty array, an alert is displayed to the user saying that nothing is selected.

You might want to override this method in your application if you decide to build the `routeScripts` array dynamically. For more information, see the section "Providing Application-Specific Routing Actions" beginning on page 2-27.

GetTargetInfo

view`:GetTargetInfo(`*reason*`)`

This view method retrieves target information from the view to which this message is sent.

reason A symbol identifying what the information is to be used for. This parameter is useful if you override this method. It is provided as hook for you to implement special behavior depending on its value. During a routing operation, the system sends the

`GetTargetInfo` message with this parameter set to the value `'routing`.

This method returns a frame that has the following slots:

`target`	The value of the `target` slot. The search for this slot begins in the view receiving the `GetTargetInfo` message and uses full proto and parent inheritance.
`targetView`	The value of the `targetView` slot. The search for this slot begins in the view receiving the `GetTargetInfo` message and uses full proto and parent inheritance.
`targetStore`	If `target` is a soup entry, then the store on which the entry resides is returned in this slot.

You can override this method, defining other *reason* values that cause different or additional information to be returned in the resulting frame. The frame returned by your version of this method must include the `target`, `targetView`, and `targetStore` slots, however.

Since you can return only a single target item, if multiple items are selected for routing, you will need to create a single object that encapsulates them. You can use the function `CreateTargetCursor` (page 2-67) to create a multiple item target object that can be stored in a soup. (Normal cursors can't be stored in a soup.)

RegAppClasses

`RegAppClasses(`*appSymbol*`,` *dataClasses*`)`

Registers an application to accept data of the specified classes.

appSymbol	A symbol identifying your application or transport which is registering to handle this data. Specify the value of the `appSymbol` slot of the application or transport.
dataClasses	An array of symbols identifying data classes that your application can accept.

This registry is used when the user chooses to put away an In Box item. The In Box displays a picker listing all of the applications that have registered to

handle items with that data class. The user can choose to which application the item should be put away. If the user chooses your application, it will be sent the `PutAwayScript` message, with the item to be put away. The `PutAwayScript` method should be able to handle data of all the classes for which you have registered with `RegAppClasses`.

RegInboxApp

`RegInboxApp(`*appSymbol*`,` *test*`)`

Registers an application with the In Box to receive data from other applications or non-Newton sources. Whenever a new item is added to the In Box, the In Box checks the registered applications to find an owner for the new item.

appSymbol A symbol identifying your application.

test A string or a function object used to match an incoming item with an application. If you specify a string, then the string will be compared with the `title` slot in the incoming item. If the string in the `title` slot begins with the *test* string, then the item's `appSymbol` slot is set to the value in your application's `appSymbol` slot.

If you specify a function object for *test*, the function is called with the incoming item as its parameter. If the function returns `true`, the item's `appSymbol` slot is set to the value in your application's `appSymbol` slot.

RegisterViewDef

`RegisterViewDef(`*formatFrame*`,` *classSymbol*`)`

Registers a routing format frame as a view definition in the global registry.

formatFrame A routing format frame.

classSymbol A symbol under which the routing format is to be registered. This symbol corresponds to the class of objects with which this routing format can be used. Be sure to append your developer signature, because this

symbol should be unique in the view definition registry, unless you're adding a format to an existing class of formats.

For more information about view definitions and the use of this global function, refer to Chapter 5, "Stationery," in *Newton Programmer's Guide: System Software*.

TransportNotify

`TransportNotify(`*transport*`,` *message*`,` *paramArray*`)`

Sends a message to a transport or to all transports.

transport A symbol identifying the transport to which you want to send a message. You can specify a transport group symbol, and the message will be sent to the current (last used) transport in that group. Specify the symbol `'_all` to send the message to all transports.

message A symbol that is the name of the message to send.

paramArray An array of parameters to be passed with the message.

The `TransportNotify` function returns the return value of the message it sent. If it is broadcasting to all transports, it returns the return value of the last message it sent.

If *transport* is not the symbol `'_all` and the method does not exist in the transport, the symbol `'NoMethod` is returned.

If *transport* is not found, the symbol `'noTransport` is returned.

The `TransportNotify` function is a mechanism that can be used by applications to communicate directly to any number of transports without making specific calls to a single transport.

There are two messages that the system currently sends to transports by using `TransportNotify`. They are `AppOpened` (page 3-50) and `AppClosed` (page 3-49).

UnRegAppClasses

`UnRegAppClasses(`*appSymbol*`)`

Unregisters an application (and all its data classes) that had previously been registered by the function `RegAppClasses`.

appSymbol A symbol identifying your application.

UnRegInboxApp

`UnRegInboxApp(`*appSymbol*`)`

Unregisters an application that had previously been registered by the function `RegInboxApp`.

appSymbol A symbol identifying your application.

UnRegisterViewDef

`UnRegisterViewDef(`*formatSymbol*`,` *classSymbol*`)`

Unregisters a routing format frame from the global view definition registry.

formatSymbol A symbol identifying the routing format you want to unregister. This is the value of the `symbol` slot in the format.

classSymbol A symbol under which the routing format was registered.

For more information about view definitions and the use of this global function, refer to Chapter 5, "Stationery," in *Newton Programmer's Guide: System Software*.

UnRegTheseAppClasses

`UnRegTheseAppClasses(`*appSymbol*`,` *dataClasses*`)`

Unregisters specific classes that an application had previously registered with the function `RegAppClasses`. If all classes registered by an application are unregistered, this function will also unregister the application.

appSymbol A symbol identifying your application.

dataClasses An array of symbols identifying data classes that you want to unregister.

Application-Defined Methods

This section describes methods that are defined in an application to implement particular features.

AutoPutAway

app`:AutoPutAway(`*item*`)`

When an item is received by the In Box, and the In Box can identify an owner application for the item, the In Box sends the base view of the owner application the `AutoPutAway` message. This gives an application the opportunity to immediately receive and do something with an incoming item.

item A frame that is the incoming In Box item.

If the `AutoPutAway` method returns a non-`nil` value, it is assumed that the application handled the item and it may be deleted from the In Box soup. The transport determines what is done with the item in the In Box. For more details, see the section "Automatically Putting Away Items" beginning on page 2-37.

If `nil` is returned, the item is saved in the In Box soup.

PutAwayScript

app:PutAwayScript(*item*)

When the user is viewing an In Box item and taps the Put Away button, and the In Box can identify an owner application for the item, the In Box sends the base view of the owner application the PutAwayScript message. This gives an application the opportunity to do something with the item.

item A frame that is the In Box soup entry. Usually, you will be interested only in the body slot of this frame; other slots contain routing and transport information.

If the PutAwayScript method returns a non-nil value, it is assumed that the application handled the item and it may be deleted from the In Box soup. The transport determines what is done with the item in the In Box. For more details, see the section "Manually Putting Away Items" beginning on page 2-39.

If nil is returned, the item is saved in the In Box soup and an alert is displayed telling the user that the item could not be put away.

If your application defines this method, it should support putting away data of all the classes for which it registered with the RegAppClasses function. If it registers to handle multiple data classes and data of different classes needs to be handled differently, it should check the class of the data it receives.

ItemCompletionScript

app:ItemCompletionScript(*item*)

The In Box sends this message to the base view of an application when the state of a sent item changes or when errors occur while the item is being sent.

item The In Box entry being sent.

This message allows an application to track what happens to a sent item. You can control whether or not this message is sent by setting the completionScript slot in the item frame passed to the Send function, as described on page 2-47.

VerifyRoutingInfo

app:VerifyRoutingInfo(*targetInfo*, *item*)

The system sends this message to the base view of your application when the routing slip is opened. This method gives your application a chance to make modifications to the target object before it is passed to the transport.

targetInfo — A frame, containing `target` and `targetView` slots, as returned by the `GetTargetInfo` function (page 2-70).

item — An item frame, as obtained from the transport method `NewItem` (page 3-68). From this frame you can derive other information you might need, such as the transport name. For more information about the item frame, see the section "Item Frame" beginning on page 3-3.

This method should return *targetInfo*, modified if you want. If you return `nil` from this function, the routing action is canceled without notice to the user. (The `OpenRoutingSlip` function returns `'skipErrorMessage`.)

Note that the `VerifyRoutingInfo` method is executed before the format's `SetupItem` method is executed, so you can make changes to the *targetInfo* frame before it gets passed to `SetupItem`.

Summary of the Routing Interface

Constants

```
ROM_RouteDeleteIcon     // bitmap for delete icon
ROM_RouteDuplicateIcon  // bitmap for duplicate icon
```

Data Structures

Item Frame

```
itemFrame := {
appSymbol: symbol,     // appSymbol of sender
destAppSymbol: symbol, // receiving app, if different
body: frame,           // the data to send
title: string,         // item title, e-mail subject
text: string,          // text of msg, for eWorld
toRef: array,          // array of name refs for recipients
cc: array,    // array of name refs for copied recipients
bcc: array,   // array of name refs for blind copies
fromRef: frame,        // name ref for sender
currentFormat: symbol, // routing format to use
connect: Boolean,      // try to connect immediately?
hidden: Boolean,       // hide UI and hide in Out Box?
covert: Boolean,       // not logged or saved in Out Box?
completionScript: Boolean,   // notify app of state change?
needsResolve: Boolean, // body slot contains an alias?
printer: frame,        // printer frame; the printer to use
coverPage: Boolean,    // use a cover page for fax?
faxResolution: symbol, // 'fine or 'normal fax resolution
phoneNumber: string,   // phone number, for call transport
name: string,          // name, for call transport
serviceProvider: symbol, // 'modem, 'speaker, or nil
saveAsLog: Boolean,    // log call in Calls app?
}
```

RouteScripts Array Element

```
RouteScriptsArrayElement := {
title: string,         // string name of picker item
icon: bitmap object,   // icon for picker item
```

```
RouteScript: symbol,   // func called if this action chosen
GetTitle: function,    // supplied instead of title slot
...                    // other slots used by your app
}
```

Routing Format Frame

```
RoutingFormatFrame := {
_proto: routing proto,   // proto for format
dataTypes: array,        // identifies supported data types
title: string,           // name of format
symbol: symbol,          // unique id - include signature
type: 'printFormat,      // identifies this format as
                         // used for routing
version: integer,        // version number of format
auxForm: viewTemplate,   // defines auxiliary view
storeAlias: Boolean,     // store alias?
showMessage: Boolean,    // warn user when aliasing?
sizeLimit: integer,      // maximum size without aliasing
storeCursors: Boolean,   // store multiple items as a cursor?
usesCursors: Boolean,    // format handles cursor for layout?
orientation: symbol,     // 'portrait or 'landscape
margins: boundsFrame,    // margin insets for layout
SetupItem: function,     // called if this format selected
TextScript: function,    // gets text data from item
TargetSize: function,    // determines target size
MakeBodyAlias: function,   // makes an alias
ResolveBody: function,   // resolves alias body slot
ViewSetupChildrenScript: function,  // set up the children
PrintNextPageScript: function,   // constructs next page
GetCursorFormat: function,       // gets format for an item
FormatInitScript: function,      // initialization
CountPages: function,    // counts pages for a fax
}
```

Protos

protoActionButton

```
aProtoActionButton := {
_proto: protoActionButton,
viewBounds : boundsFrame,
viewJustify: justificationFlags,
viewFormat: formatFlags,
...
}
```

protoPrinterChooserButton

```
aPrinterChooserButton := {
_proto: protoPrinterChooserButton,
viewBounds : boundsFrame,
}
```

protoRoutingFormat, protoPrintFormat, and protoFrameFormat

```
aFormat := {
_proto: protoRoutingFormat, // or one of the other protos
type: 'printFormat,     // don't change this
title: string,          // name of format
symbol: symbol,         // unique id - include signature
dataTypes: ['frame, 'text],// supports frame & text data
version: integer,       // version number
auxForm: viewTemplate,  // for auxiliary view
storeAlias: Boolean,    // store alias?
showMessage: Boolean,   // warn user when aliasing?
sizeLimit: integer,     // maximum size without aliasing
storeCursors: Boolean, // store cursor to multiple items?
SetupItem: function,    // puts target into item frame
```

```
TextScript: function,      // gets text data
TargetSize: function,      // determines target size
MakeBodyAlias: function,   // makes an alias
ResolveBody: function,     // resolves alias body slot

// for protoPrintFormat variant only
dataTypes: ['view],  // print formats support view data
usesCursors: Boolean,  // handles multiple items on a page?
orientation: symbol,       // 'portrait or 'orientation
margins: boundsFrame,      // inset from edges
pageWidth: integer,        // width of view
pageHeight: integer,       // height of view
ViewSetupChildrenScript: function,  // set up the children
PrintNextPageScript: function,      // for multiple pages
GetCursorFormat: function, //returns format for next item
FormatInitScript: function,         // initialization
CountPages: function,      // counts pages for a fax
...
}
```

Functions and Methods

Send-Related Functions and Methods

`Send(`*transportSym*`, `*item*`)`
`GetRouteFormats(`*item*`)`
`GetFormatTransports(`*formatArray*`, `*target*`)`
view`:GetDefaultFormat(`*transport*`, `*target*`)`
view`:SetDefaultFormat(`*transport*`, `*target*`, `*format*`)`
`OpenRoutingSlip(`*item*`, `*targetInfo*`)`

Cursor-Related Functions

`CreateTargetCursor(`*class*`, `*dataArray*`)`
`GetTargetCursor(`*target*`, `*param2*`)`
`TargetIsCursor(`*target*`)`

Utility Functions and Methods

`AppInstalled(`*appSymbol*`)`
`ClassAppByClass(`*dataClass*`)`
view`:GetActiveView()`
`GetItemTransport(`*item*`)`
view`:GetRouteScripts()`
view`:GetTargetInfo(`*reason*`)`
`RegAppClasses(`*appSymbol*`, `*dataClasses*`)`
`RegInboxApp(`*appSymbol*`, `*test*`)`
`RegisterViewDef(`*formatFrame*`, `*classSymbol*`)`
`TransportNotify(`*transport*`, `*message*`, `*paramArray*`)`
`UnRegAppClasses(`*appSymbol*`)`
`UnRegInboxApp(`*appSymbol*`)`
`UnRegisterViewDef(`*formatSymbol*`, `*classSymbol*`)`
`UnRegTheseAppClasses(`*appSymbol*`, `*dataClasses*`)`

Application-Defined Methods

app`:AutoPutAway(`*item*`)`
app`:PutAwayScript(`*item*`)`
app`:ItemCompletionScript(`*item*`)`
app`:VerifyRoutingInfo(`*targetInfo*`, `*item*`)`

Transport Interface

This chapter describes the Transport interface in Newton system software. The Transport interface allows you to provide a new communication service to the system.

You should read this chapter if you are writing a low-level communication tool or special endpoint that you want to make available as a transport for applications to use. If you are writing only an application, you need only use the Routing interface described in Chapter 2, "Routing Interface."

This chapter describes how to

- create a new transport and make it available to the system
- create a routing information template, for use by the In/Out Box
- control the built-in status templates, if you need to provide status information to the user
- create a routing slip template, if your transport sends data
- create a transport preferences template, if your transport has user-configurable options

About Transports

A **transport** is a NewtonScript object that provides a communication service to the Newton In/Out Box. It interfaces between the In/Out Box and an endpoint (see Figure 1-1 on page 1-2), moving data between the two. This chapter describes the transport object and its interface to the In/Out Box.

Applications interact with transports through the Routing interface and the In/Out Box. The In/Out Box is the bridge between applications and transports, without either knowing about the other.

In the user interface, most transports are visible as items in the Action picker menu. The transports available in the picker are not specified directly by an application, but consist of all the transports found that can handle the kind of data the application routes. Because this menu is constructed dynamically, applications can take advantage of additional transports that might be installed in the system at any time. An application need not know anything about the transports available. Likewise, transports can be removed from the system without any effects on applications.

Transport Parts

In writing a transport, you will need to provide the following parts:

- the transport object itself, created from `protoTransport`
- an optional routing information template for the In/Out Box, created from `protoTransportHeader`
- an optional status template for displaying status information to the user, created from `protoStatusTemplate` (if you don't provide one, the default, `statusTemplate`, is used)
- a routing slip template for obtaining routing information from the user, created from `protoFullRouteSlip` (this is needed only for transports that send data)

- a preferences template for user-configuration settings, created from `protoTransportPrefs` (this is needed only for transports that have user-configurable options that you want to store as preferences)

Item Frame

Anything sent or received through the In/Out Box by a transport is passed as a single frame. The frame can contain any number of slots. Some slots are required, and others are optional. The item is treated like a tagged file format where each slot has a symbol (tag) that can be examined. Some slots (`body`) have meaning only to the application that created the item, other slots have meaning only to the In/Out Box itself, and other slots are for the transport. You should ignore any slot you don't recognize, since it may be used internally.

These are some of the slots in the item frame that you need to know about:

`timestamp`	The time this item was submitted to the In/Out Box. The transport shouldn't change this value.
`category`	The `appSymbol` of the transport doing the transfer.
`appSymbol`	A symbol identifying the owner application. If this symbol is missing, or the application cannot be located, the `class` slot inside the `body` frame is used to find an application that can put away the item, for an incoming item.
`destAppSymbol`	A symbol identifying the application to receive the item, if it is different from the sending application. This is set automatically; the transport shouldn't change this value.
`body`	The frame being sent (or references to the data). This is supplied by the application.
`title`	The string that shows up in the In/Out Box as the item's title. The transport may provide this for received items that do not contain a data definition.
`remote`	A Boolean that is set to `true` to identify an item whose body is stored remotely. The transport must set this slot if it downloads just the title of an item but leaves the body stored remotely. When the user tries to view the

	item, the In Box alerts the transport to download the body of the item from the remote host by sending it the `ReceiveRequest` message.
`connect`	A Boolean used for items to be sent. This slot is set to `true` if the user chose to send the item immediately with the Send button in the routing slip. If the user chose to send the item later, this slot is set to `nil`.
`error`	An integer error code; non-`nil` indicates an error. This is usually set by `ItemCompleted`.
`currentFormat`	A symbol identifying the selected format for this item.
`hidden`	Boolean; if `true`, the item is not displayed in the In/Out Box.If set to true, the `completionScript` slot must also be set to `true` and the application must have an `ItemCompletionScript` method.
`covert`	Boolean; if `true`, the item is not logged or saved.
`state`	A symbol indicating status: `'ready`, `'sent`, `'received`, `'read`, `'remote`, `'pending`, or `'logged`. This is usually set by `ItemCompleted`. Do not set this slot directly.
`completionScript`	Boolean; if `true`, the transport sends the `ItemCompletionScript` message (page 3-94) to the application when the item's state changes or when errors occur. For more details on this mechanism, see the section "Completion and Logging" beginning on page 3-17.
`needsResolve`	A Boolean that is set to `true` if the `body` slot contains an alias, rather than the actual data.

For transports that need addressing information, such information is usually encapsulated in name references. 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 it includes associated view definitions that can display the information stored in or referenced by a name reference. For more information about how name references are used for addressing information, see the section "Creating a Name Reference" beginning on

page 2-34. For more information about name references in general, see the documentation of the `protoListPicker` in Chapter 6, "Pickers, Pop-up Views, and Overviews," in *Newton Programmer's Guide: System Software*.

The following two slots in the item frame define the recipient and sender addresses:

`toRef`	An array containing one or more name references used to identify the recipient(s) of the item.
`fromRef`	A name reference or other information that identifies the sender. This information is usually extracted from the sender's current owner card, or persona. The transport normally sets this slot in its `NewItem` method. For more information, see the section "Obtaining an Item Frame" beginning on page 3-14.

In addition, there may be other address slots used by some transports. For example, the eWorld transport uses the slots `cc` and `bcc` to hold additional name references for copy and blind copy recipients.

For a detailed description of all the item frame slots that are important to the Routing interface, see the section "Item Frame" on page 2-44.

Using the Transport Interface

This section describes how to use the Transport interface to perform these specific tasks:

- create a new transport object
- create a routing information template, for use by the In/Out Box
- control the built-in status template, if you need to provide status information to the user
- create a routing slip template, if your transport sends data
- create a transport preferences template, if your transport has user-configurable options

Providing a Transport Object

To make a new transport object, you create a frame with a prototype of `protoTransport`.

Transports are not the same as applications, they are built as auto parts. This means that when installed, they add their services to the system but do not add an application to the Extras Drawer. (They are represented by an icon in the Extras Drawer, but you can't tap it to open it like you can an application icon.)

For a complete description of the `protoTransport` object, see the section "protoTransport" beginning on page 3-43.

The following subsections describe operations that a transport can perform, and the methods that you must supply or call in your transport object to support these operations.

Installing the Transport

To install a new transport in the system, call the `RegTransport` function (page 3-90) from the `InstallScript` method of your package and pass it the transport `appSymbol` and transport template. The `RegTransport` function additionally sends your transport object the `InstallScript` message (page 3-62); this message is unrelated to the `InstallScript` message used by packages. The `InstallScript` message sent to your transport simply provides the opportunity for the transport to perform initialization operations when it is installed.

When your transport is removed, you can use the `UnRegTransport` function (page 3-91) to unregister the transport from the system. You pass the `UnRegTransport` function the transport `appSymbol`.

When your transport is scrubbed by the user from the Extras Drawer, the system also calls the `DeletionScript` function in its package. In the `DeletionScript` function, you should call the `DeleteTransport` function (page 3-91). This function removes user configuration information related to the transport.

Setting the Address Class

The transport object contains a slot, `addressingClass`, that holds a symbol. This symbol identifies the class of the address information used by the transport, such as that stored in the `toRef` and `fromRef` slots of an item. (See the section "Item Frame" beginning on page 3-3.) The In/Out Box uses this symbol to look up and display the to and from address information based on soup entries (usually from the Names soup).

The class of address information is defined by name reference data definitions registered in the system. You can specify one of the following built-in name reference data definitions in the `addressingClass` slot:

- `'|nameRef.email|`, for use with a transport that handles e-mail
- `'|nameRef.fax|`, for use with a transport that handles fax phone calls
- `'|nameRef.phone|`, for use with a transport that handles other phone calls

Or you can specify a custom name reference data definition that you have created and registered with the system. Note that all name reference data definitions must be registered under a symbol that is a subclass of `'nameref`.

The default setting of the `addressingClass` slot is the symbol `'|nameRef.email|`.

For more information about how name references are used for addressing information, see the section "Creating a Name Reference" beginning on page 2-34. For more information about name references in general, see the documentation of the `protoListPicker` in Chapter 6, "Pickers, Pop-up Views, and Overviews," in *Newton Programmer's Guide: System Software*.

Grouping Transports

Two or more transports can be grouped together in the Action picker under a single action. For example, there might be several different e-mail transports grouped together under the single action "Mail." The user selects a particular e-mail transport from a picker that is supplied by the system in the routing slip, if there are multiple transports registered for that group. (The picker doesn't appear if there is only one transport in the group.)

Each group of transports is identified by a common symbol, called the group symbol. You indicate that your transport should be a member of a group by specifying its group symbol in the `group` slot (page 3-44), its title in the `groupTitle` slot, and its icon in the `groupIcon` slot. All transports in the same group should specify the same group icon. This icon is shown in the Action picker for that transport group. The individual transport icon (specified in the `icon` slot) is shown in the routing slip when the transport is selected from the transport group picker.

You can use the following built-in bitmaps in the `groupIcon` slot of your transport, if it belongs to one of the predefined groups. Here are the magic pointer constants:

Group	Icon bitmap constant
`'mail`	`ROM_RouteMailIcon`
`'print`	`ROM_RoutePrintIcon`
`'fax`	`ROM_RouteFaxIcon`
`'beam`	`ROM_RouteBeamIcon`

After the user chooses a particular transport in a group from the picker in the routing slip, the system remembers the last choice and sets the routing slip to that choice when the user later chooses the same routing action from the Action picker. If the user changes the particular transport in the routing slip group picker, 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 (page 3-82). This allows the routing slip to take any necessary action such as alerting the user that information might be lost as a result of changing transports. If `TransportChanged` returns a non-`nil` value, the transport is not changed and the routing slip is not closed. If `TransportChanged` returns `nil`, then the transport is changed and operations continue normally.

You can use the function `GetGroupTransport` (page 3-93) to determine the name of the current (last-used) transport in a group. Note that when you first install a group transport, it becomes the current transport for that group.

Transports that are part of a group are individually selectable on the Send, Receive, and Preferences pickers in the In/Out Box.

Sending Data

The Out Box sends the `SendRequest` message to your transport when data needs to be sent. If your transport supports sending data, you must define this method to actually send the data. For a complete description of the `SendRequest` method, see the section "SendRequest" beginning on page 3-72.

The Out Box will put its own query information in the *request* frame argument to `SendRequest`. Your `SendRequest` method must pass it back to the Out Box in an `ItemRequest` message to get the item (or next item) to send. In your `SendRequest` method, keep calling `ItemRequest` until it returns `nil`, signalling no more items to send. For a complete description of the `ItemRequest` method, see the section "ItemRequest" beginning on page 3-65.

You can choose to comply with or ignore any request to send, depending on the communication resources available and their status. If you choose to comply, your `SendRequest` method must obtain one or more items from the Out Box (using the `ItemRequest` method) and send them by whatever means the transport uses to communicate. For example, many transports use the endpoint interface to establish and operate a connection.

If *request*.`cause` is `'submit`, the item is queued in the Out Box for later sending, but the Out Box still notifies the transport by sending it this `SendRequest` message. Typically, a transport doesn't need to take any immediate action on items where *request*.`cause` is `'submit`; so you can use code like this to simply return:

```
If request.cause = 'submit then return nil;
```

If you encounter an error in your `SendRequest` method, you must call `ItemCompleted` to inform the In/Out Box that an item was not sent. `ItemCompleted` uses `HandleError` to inform the user of an error. If you

want to perform your own error notification, you can override the `HandleError` method.

Sending All Items

If your transport is able to establish a connection, and you want to take advantage of it to send all queued items from the Out Box, you can send your transport the message `CheckOutbox` (page 3-53). This method is defined in `protoTransport` and it causes the In/Out Box to send your transport a `SendRequest` for all queued items waiting to be sent. The `SendRequest` message sent back to your transport includes a *request* argument in which the `cause` slot is set to `'user`.

Applications can also send the `CheckOutbox` message directly to transports by using the `TransportNotify` global function.

Converting an E-Mail Address to an Internet Address

If you are implementing a new e-mail transport that communicates with another e-mail system, you may need to convert e-mail addresses from that system to internet-compatible addresses. The transport method `NormalizeAddress` (page 3-68) allows you to do this. You pass it a name reference containing an e-mail address, and it returns a string containing an internet-compatible e-mail address.

To register a new e-mail system so that it shows up on e-mail pickers throughout the system and to register a conversion for an internet address, use the function `RegEmailSystem` (page 3-91).

Receiving Data

The Out Box sends the `ReceiveRequest` message to your transport to request the transport to receive items. If your transport supports receiving data, you must define this method to receive it. For a complete description of the `ReceiveRequest` method, see the section “ReceiveRequest” beginning on page 3-72.

The `ReceiveRequest` method takes one parameter, a frame containing a `cause` slot whose value is a symbol. The symbol `'remote` may be used by e-mail transports or other transports that don't download all the data initially. For example, upon connection, the transport might download just the titles of messages or other data objects. In this case, the transport should insert a slot called `remote` whose value is `true` in the item frame of each of those items. This slot serves as a flag to tell the In/Out Box that the body of the item is stored remotely and has not yet been downloaded.

When the user attempts to view one of these items in the In Box, the In Box sees the `remote` slot, and sends the transport the `ReceiveRequest` message with the `'remote` cause. This alerts the transport to download the body of the item from the remote host. If multiple remote items were selected by the user for downloading, you must use the `ItemRequest` method to retrieve subsequent requested items and download them. Keep calling `ItemRequest` until it return `nil`, which signals that there are no more items to retrieve.

Some transports may ignore the `ReceiveRequest` message since they update all the time. Others may use this message as a trigger to initiate a connection.

You can choose to comply with or ignore any request to receive, depending on the communication resources available and their status. If you choose to comply, the `ReceiveRequest` method should establish a connection and begin receiving by whatever means the transport uses to communicate. For example, many transports use the endpoint interface to establish and operate a connection. After receiving the item, you should call the transport method `NewFromItem` (page 3-67) to copy it into a standard item frame used by the In/Out Box.

If your transport creates virtual binary objects, you must use the method `GetDefaultOwnerStore` (page 3-54) to determine on which store to create them.

Note

Every received item must have a `class` slot to identify its data class. Items received from other Newtons always have a `class` slot. For items received from other systems, your transport must assign a meaningful class to each item. ◆

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 (if your transport is non-modal). 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` (page 3-71) to queue up requests for sending or receiving, if the transport already has an active communication session in progress. You call `QueueRequest` from the `SendRequest` or `ReceiveRequest` method, whichever one you receive as a result of a user request.

By the way you call it, you can make `QueueRequest` append the new request to a request in progress, or you can make `QueueRequest` start another connection when the current request is finished. To append the new request to a request 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. For example:

```
// you receive an initial send request from the system
// which you process and begin sending
yourTransport:SendRequest func (foo)
...
// while in progress, you receive another send request
// in which you call QueueRequest
yourTransport:SendRequest func (bar)
begin
   if status <> 'idle then // check if I'm active
```

```
      :QueueRequest(foo, bar);//append to current request
   else
      // do a normal send here
   end,
```

When a new request is appended to an in-progress request, items from the new request will be 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 will be another `SendRequest` or `ReceiveRequest` method. Here's an example:

```
// you receive an initial send request from the system
// which you process and begin sending
yourTransport:SendRequest func (foo)
...
// while in progress, you receive another send request
// in which you call QueueRequest
yourTransport:SendRequest func (bar)
begin
   if status <> 'idle then // check if I'm active
      :QueueRequest('SendRequest, bar); // wait for idle
   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 should be defined by all transports. This method is described in the section "CancelRequest" on page 3-51.

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

The `CancelRequest` method should return `true` if it is ok to turn off power immediately after this method returns. This method should return `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 this `CancelRequest` method. When you receive the callback message from your endpoint cancel request confirming that the cancel is done, you then 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. For a complete description of the `NewItem` method, see the section "NewItem" beginning on page 3-68.

This method is supplied by the `protoTransport`, but should be overridden by your transport to fill in extra values used by your transport. If you do 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 3-3.

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. Note that you shouldn't just use the value of `GetUserConfig('currentPersona)` because it is simply an alias to a names file entry. You must 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 just 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
fromRef := datadef:GetRoutingInfo(fromRef);
// returns frame like this:
[{name: "Chris Smith", email: "cbsmith@apple.test.com"}]
```

The `GetRoutingInfo` method returns a frame with at least a `name` slot containing a string. Depending on the `addressingClass` slot passed to the `GetDataDefs` function, the returned frame will also contain 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`, or you can add the slots you want to the `fromRef` frame by extracting them from the original name reference by using the `Get` method, like this:

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

If, instead of extracting the address and sending it as a string, your transport sends addressing information as a frame, like the beam transport, then 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`, like this:

```
// 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.

For more information about name references and the methods of name reference data definitions, see the section "Creating a Name Reference" beginning on page 2-34, and the documentation of `protoListPicker` in Chapter 6, "Pickers, Pop-up Views, and Overviews," in *Newton Programmer's Guide: System Software*.

Here is an example of how you would 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 (page 3-67) 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.

Completion and Logging

After your transport has completed processing an item (either sending or receiving, with or without errors), you must send the transport the message `ItemCompleted` (page 3-63). This method must be used when an item is altered in any way. This method 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.

Be sure to 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 could 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` only sends the `ItemCompletionScript` callback message to the application 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.

To perform logging, `ItemCompleted` sends your transport the message `MakeLogEntry` (page 3-66), 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` method is called to get the title.

Storing Transport Preferences and Configuration Information

Transports can store user-configurable preferences and other configuration information in the system `userConfiguration` entry. This entry in the System soup stores all kinds of system information and user preferences. Configuration information for a transport is stored in a slot in this entry that is named with the transport `appSymbol`. Typically, you store a frame containing slots 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` (page 3-54) and `SetConfig` (page 3-73) to retrieve and set slots in the frame for your transport.

The default preferences for a transport are set by the `defaultConfiguration` slot (page 3-47) in the transport object. This slot holds a frame containing values that correspond to items in a preferences slip that you might want to use for your transport to allow the user to set preferences. For more information about displaying a preferences slip to the user so they can set user preferences, see the section "Providing a Preferences Template" beginning on page 3-39.

You don't have to use this preferences dialog or the setting of the `defaultConfiguration` slot in `protoTransport`. You can override this 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` pointer to the default frame since the contents of the `defaultConfiguration` slot is stored in a soup and `_proto` pointers can't be stored in soup entries.

Extending the In/Out Box Interface

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

The In/Out Box Action picker includes only the Put Away and Log items by default. You can add other transport-dependent items by implementing the `GetTransportScripts` method (page 3-59). For example, the picker shown above includes Reply and Forward items that were added by an e-mail transport to allow the user to perform those operations on e-mail directly in the In/Out Box.

When the user taps on the Tag button, the system sends your transport the `GetTransportScripts` message, if you've implemented it. This method should return an array of frames that describe new items to be added to the In/Out Box Action picker. The array is exactly the same as the `routeScripts` array that is used to add 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
routeScript: 'MyReplyFunc, // called if action selected
},
{title: "Forward", // name of action
icon: ROM_RouteForward, // picker icon
routeScript: 'MyForwardFunc, // called if action selected
} ]
```

The `routeScript` slot contains a symbol identifying a method in the transport base frame that is called if the user selects that item from the picker.

For more detailed information about the items in the array, see the section "Providing Application-Specific Routing Actions" beginning on page 2-27.

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 file, `ROM_RouteAddSender`
- copy text to the Notepad, `ROM_RoutePasteText`

If you are adding one of these actions, you should 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 (page 3-52), if you've implemented it. This method allows your transport to add a put away option for the item to the Put Away picker.

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 (page 3-62). This hook allows your transport to add functionality to items in the In/Out Box by modifying the list of view definitions available for an item. For example, the mail transport adds a text viewer view definition to its entries in the In/Out Box.

Application Messages

Applications can send messages directly to a single transport or to all transports by using the `TransportNotify` global function (page 2-73). This mechanism is provided as a general way for applications to communicate with transports. Here's 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 (page 3-50) notifies the transport that an application has opened and is interested in data from the application. The In/Out Box sends this message to all transports when it opens. 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 `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. 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, for example. Another use might be for an application to notify a transport that automatically makes a connection whenever the application is open.

The `AppClosed` message (page 3-49) 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, you should keep track of how many times it's received the `AppOpened` message and not actually disconnect until it receives the same number of `AppClosed` messages.

The `AppInFront` message (page 3-50) 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.

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.

Error Handling

The default exception handling method implemented by `protoTransport` is `HandleThrow` (page 3-61). This method will catch and handle 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` (page 3-61) to give your transport a chance to screen out benign errors. If `IgnoreError` returns `true`, then `HandleThrow` returns `nil` and stops.

Assuming the error is not rejected by `IgnoreError`, `HandleThrow` next checks if an item is currently being processed. If it is, it sends your transport the `ItemCompleted` message (page 3-63) and then returns `true`. Note that `ItemCompleted` calls `HandleError` (page 3-60) 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` message calls `TranslateError` (page 3-75) 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 for you. This power-off handler is registered 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 for 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 (page 3-51) and waits for the transport to become idle before allowing the power to be turned off.

You can override the default `PowerOffCheck` method if you wish. For details, see the `PowerOffCheck` method on page 3-70.

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 was active.

Providing a Status Template

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

Probably you will not need to create your own status template. The `protoTransport` is defined with a default status template named `statusTemplate` (based on `protoStatusTemplate`). The `statusTemplate` includes six different predefined subtypes, described in Table 3-1 and shown in Figure 3-1. Each of the predefined subtypes consists of a set of child views that are added to the base status view. The base status view includes just a transport icon and a close box, to which different child views are added, depending on the specified subtype name.

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 of the subtypes is also shown in Table 3-1. This information is necessary for use in the `SetStatusDialog` method.

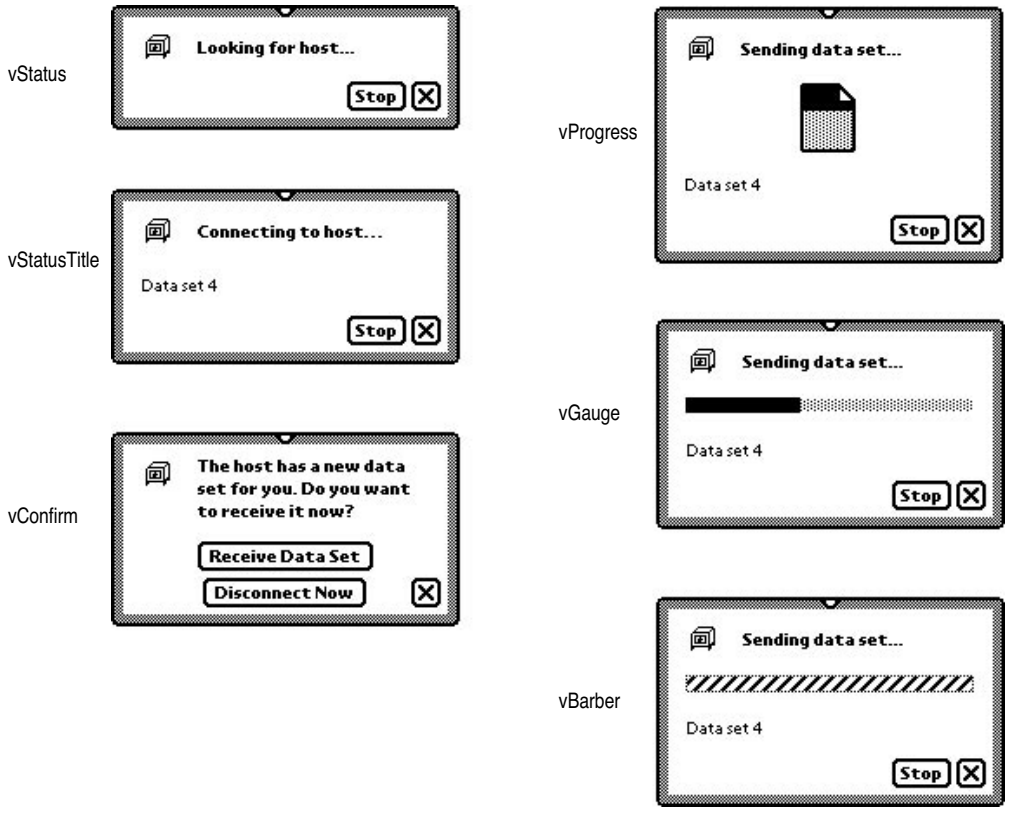
Table 3-1 Status view subtypes

Subtype name	Important values	Description
`vStatus`	`statusText` (top string)	A view that simply 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.

Table 3-1 Status view subtypes

Subtype name	Important values	Description
`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 3-1 Status view subtypes



A transport can specify that one of these subtypes be used in the status view either by setting the transport `viewStatus` slot (for example, `viewStatus := 'vProgress`), or by passing the subtype name in the *name* parameter to the `SetStatusDialog` transport method (page 3-74). 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).

By using this set of predefined status templates, all transports will have a similar user interface and will match the use of other status views throughout the system.

For more detailed information on `protoStatusTemplate` and the various predefined subtypes, refer to Chapter 16, "Additional System Services," in *Newton Programmer's Guide: System Software*.

Controlling the Status View

Your transport should display a status view to the user whenever it is engaged in some 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, you use the transport method `SetStatusDialog` (page 3-74). If the `autoStatus` slot of the transport preferences frame is `true`, the status view will automatically be opened when you first 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 will not be opened 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 (page 3-74) 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 (page 3-45), 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 you can specify 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 16, "Additional System Services," in *Newton Programmer's Guide: System Software*.

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 each of the predefined status subtypes are listed in the "Important values" column in Table 3-1.

Here are some examples of how you'd use the `SetStatusDialog` method to set the different status subtypes to create the status views shown in Figure 3-1:

```
// 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() your function here},
primary:{text:"Disconnect Now", script: func() your function here}});

// 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` to do so. This method of `protoStatusTemplate` simply 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, here's how you would use `UpdateIndicator` to update the `vGauge` subtype:

```
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 like it's moving, you would use the `UpdateIndicator` method in a `ViewIdleScript` method, like this:

```
// 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()
```

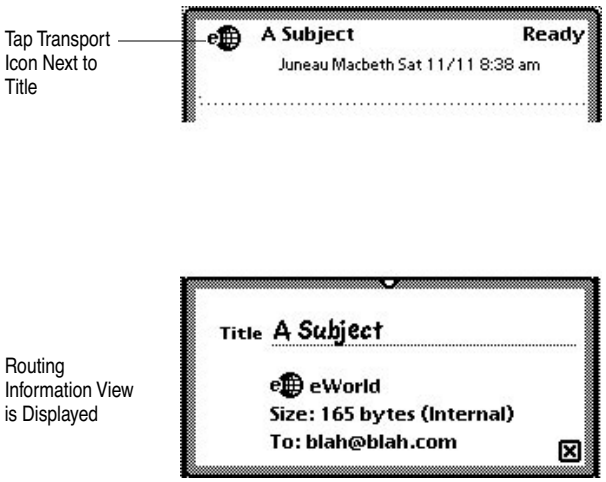
```
   begin
   :UpdateIndicator(barberValueFrame); // spin the barber
   return 500; // idle for 0.5 seconds
   end;
...
// start the idler
statusDialog:setupidle(500)
```

If the `autoStatus` slot of the transport preferences frame is `true`, the status view will automatically be closed when you send the `SetStatusDialog` message with `'idle` as the first parameter. If `autoStatus` is `nil`, the status view will not have been opened in the first place.

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 will reopen.

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, and for a mail item, the e-mail header is shown. Here is an example:



You should create a template for a routing information view for your transport, using `protoTransportHeader` (see page 3-77). If you don't specify a header view, your transport will use the default view, which displays the item title, the transport icon and name, and the item's size in the In/Out Box soup (these are the first three elements in the picture above).

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 provide a `BuildText` method (page 3-77). From your `BuildText` method, you call the `AddText` method (page 3-77) for each additional line of text you want to add below the existing elements. Alternatively, you can simply add child views to the routing information 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 (see page 3-78). This message is sent to whatever object you designate when the header view is closed.

Providing a Routing Slip Template

A routing slip is used by a transport when sending an item. The purpose of a routing slip is 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.

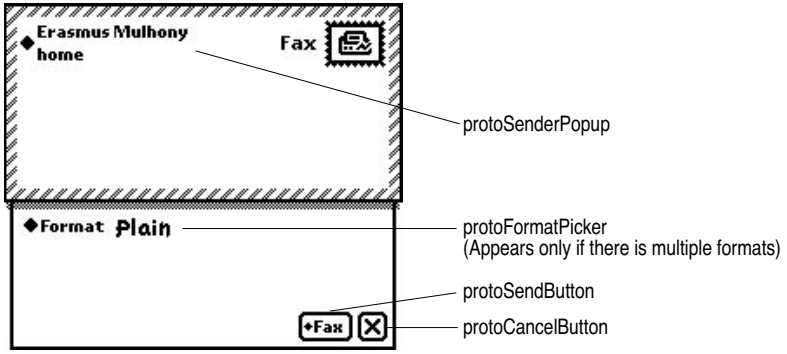
In your transport object, store a reference to your routing slip template in the `routingSlip` slot.

You use the `protoFullRouteSlip` template to create a routing slip. This proto is described in the following section.

Two additional protos that you might want to use in routing slips are also described in the following sections: `protoAddressPicker` and `protoSenderPopup`.

Using protoFullRouteSlip

This routing slip proto already includes most of the elements required in a routing slip. For a complete description of this proto, see the section "protoFullRouteSlip" beginning on page 3-78. Here is an example:



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 `formatPicker` child in `protoFullRouteSlip` provides the picker list for choosing among multiple formats. It is based on `protoFormatPicker`. The current format is initially displayed. This proto provides for opening an auxiliary view if one is associated with the current format. This proto uses the `currentFormat` slot in the item (the `fields.currentFormat` slot in the routing slip) and the `formats` array and `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 `lastFormats` frame in the application is updated, and `currentFormat` in the item is updated. This proto also sends the `SetupItem` message to the format itself. If the format contains an `auxForm` slot, the view specified in the `auxForm` slot is opened when the format is selected.

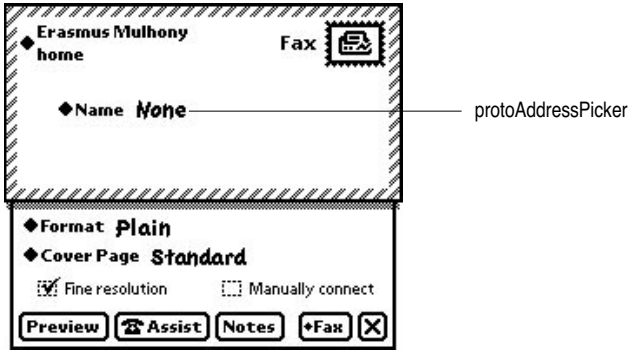
The `sendButton` child in `protoFullRouteSlip` provides the button that actually sends the item to the Out Box and can also activate the transport. It is based on `protoSendButton`. When tapped, the button may display a picker with the choices "Now" and "Later," or it may immediately send the item now or later. It's operation depends on the user preference setting of the `nowOrLater` slot in the preferences configuration frame described in Table 3-2 on page 3-48, and on the return value of the transport `ConnectionDetect` method (page 3-53), 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 (`allowBodyCursors` transport slot is `nil`), the system sends the transport the `VerifyRoutingInfo` method (page 3-76). This method allows the transport to modify each of the individual items, if necessary.

The general function of the Send button is to submit the content 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.) If "Now" is selected, the button also sets the `connect` slot to `true` in the item, which the transport can choose to act on by connecting immediately.

The name of the current transport is shown 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 (page 3-82). This allows the routing slip to take any necessary action such as alerting the user that information might be lost as a result of changing transports. For more information on grouped transports, see the section "Grouping Transports" beginning on page 3-7.

Besides the supplied elements, your transport needs to add additional elements to the routing slip view that are transport specific. 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. The following illustration shows what a complete routing slip might look like, after you add transport-specific items:



In the middle of the envelope portion of your routing slip template, you typically include a view that gathers and displays routing or address 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 3-36.

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 only included 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` (page 3-80). 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. Any other elements you add should be positioned sibling-relative to it.

Using Owner Information

The `protoFullRouteSlip` view sends the `OwnerInfoChanged` callback method (page 3-81) to itself if the user changes the selection of owner name or worksite location in the `protoSenderPopup` view. The `OwnerInfoChanged` method provides you a 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 3-14.

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, any data you use from the owner card should be obtained from the current persona at the time the item is queued by the user.

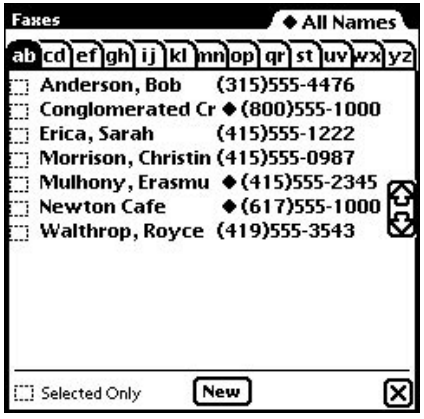
Such information might include the sender's name, return address, credit card information, and so on.

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 possibly 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 provides a picker list that you can use in the routing slip to allow the user to choose the recipient(s) of the item being sent.

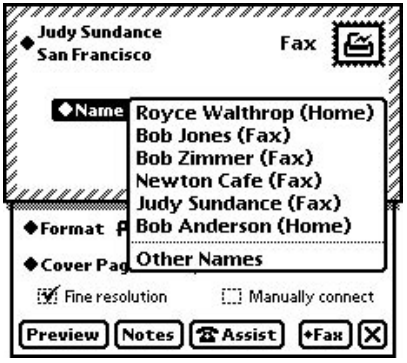
The first time ever that the user taps on the address picker, it opens a view that displays a list of names from the Names file, from which the user can choose one or more recipients.



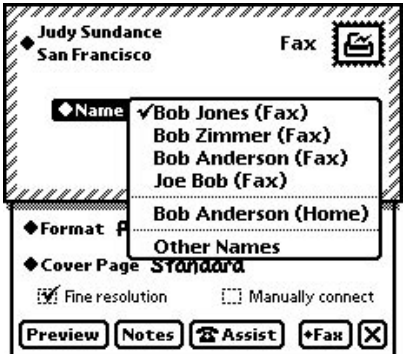
This view uses the `protoPeoplePicker` to provide the name picking facility. The address picker is customizable so that you can substitute a different name picking service other than `protoPeoplePicker` by setting

the `_picker` slot. For example, an e-mail transport might want to use this facility to provide an alternate directory service.

Once the user picks a name, this information is saved, and the next time the address picker opens, it displays a small picker listing this saved name and the choice "Other . . . ". The user can choose "Other . . ." 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. The address picker remembers the last eight names selected.



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. It might look like this:



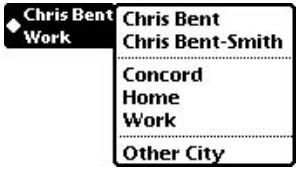
For a complete description of `protoAddressPicker`, see the section "protoAddressPicker" beginning on page 3-84.

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 it includes 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, refer to the `protoListPicker` documentation in Chapter 6, "Pickers, Pop-up Views, and Overviews," in *Newton Programmer's Guide: System Software*.

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` will contain 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` (page 3-57) to do the same thing.

Using protoSenderPopup

This proto is a child of `protoFullRouteSlip` and provides the picker view in the upper left corner of the routing slip for choosing the appropriate owner and worksite to identify the sender of the item.



This proto is documented here in case you want to use it in a custom routing slip of some kind.

This picker allows the sender of the item to select a different owner persona or worksite, which might affect how the owner's name and address appear and how the item is sent. For example, if you choose a home worksite which has a different area code from your work, and you are sending a fax to work, the system will automatically insert a 1 and the work area code before the phone number, which it wouldn't do if you told the system you were located in the work 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.

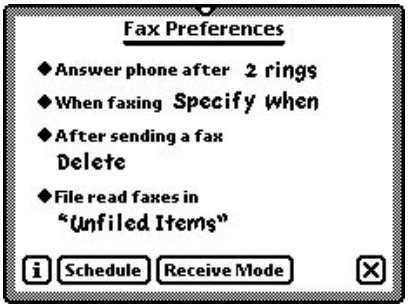
There are no slots you need to set to use `protoSenderPopup`. Simply create a view based on this proto and include it as a child of your routing slip.

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 that has a preferences view is listed in the information

menu, as shown in this example:

Information Button Menu



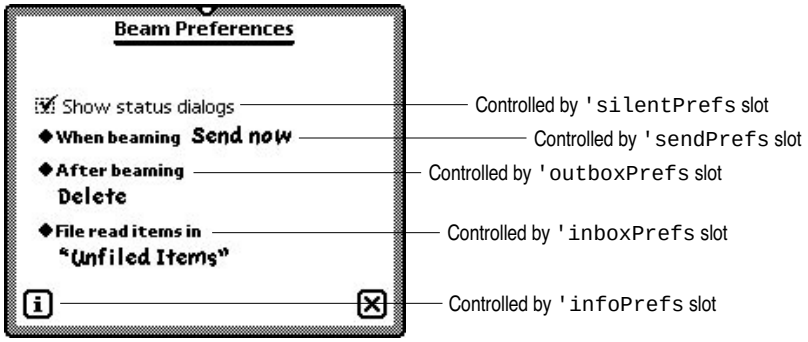
Example Preferences View

To make a preferences view for a transport, you create a template with a prototype of `protoTransportPrefs` (see page 3-86). In your transport object, store a reference to your preferences view template in the `preferencesForm` slot. When the information menu is displayed, a menu item is automatically included 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 here:

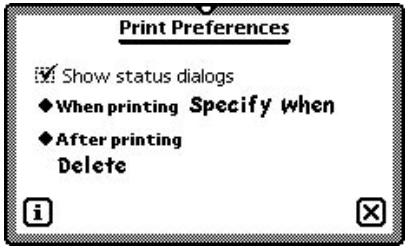


You can selectively remove any of the elements shown above by setting the corresponding slot to `nil` in the `protoTransportPrefs` view. If you want to include additional items in your preferences view, you can add child views to the `protoTransportPrefs` view. The default child elements positioned in 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, simple increase the height of the view and add your elements above those that already exist, except for the title.

Another feature of the `protoTransportPrefs` template is that it automatically checks your transport and displays or hides the In Box and Out Box preference elements. If your transport does not contain a `SendRequest` method, then the Out Box preference element is not displayed. If your transport does not contain a `ReceiveRequest` method, then the In Box preference element is not displayed. If the latter element is not included, 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

is illustrated here:



The Info button is included in the `protoTransportPrefs` template so that you can give the user access to About and Help views for the transport. The Info button is built from the standard `protoInfoButton` proto. To include particular 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 simply 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 also add your own custom items to the Info picker. To do this, you must supply `GenInfoAuxItems` and `DoInfoAux` methods in the `infoPrefs` frame. For more information about these methods and how the Info button works, see the documentation for `protoInfoButton` in Chapter 7, "Controls and Other Protos," in *Newton Programmer's Guide: System Software 2.0*.

The `defaultConfiguration` slot (page 3-47) 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'll have to construct an identical one with different values. You can't use a `_proto` pointer to the default frame since the contents of the `defaultConfiguration` slot are stored in a soup and `_proto` pointers can't be stored in soup entries.

Transport Interface Reference

This section describes the protos and routines provided by the Transport interface.

Protos

This section describes transport-related protos.

protoTransport

This is the basic transport object.

Slot descriptions

`appSymbol`	Required. A symbol used to identify this transport in In/Out Box soup items and in the global `transports` array. This symbol must be unique, so it is recommended that you append your registered developer signature.
`title`	Required. A string that is the name used to identify this transport to the user.
`dataTypes`	Required. An array of symbols representing routing types supported by this transport. The currently defined types in the system include `'view`, `'frame`, `'text`, and `'binary`. Other types may be defined, but only those applications aware of them can use them. You can omit this slot if your transport will not appear in the Action picker.
`actionTitle`	Optional. A string that is the name of the routing action to take place. If you don't provide this slot, the default is "Send". This string is displayed in the routing slip next to the stamp in the upper-right corner, and is used for the text on the send button.

`icon` Optional. The bitmap frame for an icon used for this transport (in the Action picker and In/Out Box), as returned by the NTK picture slot editor or the `GetPictAsBits` function. If this slot is not specified, the default icon is the one used for the Mail action. If your transport is a member of a group, the `groupIcon` slot specifies the icon to use in the Action picker, and the `icon` slots specifies the icon to use in the routing slip.

`group` Optional. A symbol specifying which transport group the transport belongs to, if it belongs to a group. The following group symbols are defined internally: `'mail`, `'print`, `'fax`, and `'beam`.

`groupTitle` Optional. A string that is the name to be displayed for this transport when it is shown in a transport group picker in the routing slip. The strings corresponding to the built-in transport groups include: “Mail”, “Print”, “Fax”, and “Beam”.

`groupIcon` Optional. The bitmap frame for an icon used for this transport group (in the Action picker and In/Out Box), as returned by the NTK picture slot editor or the `GetPictAsBits` function. All transports in the same group should specify the same icon. If you specify this slot, then you can include a unique icon for your transport in the `icon` slot. The following magic pointer constants reference built-in bitmaps that are for use with the built-in transport groups: `ROM_RouteMailIcon`, `ROM_RoutePrintIcon`, `ROM_RouteFaxIcon`, `ROM_RouteBeamIcon`.

`routingSlip` Required for transports that send data. A template for the routing slip. See the section “Providing a Routing Slip Template” beginning on page 3-32. If you don’t specify this slot, the default is `protoFullRouteSlip`.

`transportInfoForm` Optional. A template for the routing information view displayed in the In/Out Box. This template is created using `protoTransportHeader`. If you don’t include this slot, you will get the default template that shows

the item title, transport, and item size. See the section "Providing a Routing Information Template" beginning on page 3-30.

`preferencesForm` Optional. A template to use for creating a preferences view for this transport. This template should be based on `protoTransportPrefs`. Transport preferences are accessed from the Info button in the In/Out Box. If you don't specify this slot, the default is `protoTransportPrefs`. See the section "Providing a Preferences Template" beginning on page 3-39.

`statusTemplate` Optional. A template for the status dialog, based on `protoStatusTemplate`. You use the method `SetStatusDialog` to manipulate the contents of the status dialog.

`statusDialog` A reference to the status dialog view (an instantiated `statusTemplate`). When this view is not open, this slot is `nil`, obviously.

`viewStatus` The particular type of status subview to use when opening or updating the status dialog. For instance, if `viewStatus` is `'vProgress`, the `vProgress` subview is used the next time `SetStatusDialog` is called. For more information on status subviews, see the section "Providing a Status Template" beginning on page 3-23.

`modalStatus` Optional, a Boolean. `True` means that you want your status dialogs to be modal, meaning that they won't include a close box. The default is `nil`, meaning that status dialogs are non-modal and do include a close box.

`dialogStatusMsgs`
Optional. A frame containing the status to status-string mappings. Your transport can override this if it wants different status to status-string mappings. This is the default frame:

```
{Idle: "",
Connecting: "Connecting…",
```

```
Sending: "Sending…",
Receiving: "Receiving…",
Confirming: "Confirming…",
Disconnecting: "Disconnecting…",
Canceling: "Canceling…",
Listening: "Listening…",}
```

`itemStateMsgs` Optional. A frame containing the item status to progress-string mappings. Your transport can override this if you want a different set of strings. (You may also add items to this frame, but do not remove any.) This is the default frame:

```
{Received: "New",
Read: "Read",
Ready: "Ready",
Sent: "Sent",
InLog: "Logged",
OutLog: "Logged",
Pending: "Pending",
Remote: "Remote",
Error: "Error"}
```

`status` A symbol that identifies the current state of the transport. Do not set this slot, only read it. The possible values correspond to the slot names in the `dialogStatusMsgs` frame.

`addressingClass`

A symbol specifying the class of the address information in the `toRef` and `fromRef` slots of an item. A name reference data definition for this class must be registered in the system. The default is `'|nameRef.email|`. The In/Out Box uses this to display the to/from address information. For more information, see the section "Setting the Address Class" beginning on page 3-7.

`addressSymbols` An array of symbols identifying e-mail classes that do not need to be translated for use by this transport. For more information on how this slot is used, see the `NormalizeAddress` method (page 3-68).

`allowBodyCursors`

A Boolean value that indicates if the transport can handle a cursor object in the `body` slot of an item in a send request. If the transport can parse and handle a cursor, set this slot to `true`. Otherwise, set this slot to `nil`, and the In/Out Box will never send the transport a cursor object in the body slot; it will always parse the cursor ahead of time and send the transport multiple send requests—one for each item.

`defaultConfiguration`

A frame holding values representing the initial user preferences for the transport. The default value of this slot is the frame described in Table 3-2. If you want to override this frame, you'll have to construct an identical one with different values. You can't use a `_proto` pointer to the default frame since this slot is stored in a soup and `_proto` pointers aren't stored in soup entries. Note that the transport preferences view based on `protoTransportPrefs` interacts with this frame when the user changes preferences.

IMPORTANT

Never set the `defaultConfiguration` slot to `nil`. ▲

Table 3-2 Preferences slots

Slot	Description
`autoStatus`	A boolean. `True` means that you want the `protoTransport` to open and close the status dialog based on the transport's status. `Nil` means that the status slip stays hidden. This slot corresponds to the "Show status dialogs" preferences check box. The default setting is `true`.
`inboxLogging`	One of the values `'save`, `'log`, or `nil`. This value determines what's done with an entry after the received item has been put away. The value `'save` means the item is saved in the In Box; `'log` means the item is deleted from the In Box and a log entry is made; and `nil` means the item is deleted from the In Box. The default is `nil`. See "ItemCompleted" beginning on page 3-63.
`outboxLogging`	One of the values `'save`, `'log`, or `nil`. This value determines what's done with an entry after the send completes successfully. The value `'save` means the item is saved in the Out Box; `'log` means the item is deleted from the Out Box and a log entry is made; and `nil` means the item is deleted from the Out Box. The default is `nil`. See "ItemCompleted" beginning on page 3-63.

Table 3-2 Preferences slots

Slot	Description
`inboxFiling`	A symbol indicating in which In Box folder an item is to be filed when it is received. Specify a symbol representing a folder name, or `nil` to file incoming items in the Untitled folder, or the symbol `'same` to leave the item where it is (note that this is essentially the same effect as `nil`). Note that filing doesn't happen until after the In/Out Box is closed. The default is the symbol `'same`.
`outboxFiling`	A symbol indicating in which Out Box folder an item is to be filed after it is sent. Specify a symbol representing a folder name, or `nil` to file sent items in the Untitled folder, or the symbol `'same` to leave the item where it is. Note that filing doesn't happen until after the In/Out Box is closed. The default is the symbol `'same`.
`noworlater`	A symbol indicating what action the Send button in the routing slip should take when the user taps it. Specify the symbol `'now` to force the button to always send items immediately (corresponds to the "Send now" preferences choice). Specify the symbol `'later` to force the button to always send items later (corresponds to the "Send later" preferences choice). Specify `nil` to force the button to display a picker allowing the user to choose now or later each time (corresponds to the "Specify when" preferences choice). The default is `nil`.

Note that the `translate` slot of `protoTransport` is used internally and is reserved.

The methods that are of interest in `protoTransport` are described in the following subsections, in alphabetical order.

AppClosed

transport`:AppClosed(`*appSymbol*`)`

This message notifies the transport that an application has closed. The In/Out Box sends this message to all transports when it closes.

appSymbol A symbol identifying the application that closed.

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.

For more information about using the `AppClosed` method, see the section "Application Messages" on page 3-20.

AppInFront

transport`:AppInFront(`*inFront*`,` *appSymbol*`)`

This message notifies the transport that an application is no longer the frontmost application or that it is now the frontmost application. The In/Out Box sends this message to all transports when it becomes frontmost or is no longer frontmost.

inFront	A Boolean. This value is set to `true` if the application is now the frontmost application. This value is set to `nil` if the application is no longer the frontmost application.
appSymbol	A symbol identifying the application whose frontmost status has changed.

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.

For more information about using the `AppInFront` method, see the section "Application Messages" beginning on page 3-20.

AppOpened

transport`:AppOpened(`*appSymbol*`)`

This 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.

appSymbol	A symbol identifying the application that opened.

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 `AppOpened` message, you must define this method in your transport.

For more information about using the `AppOpened` method, see the section "Application Messages" on page 3-20.

CancelRequest

transport`:CancelRequest(`*why*`)`

This method should be defined by all transports. This message is sent to the transport to request that the current send or receive operation be canceled.

why A symbol identifying the reason why the transport should cancel the current operation. The following symbols are defined:

`'powerOff` The Newton is powering-off.

`'emergencyPowerOn`
The Newton just turned on after shutting down unexpectedly. That is, the transport was not idle and the power was lost and so the shutdown was not handled cleanly.

`'userCancel`
The user canceled the operation, usually by means of the Stop button in the status slip.

When receiving this message, the transport should terminate the communication operation as soon as possible.

This method should return `true` if it is ok to turn off power immediately after this method returns. This method should return `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.

For more information about using the `CancelRequest` method, see the section "Canceling an Operation" on page 3-13.

CanPutAway

transport:CanPutAway(*item*)

This message is sent to your transport when the user has selected an In/Out Box item and then taps the Tag button in the In/Out Box (the button that looks like a tag). This method allows your transport to add a put away option for the item. If you don't implement this method, the message won't be sent.

item An item frame containing the item the user requested be put away from the In/Out Box.

It there are no predefined put away options for the item (no applications have registered to handle that data class), and you do not add an option using CanPutAway, then the Put Away choice is not included in the In/Out Box Action picker. If there is one option, then the Put Away choice does appear and, if selected, the single option is shown in the Put Away slip. If there are multiple options, then a Put Away picker is displayed in the Put Away slip.

If you want to do nothing with the item, or do not know how to put it away, you can return nil from CanPutAway. In this case, no option is added by this method.

If you want to allow the item to be put away by a particular application that hasn't previously registered to handle data of the item's class, you can return the appSymbol of that application from this method. Then that application will be added as a put away option for the user to choose.

If you want to allow the item to be put away by a particular application (or even by your transport), and you want a different name displayed to the user in the Put Away picker, you can return a frame that looks like this:

```
{appName: string, // app name shown to user
appSymbol: symbol} // appSymbol of app to put away item to
```

The latter option allows your transport to put away the item to itself and do some kind of special handling, while telling the user that the item is being put away to a different application. For example, a transport might want to

convert the item to another data type and then internally call the `PutAwayScript` method of another application.

In any case, the `CanPutAway` method simply adds a put away alternative to those already available to the user for the item. The user can choose any alternative.

CheckOutbox

transport`:CheckOutbox()`

Send this message to cause the In/Out Box to send your transport a `SendRequest` for all queued items waiting to be sent. The `SendRequest` message includes a *request* argument, in which the `cause` slot is set to `'user`.

Do not override this method.

CloseStatusDialog

transport`:CloseStatusDialog(`*fromUser*`)`

Send this message to explicitly close the status dialog.

fromUser	A Boolean that should be set to `true` if the close is a result of the user tapping the status slip close box. When you call this method, you should always set this parameter to `nil`.

Do not override this method.

If you close the status slip programmatically (*fromUser* is `nil`), the next call to `SetStatusDialog` with a status other than `'idle` will reopen the status slip. If the user closes the status slip, it will remain closed for the remainder of the current communication transaction.

ConnectionDetect

transport`:ConnectionDetect()`

This message is sent to a transport when the routing slip is displayed. It provides an opportunity for the transport to control the operation of the Send button in the routing slip.

In most transports, the Send button contains a picker with the choices "Now" and "Later." From this picker, the user can choose whether to send the item immediately or queue it in the In/Out Box for sending later. The default transport preferences interface also allows the user to set a preference for the Send button. The user can make this button always send now, later, or display a picker so the user can choose between now or later.

If you want to force the Send button to send now or later, you can implement the `ConnectionDetect` method. You should return the symbol `'now` or `'later`, to specify when the item should be sent (no picker is displayed). You can also return `nil`, which causes the "Now/Later" picker to be displayed.

The default version of this method implemented in `protoTransport` returns the value stored in the `nowOrLater` slot (page 3-49) from the transport configuration frame, so it simply respects the user preference setting. You can override this method to force a different behavior.

GetConfig

transport`:GetConfig(`*prefName*`)`

Send this message to return a value from the transport preferences stored in the system `userConfiguration` frame.

prefName A symbol identifying a transport preferences slot.

This method fetches the value of the slot identified by *prefName* in the frame identified by the `appSymbol` of the transport in the system `userConfiguration` entry. You can override this method if you want to do something different.

GetDefaultOwnerStore

transport`:GetDefaultOwnerStore()`

Send this message to return the default store for the transport owner application (the In/Out Box). If your transport creates virtual binary objects, you must use this method to determine the store on which to create a virtual binary object.

GetFolderName

transport`:GetFolderName(`*item*`)`

This message is sent to a transport by the In/Out Box when an item's status changes such that it can be filed. This would occur after an item is sent or after it is put away. This function returns the name of the folder where the item should be filed.

item A frame that is the item to file.

This function returns a string or a symbol indicating the folder in which to file the item. The folder returned is based on the user preferences set for the In/Out Box. The default is the current folder (the symbol `'same`).

Note that the item is not actually filed until after the In/Out Box closes. The item appears filed in its new location the next time the In/Out Box opens.

You probably won't need to override this method.

GetFromText

transport`:GetFromText(`*item*`)`

Define this message if you want to override the default method of obtaining a string that represents the sender of the item, for display in the item header in the In Box.

item A frame that is the item from which the system needs to obtain sender information.

When the system is constructing the header information for an item in the In Box, it sends your transport this message to allow you an opportunity to provide a string for the sender. You should return a string representing the sender's name, address, and/or other information.

If you don't define this method, or it returns `nil`, then the system obtains the sender information from the `fromRef` slot of the item, using the `GetRoutingTitle` method of the name reference data definition.

This method is called by `GetItemInfo`.

You need to provide this method only if the default behavior doesn't suit your needs.

GetItemInfo

transport`:GetItemInfo(`*item*`,` *length*`,` *fontInfo*`)`

Send this message to return a string that is used as the second line of information when the item header is displayed.

item	A frame that is the item for which you want to retrieve an informational string.
length	An integer specifying the maximum length, in pixels, of the string that is returned.
fontInfo	A font specification, which is used to determine how many characters of the string will fit in the specified length, so it can be truncated appropriately.

This methods builds a string containing the name of the sender or recipient concatenated with the date and time the item was sent or received. This method calls `GetItemTime` and `GetToText` (for Out Box items) or `GetFromText` (for In Box items) to provide your transport an opportunity to customize the sender or recipient information.

Internally, `GetItemInfo` calls `StyledStrTruncate` to truncate the string returned by these methods.

GetItemStateString

transport`:GetItemStateString(`*item*`)`

Send this message to return the status string based on the state of the specified item. This method fetches the string from the `itemStateMsgs` frame, based on the value of the *item*`.state` slot.

item	A frame that is the item for which you want to retrieve a status string.

GetItemTime

transport`:GetItemTime(`*item*`)`

Define this message if you want to override the default method of obtaining a string that represents the time and date stamp of the item, for display in the item header in the In/Out Box.

item A frame that is the item from which the system needs to obtain time and date information.

If you decide to override this method, you should return a string containing time and date information for the item.

The default method simply extracts the time and date from the `timeStamp` slot in the item frame. You need to provide this method only if the default behavior doesn't suit your needs.

This method is called by `GetItemInfo`.

GetItemTitle

transport`:GetItemTitle(`*item*`)`

Send this message to return a string that is the title of the item. This method gets the string from the item data definition, if one exists, or from the `title` slot in the item. The In/Out Box also calls this method to get a title for the overview and the item view.

item A frame that is the item for which you want to retrieve a title string.

GetNameText

transport`:GetNameText(`*nameRef*`,` *length*`,` *fontInfo*`)`

Send this message to the transport to obtain a string representation of the names contained in one or more name references.

nameRef A name reference or an array of name references.

length	An integer specifying the maximum length, in pixels, of the string that is returned.
fontInfo	A font specification, which is used to determine how many characters of the string will fit in the specified length, so it can be truncated appropriately.

This method returns a string containing the name or names extracted from the name reference, as you would normally see them displayed in the routing slip. If you specify an array for *nameRef*, the returned string contains the names concatenated, with commas between each name. The string is truncated as specified by the *length* and *fontInfo* parameters.

GetStatusString

transport`:GetStatusString()`

Send this message to return the status string based on the current status. This method fetches the string from the `dialogStatusMsgs` frame.

GetTitleInfoShape

transport`:GetTitleInfoShape(`*item*`,` *bounds*`)`

Send this message to return a shape that fills the area of the item header to the right of the transport icon. This shape contains a title that identifies the item, the item's status, and information about the sender or recipient and a time stamp.

item	A frame that is the In/Out Box item.
bounds	A bounds frame describing the area of the item header that the shapes must fit into.

The exact area of the shape is shown shaded here:

The item header appears in both the In/Out Box overview and the individual item view. The `GetTitleInfoShape` method calls `GetItemTitle` and `GetItemInfo` to generate text shapes for the two lines of the default item header. It also calls `GetItemStateString` to obtain the item status string, which is placed at the far right of the view. You can override `GetTitleInfoShape` to do something different, like add special graphics to the header.

GetToText

transport`:GetToText(`*item*`)`

Define this message if you want to override the default method of obtaining a string that represents the recipient of the item, for display in the item header in the Out Box.

item	A frame that is the item from which the system needs to obtain recipient information.

When the system is constructing the header information for an item in the Out Box, it sends your transport this message to allow you an opportunity to provide a string for the recipient. You should return a string representing the recipient's name, address, and/or other information.

If you don't define this method, or it returns `nil`, then the system obtains the recipient information from the `toRef` slot of the item, using the `GetRoutingTitle` method of the name reference data definition.

This method is called by `GetItemInfo`.

You need to provide this method only if the default behavior doesn't suit your needs.

GetTransportScripts

`GetTransportScripts(`*target*`)`

This message is sent to your transport when the user has selected an In/Out Box item and then taps the Tag button in the In/Out Box (the button that looks like a tag). This method allows your transport to add items to the In/

Out Box Action picker that is displayed as a result of the button tap. If you don't implement this method, the message won't be sent.

target The In/Out box entry that is selected. Note that this could consist of a multiple item target object, if multiple items were selected from the In/Out Box overview.

You can use the global functions `TargetIsCursor` (page 2-68) and `GetTargetCursor` (page 2-67) to check if *target* is a multiple item target object and iterate over it.

The `GetTransportScripts` method should return an array of frames that describe new items to be added to the In/Out Box Action picker. The array is exactly the same as the `routeScripts` array that is used to add items to the Action picker in an application. Each frame in the array should include these slots:

`title` A string that is the name of the action you want to add.

`icon` A bitmap that is the icon that appears next to the name in the picker.

`routeScript` A symbol identifying a function that is called if this action is selected by the user. The function should be implemented as a method in the transport base frame. It is passed two parameters, the target item (the In/Out Box entry) and the target view (the view displaying that entry), respectively. Again, note that the target item passed to this function might be a multiple item target object, so the function should be able to handle that.

For more detailed information about the items in the array, see the section "Providing Application-Specific Routing Actions" beginning on page 2-27.

HandleError

transport`:HandleError(`*error*`)`

Translates an error code into an error string and displays an alert to the user with the transport title and the error string.

error An integer error code.

This method calls `TranslateError` to translate the error code and then `Notify` to display the alert. You can override `HandleError` to do your own error handling, if you wish.

This method is called by `HandleThrow` and `ItemCompleted` when errors occur.

HandleThrow

transport`:HandleThrow()`

The default exception handling method for transports. This method catches any exception on standard transport methods. It calls `CurrentException` to obtain the current exception.

This method calls `IgnoreError` to screen out benign errors. If there is an item being processed, `ItemCompleted` is called for the item. Then `HandleError` is called to translate the error code and display an alert to the user.

`HandleThrow` returns `true` if it handled the error (that is, it did not ignore it). This gives the transport a chance to close things down cleanly on an error. `HandleThrow` returns `nil` if it ignored the error.

If you want, you can override the `HandleThrow` method to implement a different way of handling exceptions.

Also note that `HandleThrow` calls some other functions that you can override to modify it's functionality. These include `IgnoreError` and `HandleError`.

IgnoreError

transport`:IgnoreError(`*error*`)`

Allows your transport an opportunity to specify that a particular error is benign, when an error condition occurs.

error An integer error code.

If this method returns `true`, no error alert is displayed; if this method returns `nil`, an error alert is displayed by the `protoTransport`.

This method handles several benign errors. If you want to override it, be sure to call the inherited method first.

This method is called by `HandleThrow` and `ItemCompleted` when errors occur.

InstallScript

transport`:InstallScript(`*symbol*`)`

This message is sent to the transport when it is registered in the system by `RegTransport`. The `InstallScript` method simply provides the transport with the opportunity to perform any initialization operations that might be necessary.

symbol	The transport `appSymbol` that was passed to `RegTransport`.

IOBoxExtensions

transport`:IOBoxExtensions(`*item*`,` *target*`,` *viewDefs*`,` *reserved*`)`

This message is sent to your transport when an item belonging to the transport is displayed in the In/Out Box. This method allows your transport to add functionality to items in the In/Out Box by modifying the list of view definitions available for an item.

item	A frame that is the In/Out Box entry.
target	The stationery frame within *item* (usually the `body` slot).
viewDefs	The array of view definitions found by the system for the current data definition.
reserved	You can ignore the data passed in this parameter.

Your transport can add to or delete from the *viewDefs* array.

If you want to change the view definition to be used by the item, you should return that view definition from this function. If you don't want to change the item's current view definition, return `nil`.

IsInItem

transport`:IsInItem(`*item*`)`

Returns `true` if the *item* is in the In Box (it's been received, read, or logged), or `nil` otherwise.

item An item frame.

IsLogItem

transport`:IsInItem(`*item*`)`

Returns `true` if the *item* has been logged, or `nil` otherwise.

item An item frame.

ItemCompleted

transport`:ItemCompleted(`*item*`,` *state*`,` *error*`)`

Send this message after the transport has completed processing an item, whether that be sending or receiving, with or without errors. This method should be used when an item is altered in any way.

item A frame that is the item sent or received.

state The new state to set for the item. For the state, specify a symbol identifying one of the slot names listed in the `itemStateMsgs` frame (page 3-46). Generally you specify `'sent` for sent items and `'received` for received items. You can specify `nil` to leave the item state unchanged from its current value.

error An error to set for the item. Specify `nil` for no error.

The return value of this method is undefined; do not rely on it.

The `ItemCompleted` method first sets the state and error of the item. Next, if the item's `completionScript` slot is set to `true`, this method sends the `ItemCompletionScript` message (page 3-94) to the base view of the application identified by the item's `appSymbol` slot. The item is passed as a parameter.

If the `completionScript` slot is `nil`, and if the error is not zero and `IgnoreError` returns `nil`, then `ItemCompleted` calls `HandleError` to display an error alert showing the error. Then, for items whose state is `'sent`, `ItemCompleted` writes the updated item entry back to the Out Box soup, or turns the item into a log entry (calls `MakeLogEntry`), or deletes the item from the Out Box, depending on the error conditions and on the setting of the `outboxLogging` slot.

An item whose state is `'pending` is added to the Out Box and is made the active view; that is, the item view is displayed for the user in the Out Box. This is used for replying to a received item. To do a reply to an item, you can simply change the status to `'pending` and call `ItemCompleted` and the item will be created in the Out Box and displayed to the user for editing.

For items with other kinds of status values, the item is written to the In Box soup.

Do not override this method.

ItemDeleted

transport`:ItemDeleted(`*item*`)`

This message is sent to a transport by the In/Out Box just before an item belonging to that transport is deleted from the In/Out Box.

item — The In/Out Box entry to be deleted. This will always be a single item, not a cursor.

If many items are being deleted, this method is called many times in succession.

The return value of this method is ignored.

This method is not implemented in `protoTransport`. If you want to take some action as a result of the item being deleted, you can implement this method to do so, however, you cannot prevent the item from being deleted.

ItemDuplicated

transport`:ItemDuplicated(`*item*`)`

This message is sent to a transport by the In/Out Box just after an item belonging to that transport is duplicated from within the In/Out Box.

item	The duplicate In/Out Box entry. This will always be a single item, not a cursor. You can modify this object to modify the duplicate entry.

If many items are being duplicated, this method is called many times in succession.

The return value of this method is ignored.

This method is not implemented in `protoTransport`. If you want to take some action as a result of the item being duplicated, you can implement this method to do so, however, you cannot prevent the item from being duplicated.

ItemPutAway

transport`:ItemPutAway(`*item*`)`

This message is sent to a transport by the In/Out Box right after an item has been put away by an application. By default, `ItemPutAway` saves the updated item information, or turns the item into a log entry (calls `MakeLogEntry`), or deletes the item (based on the setting of the `inboxLogging` slot). You can override this behavior, though it is usually not necessary.

item	A frame that is the item put away.

ItemRequest

transport`:ItemRequest(`*request*`)`

Send this message to get an item, or the next item in the queue, from the In/Out Box. If there is an item frame to be sent or a remote item to be received, it is returned; otherwise a `nil` return signals the end of the current request.

request Pass the *request* frame received in the `SendRequest` or `ReceiveRequest` message that was sent to the transport.

Do not override this method.

If you have set the `allowBodyCursors` slot in your transport to `true`, then during a send operation this method might return an item whose `body` slot contains a multiple item target object. It's up to the transport to check if the `body` slot contains such an object and resolve the individual items appropriately before sending them. You can use the global functions `TargetIsCursor` (page 2-68) and `GetTargetCursor` (page 2-67) to check for a multiple item target object and iterate over it. This is important because the items in such an object can be aliases, which must be resolved before trying to send them.

If your transport cannot handle body data that consists of multiple items, you must set the `allowBodyCursors` slot to `nil`.

MakeLogEntry

transport`:MakeLogEntry(`*logItem*`,` *item*`)`

This message is sent to your transport by the In/Out Box when `ItemCompleted` is called and a log entry needs to be made. You should override this method to add transport-specific slots to the log entry.

logItem The log entry to which you can add slots. This is already set up with the `appSymbol`, `title`, `error`, and `labels` slots from the *item* frame, as well as the correct new log state in the `state` slot.

item A frame that is the item sent or received.

This method should return the modified *logItem* frame.

The default MakeLogEntry method sets the `title` slot of *logItem* to the value returned by *transport*`:GetItemTitle(`*item*`)`.

MissingTarget

transport`:MissingTarget(`*slip*`)`

This message is sent to the transport when the user requests a routing action and there is no target to be sent. The default operation is to display an alert notifying the user, "Nothing to Send."

If you want, you can override this method to display a different message or to do something different.

slip	The routing slip view object that was created for the item. Note that this view is not open, and won't be opened.

NewFromItem

transport`:NewFromItem(`*item*`)`

Send this message to obtain a new item based on a received item.

item	An item received.

This method returns a new item frame, containing all but a few slots from the *item* parameter.

This method is useful for transports that receive frame data. This method first sends the message *transport*`:NewItem(nil)` to obtain a new item frame. Then it copies all slots from the frame passed in the *item* parameter into the new item frame, except for the following slots: `category`, `connect`, `completionScript`, and `remote`.

If a `destAppSymbol` slot exists in the *item* frame, it is copied to the `appSymbol` slot in the new item frame, and the `appSymbol` slot in the *item* frame is copied to the `fromAppSymbol` slot in the new item frame. In this way, the target application can be set differently from the originating application.

For more information about using the `NewFromItem` method, see the section "Obtaining an Item Frame" beginning on page 3-14.

NewItem

transport:`NewItem(`*context*`)`

Send this message to obtain a new item frame for the In/Out Box. The item frame returned by this method should contain default values for the transport.

context A frame defining the context from which to get the application symbol. During a send operation, the In/Out Box sets this argument to the application base view of the sending application. If *context* is not `nil`, then `NewItem` will set the `item.appSymbol` slot to the `appSymbol` found in *context*.

For more information about using the `NewItem` method, see the section "Obtaining an Item Frame" beginning on page 3-14.

If you override this method, be sure to call the inherited method first, in your version.

NormalizeAddress

transport:`NormalizeAddress(`*nameFrame*`)`

Send this message to convert a Names soup entry or name reference that contains an e-mail address into a string representation of the internet e-mail address.

nameFrame A Names soup entry, a pseudo-entry, or a name reference that contains an `email` slot. A pseudo-entry refers to a simple frame that contains at least an `email` slot, for example: `{name:{first:"Juneau", last:"Macbeth"}, email: "jmacbeth@acompany.com",}`.

Normally, this method returns a string. However, if the value of the `email` slot in *nameFrame* is not a string, then that value is returned, with no conversion.

The class of the `email` slot in *nameFrame* determines how the address is converted, if at all. `NormalizeAddress` uses the `Get` method of the built-in

`'|nameRef.email|` name reference data definition to extract the e-mail address string from the `email` slot.

After extracting the address string, the `NormalizeAddress` method uses the transport slot `addressSymbols` to determine if the e-mail address should be translated or not. If the class of the e-mail address contained in *nameFrame* is listed in the `addressSymbols` slot, then no translation is done—the system assumes that the transport knows how to handle the address as is. The address string is returned exactly as extracted from the *nameFrame*. Only addresses whose classes do not appear in the `addressSymbols` slot are translated.

For an address that is to be translated, the translation is controlled by a frame registered with the system for that class of e-mail address. New classes of e-mail addresses can be registered by the `RegEmailSystem` function (page 3-91). The translation can consist either of appending a string to the given address or of passing it to a function object which returns a translated string. Most of the built-in translations consist simply of appending a string (such as "@eworld.com") to the given address, if it is not already part of the address. After translation, any spaces are removed from the resulting string, before it is returned.

Table 3-3 lists the built-in e-mail classes and the kind of translation that is done for each. If a string is listed as the translation, that string is simply appended to the given e-mail address, if it is not already part of the given address.

Table 3-3 E-mail address translations

E-mail class	Translation done
`string`	"@eworld.com"
`|string.email|`	"@eworld.com"
`|string.email.eworld|`	"@eworld.com"
`|string.email.internet|`	nothing done
`|string.email.aol|`	"@aol.com"

Table 3-3 E-mail address translations

E-mail class	Translation done
`|string.email.mcimail|`	"@mcimail.com"
`|string.email.attmail|`	"@attmail.com"
`|string.email.easylink|`	"@eln.attmail.com"
`|string.email.prodigy|`	"@prodigy.com"
`|string.email.genie|`	"@genie.geis.com"
`|string.email.delphi|`	"@delphi.com"
`|string.email.msn|`	"@msn.com"
`|string.email.interchange|`	"@ichange.com"
`|string.email.radiomail|`	"@radiomail.net"
`|string.email.compuserve|`	Any comma (,) in the address is changed to a period (.). And then the string "@compuserve.com" is appended to the address if it is not already part of it.

PowerOffCheck

transport`:PowerOffCheck(`*why*`)`

The system sends this message to the transport when it wants to power-off and the transport is not in the idle state.

why A symbol indicating why the system is powering off. The values are as follows:

`'user`	The user turned off the unit.
`'idle`	The unit is going to sleep because it has been idle.
`'because`	Reason is unknown.

The default `PowerOffCheck` method displays a modal slip asking the user to confirm that the unit can be turned off. If the user taps OK, the unit is

turned off. If the user taps Cancel, the power-off sequence is canceled. You can override this method if you want different behavior.

If the `PowerOffCheck` method returns `true`, the system power-off sequence proceeds normally. If this method returns `nil`, the power-off sequence is canceled.

For more information about power-off handling, see the section "Power-Off Handling" on page 3-23.

QueueRequest

transport`:QueueRequest(`*doWhat*`,` *newRequest*`)`

Send this message if you want to queue a send or receive request that is made by the user while the transport is already sending or receiving.

doWhat — Either a symbol, or the request frame for a send or receive request already in progress. If you specify a symbol, it must name a transport method that the system will call when the state of the transport returns to idle. It will pass *newRequest* as a parameter to this method. This defers the new request until after the current one finishes and then invokes a new request.

If you specify a request frame, *newRequest* is appended to it, so that the `ItemRequest` method will eventually return items from *newRequest* during the same communication session. The request frame is the frame passed into a previous `SendRequest` or `ReceiveRequest` method.

newRequest — The request frame describing the new request that you want to queue. This is the parameter passed to the `SendRequest` or `ReceiveRequest` method from which you called `QueueRequest`.

For more information about using the `QueueRequest` method, see the section "Handling Requests When the Transport is Active" on page 3-12.

ReceiveRequest

transport`:ReceiveRequest(`*request*`)`

Define this method if receiving is supported by the transport.

The In Box sends this message to request the transport to receive items.

request A frame identifying the cause of the receive request. There is one important slot:

`cause` A symbol indicating the cause of the receive request. The symbol `'user` indicates that the user tapped the Receive button in the In/Out Box. The symbol `'remote` indicates the user has requested that the text of one or more remotely stored messages be retrieved.

Note that if the `cause` slot is set to `'remote`, the user might have requested that multiple remote items be downloaded. In this case, use the `ItemRequest` method to retrieve subsequent requested items and download them.

For more information about using the `ReceiveRequest` method, see the section "Receiving Data" on page 3-10.

SendRequest

transport`:SendRequest(`*request*`)`

Define this method if sending is supported by the transport.

The Out Box requests the transport to send data by sending the transport a `SendRequest` message.

request A frame identifying the data to be transmitted and the cause of the send request. There is one important slot in this frame that you might need:

`cause` A symbol indicating the cause of the send request, as described in Table 3-4.

Table 3-4 Causes of a send request

Symbol	Description of Cause
`'user`	The user selected the item and tapped the Send button in the In/Out Box.
`'item`	The user chose to send the item immediately in the routing slip (the `connect` slot is set to `true`).
`'submit`	The user chose to send the item later in the routing slip.
`'remote`	The user has requested that the text of a remotely stored sent message be retrieved. This could be used in a system in which sent items were stored remotely, to retrieve the text of one of those items.
`'periodic`	The item was sent by a transport as a result of a scheduled action.

Your `SendRequest` method must use the `ItemRequest` method (page 3-65) to get the item (or next item) to send. In your `SendRequest` method, keep calling `ItemRequest` until it returns `nil`, signalling no more items to send.

If you encounter an error in your `SendRequest` method, you must call `ItemCompleted` to inform the In/Out Box that an item was not sent.

For more information about using the `SendRequest` method, see the section "Sending Data" on page 3-9.

SetConfig

transport`:SetConfig(`*prefName*`,` *value*`)`

Send this message to set a value for the transport preferences stored in the system `userConfiguration` frame.

prefName A symbol identifying a transport preferences slot.

value A value to set in the *prefName* slot.

This method sets the value of the slot identified by *prefName* in the frame identified by the `appSymbol` of the transport in the system `userConfiguration` entry. You can override this method if you want to do something different.

SetStatusDialog

transport`:SetStatusDialog(`*newStatus*`,` *name*`,` *values*`)`

Send this message to set the current state of the transport and to display a status view to the user.

newStatus Can be any symbol such as `'Disconnected`, `'Connecting`, `' Connected`, `'Sending`, `'Receiving`, `'Disconnecting`, and `'Listening`. If *status* is `nil`, the status is not modified. This parameter sets the current state of the transport.

name A symbol identifying the status view subtype template to use for determining which child views to add to the status view. This is the value of the `name` slot in the subtype template. For more details on the status subtypes, see the section "Providing a Status Template" beginning on page 3-23. If you specify `nil`, the last symbol used is assumed, or if you haven't called this function before, the default value `'vStatus` is used.

values A string giving the current status message (if that's the only element you're using or changing). Alternately, you can specify a frame of values, one for each subtype child item you want set.

Each child template contains a `name` slot that identifies the name of the important slot that controls the appearance of that child view. You specify a slot in this frame for each child item that you want to set. The name of each slot you specify is the value of the corresponding `name` slot in the child template. The

value of the slot is the value you want to give to that child element.

For example, if a child view of the specified subtype has a `name` slot of `foo` and the `foo` slot in that child template is expected to be a string, then in *values*, you would specify a slot named `foo` whose value was a string. For more details, see the section "Providing a Status Template" beginning on page 3-23.

If you don't pass the string in this parameter, there must be an entry in the `dialogStatusMsgs` frame that corresponds to the *status* symbol, for string display purposes.

The return value of this method is always `nil`.

Do not override this method.

If a status slip is already open when this method is called, it is updated with the new status information (the child views are closed and reopened). If a status slip is not already open, and the `autoStatus` slot of the transport user preferences frame is `true`, and the transport is not idle, this method opens a status slip and sets it as specified.

TranslateError

transport`:TranslateError(`*error*`)`

Allows your transport an opportunity to translate an error code into a string error message, when an error condition occurs.

error An integer error code.

The string equivalent of the error code should be returned. If your transport does not know how to translate the error, call the inherited function to do the translation (for example, `inherited:TranslateError(error)`).

VerifyRoutingInfo

transport:`VerifyRoutingInfo(`*item*`,` *multiItem*`,` *entry*`,` *format*`)`

This message is sent to a transport when a multiple item target object is submitted to the Out Box as a result of the user tapping the Send button in the routing slip. This message is only sent if the transport cannot handle a cursor in the body slot of an item (the transport slot `allowBodyCursors` is set to `nil`). It is sent repeatedly—once for each item in the multiple target object.

This message provides an opportunity for the transport to modify each item within the multiple item target object.

item	An item being submitted for sending, after it has been passed to the `SetupItem` method of the routing format. This will always be a single item from the multiple item target object. You can modify or add slots to this item frame to change the item before it is stored in the Out Box.
multiItem	The original item frame that was submitted for sending and that contains a multiple item target object in its `body` slot.
entry	A resolved entry from *originalItem,* before it was passed to the routing format's `SetupItem` method.
format	The routing format associated with the item.

The return value of this method is ignored.

This method is not implemented in `protoTransport`. If you want to take some action as a result of a multiple item target object being submitted to the Out Box and being broken into its individual items, you can implement this method to do so.

protoTransportHeader

This proto provides a template for the routing information view. For more information about creating a routing information view, see the section "Providing a Routing Information Template" beginning on page 3-30.

Slot descriptions

`transport`	The transport object to which this information view belongs. This is set up automatically by the In/Out Box.
`target`	A reference to the In/Out Box item. This object is found automatically in context.
`context`	Optional. The view to which the `InfoChanged` message should be sent. Defaults to `nil`, meaning the message won't be sent.
`changed`	This slot is set to `true` if the user changes an entry field in the view, otherwise it is set to `nil`.

The `protoTransportHeader` is based on the `newtInfoBox` proto.

The following methods are of interest.

BuildText

headerView`:BuildText()`

Provide this method in your header view in order to add additional lines of text to the header view, below the existing elements. This method is called by the header view, before the view is opened. For each line you want to add, call the `AddText` method, passing in the string for that line.

The return value of the `BuildText` method is not used.

AddText

headerView`:AddText(`*string*`)`

You can call this method of `protoTransportHeader` from your `BuildText` method to construct a line of text, which is added to the header view, below the existing elements.

string A string of text to add to the header.

The string is given the proper font for the header view, and truncated, if necessary, to fit within the header view.

InfoChanged

context:`InfoChanged`(*changed*)

This message is sent to the view identified by the `context` slot in the routing information view (see the slot description above) when the routing information view is closed.

changed The value of the `changed` slot in the routing information view. This will be either `true`, if the user changed a value in the view, or `nil` if nothing was changed.

protoFullRouteSlip

This proto provides a template for a full-featured routing slip view. For more information about creating a routing slip, see the section "Providing a Routing Slip Template" beginning on page 3-32.

The following slots in the routing slip template are set by the system before the routing slip view is opened.

Slot descriptions

`fields` The item frame returned by the transport `NewItem` method. This frame will eventually become the In/Out Box soup entry for the item. Note that `fields.currentFormat` is set to the last routing format used for this transport by this application. The `SetupItem` method of the routing format sets the `fields.body` slot to the *target* object.

`target` The target frame returned by the application's `GetTargetInfo` method. (`GetTargetInfo` is called with the `'routing` symbol as its argument.) This target

	frame is the data being routed from the application (usually the current or selected object). The system looks at the data class of the target object to determine the list of available routing formats, but no other assumptions are made about what `target` contains.
`targetView`	The active view in the application as returned by the application's `GetTargetInfo` method. This is not used by the routing slip, but is defined in case a routing format or auxiliary slip needs to use it.
`formats`	An array of routing format frames that can be used with `target`.
`activeFormat`	The currently selected routing format.
`transport`	The transport object.

You may want to set these other slots in the routing slip template.

Slot descriptions

`viewJustify`	Optional. The default setting is `vjParentFullH + vjParentCenterV`.
`envelopeHeight`	Optional. An integer that specifies the height of the envelope image, in pixels. The default is 115, which you should generally leave as is. It is recommended that you not change this value; if you do, your envelope will have a non-standard look.
`envelopeWidth`	Optional. An integer that specifies the width of the envelope image, in pixels. The default is 230, which you should generally leave as is. It is recommended that you not change this value; if you do, your envelope will have a non-standard look.
`bottomIndent`	Optional. An integer that is the space below the envelope image, in pixels. The default is 40. This leaves space for you to include interface elements specific to

your transport. Note that this space is taken out of the overall height of the routing slip, which is used for both the envelope portion and the other portion below it.

Note that the `ViewSetupFormScript`, `ViewSetupChildrenScript`, `ViewDrawScript`, `ViewHideScript`, and `ViewQuitScript` methods are used internally in `protoFullRouteSlip` and should not be overridden. If you need to use one of these methods, be sure to call the inherited method also.

The following child views are declared to `protoFullRouteSlip`:

- `formatPicker`, the format picker view, which appears only if there's more than one routing format to choose for the item (see the section "protoFormatPicker" beginning on page 3-83)
- `sendButton`, the send button (see the section "protoSendButton" beginning on page 3-83)

The following methods of `protoFullRouteSlip` are of interest.

BottomOfSlip

routingSlip`:BottomOfSlip()`

This method returns the Y coordinate of the bottom of the routing slip—that is, the very bottom of the lower portion of the slip below the envelope image. You must use this method to determine the bottom of the slip so that you can correctly position interface elements in the lower portion of the routing slip. All items in the lower portion of the routing slip must be positioned relative to the bottom of the slip or sibling bottom-relative to the last child of the routing slip proto, which is the Send button.

FormatChanged

routingSlip`:FormatChanged(`*format*`)`

This message is sent to the routing slip view whenever the user chooses a new routing format in the format picker.

format The new routing format chosen by the user.

If you want to receive this message, define a method to handle it.

Usually, you should return `nil` from this method. This allows the format picker to proceed with executing its normal code, which means closing an auxiliary view for the old routing format, if one is open, and then executing the `LabelActionScript` method in the `protoFormatPicker`. The `LabelActionScript` method sets the `currentFormat` slot in the item, calls the routing format's `SetupItem` method, opens an auxiliary view, if one is defined in the routing format, and saves the chosen routing format in the application base view.

If the `FormatChanged` method returns `true`, the default code will not continue executing. The assumption in the latter case is that you've done all the necessary processing in your `FormatChanged` method.

OwnerInfoChanged

routingSlip`:OwnerInfoChanged()`

This message is sent to the routing slip view whenever the sender pop-up view is changed, so you can catch any changes. The sender pop-up view is the sender's name and worksite location, which is shown in the upper-left corner of the envelope.

If your routing slip depends on data in the sender's current owner card or worksite, you should define this method so that you can update addressing or other information when changes occur. For example, you'll probably want to update the `fromRef` slot in the item frame if the owner persona changes. To do that, you must implement this method.

PrepareToSend

routingSlip`:PrepareToSend(`*when*`)`

This message is sent to the routing slip view when the user taps the send button and selects Now or Later from the picker.

when	A symbol, `'Now` or `'Later`, indicating when the user chose to send the item from the Send button picker.

If you want to do anything to the item before it is sent, you must define this method. For example, you might want to validate the entries in the routing slip or check something in the item itself before allowing it to be sent.

Your `PrepareToSend` method should send the message `ContinueSend` to the routing slip view if you want to continue the submission process. If, as a result of your `PrepareToSend` method, you do not want to submit the item to the Out Box, do not send the `ContinueSend` message, and the process will be canceled.

The `PrepareToSend` method is defined in the `protoFullRouteSlip` template. The default version simply sends the `ContinueSend` message to itself to continue the submission process.

ContinueSend

routingSlip`:ContinueSend(`*when*`)`

Send this message to the routing slip view from your `PrepareToSend` method if you want to continue with the process of submitting the item to the Out Box. If you don't want to submit the item, don't send this message.

when	A symbol, `'Now` or `'Later`, indicating when the user chose to send the item from the Send button picker.

TransportChanged

routingSlip`:TransportChanged(`*newSymbol*`)`

This message is sent to the routing slip view if the transport is a member of a group and the user changes the transport to a different member of the group.

newSymbol	The `appSymbol` of the new transport chosen by the user.

This message provides an opportunity for you to take any necessary action such as alerting the user that information might be lost as a result of changing transports. If `TransportChanged` returns a non-`nil` value, the transport is not changed and the routing slip is not closed. If `TransportChanged` returns `nil`, then the transport is changed and the routing slip is closed and reopened.

You don't need to supply this method. If you don't supply it, the message won't be sent.

protoFormatPicker

This proto provides the picker list in the routing slip for choosing among multiple formats. It is documented here because `protoFullRouteSlip` has a declared child view based on this proto, but it is not for general use. In general, you should use the method `FormatChanged` of `protoFullRouteSlip` to determine when the format changes.

The following method is of interest.

LabelActionScript

formatPicker`:LabelActionScript(`*index*`)`

The system sends this message to the format picker view whenever the user chooses a new routing format. You can override this message if you want to be notified when the user changes routing formats.

index	The index, in the array of routing formats, of the new format chosen by the user. The array of routing formats is stored in the `formats` slot in the format picker, so you can obtain the chosen format using the expression `formats[`*index*`]`.

If you override this message, you must call the inherited version, like this:

```
inherited:LabelActionScript(index);
```

protoSendButton

This proto provides the button in the routing slip that actually sends the item to the Out Box and can also activate the transport. It is documented here because `protoFullRouteSlip` has a declared child view based on this proto, but it is not for general use. In general, you should use the method `PrepareToSend` of `protoFullRouteSlip` to determine when the Send button is tapped.

Slot description

`text`	A string that is the button text. It should name the action to be performed (for example, “Mail,” “Print,” “Beam,” and so on).

The following method is of interest.

PickActionScript

sendButton`:PickActionScript(`*index*`)`

The system sends this message to the send button view when the user taps the send button and selects the “Now” or “Later” choice. You can override this message if you want to be notified when the user taps this button.

index	The index, in the array of picker choices, of the choice selected by the user. If the user chooses “Now,” *index* is 0; if the user chooses “Later,” *index* is 1.

If you override this message, you must call the inherited version, like this:

```
inherited:PickActionScript(index);
```

protoAddressPicker

This proto provides a picker list that you can use in the routing slip for choosing an address from the Names file on the Newton. The `protoAddressPicker` is based on the `protoLabelPicker` and `protoPeoplePopup`. For more information on these protos, see Chapter 6, “Pickers, Pop-up Views, and Overviews,” in *Newton Programmer’s Guide: System Software*.

Slot description

`viewBounds`	Set to the size and location you want for the picker.
`text`	A string that is the picker label. The default is “Name”.
`otherText`	A string that is the last item to be shown in the picker, below the separator line. The default is “Other Names”.
`selected`	An array of initially selected name references, or `nil`, if you don’t want any to be selected initially. You will

probably want to set this slot to the `toRef` array in the item frame. When the picker is closed, this array contains the name references selected from the picker.

`alternatives` An array of alternative name references to show in the picker. This is set up by the Intelligent Assistant.

`class` A symbol identifying a data definition for a name reference object. This symbol identifies the type of name reference object to be used in creating the list, and determines what information is displayed in each of the columns of the list. The following name reference data definitions are built into the system:

`'|nameRef.email|`
Lists names and e-mail addresses

`'|nameRef.fax|`
Lists names and fax phone numbers

`'|nameRef.phone|`
Lists names and voice phone numbers

`_picker` A view template defining the picker to display when the user wants to choose from all the available recipients. The default is `protoPeoplePopup`, which provides a name picker based on `protoPeoplePicker`. Setting this slot allows you to substitute an alternative directory service that has the same interface as the `protoPeoplePopup`.

protoSenderPopup

This proto provides the picker list in the routing slip for choosing the name and worksite location of the sender of the item.

No slots or methods are documented because none are required to use this proto. Simply include a view based on this proto in your routing slip. It is automatically positioned in the top left corner of its parent view.

Note that a view based on this proto is automatically included in `protoFullRouteSlip`. You can use the method `OwnerInfoChanged` of `protoFullRouteSlip` to determine when the sender name or worksite in the `protoSenderPopup` view changes.

protoTransportPrefs

This proto provides a template for a preferences view for your transport. This proto is based on the `protoFloater`. For more information about creating a preferences view, see the section "Providing a Preferences Template" beginning on page 3-39.

Slot descriptions

`viewBounds`	The size of the view and location where it should appear.
`title`	Optional. A string that is the title of this transport. This string is displayed as part of the title at the top of the preferences view, if you include it. If you don't include a `title` slot, you must supply an `appSymbol` slot.
`appSymbol`	Optional. The transport `appSymbol`. This symbol is used to look up the transport title, if you don't include a `title` slot.
`silentPrefs`	A frame defining the text of the checkbox that controls whether or not status dialogs are to be shown. This frame has as its default value the slots described in Table 3-5 on page 3-87. If you don't want to include this item in your preferences dialog, set this slot to `nil`.
`sendPrefs`	A frame defining the choices applicable to when an item is sent. This frame has as its default value the slots described in Table 3-6 on page 3-88. If you don't want to include this item in your preferences dialog, set this slot to `nil`.
`outboxPrefs`	A frame defining the preference item applicable to the Out Box. This frame has as its default value the slots described in Table 3-7 on page 3-89. If you don't want to include this item in your preferences dialog, set this slot to `nil`.
`inBoxPrefs`	A frame defining the preference item applicable to the In Box. This frame has as its default value the slots described in Table 3-8 on page 3-90. If you don't want to include this item in your preferences dialog, set this slot to `nil`.

`infoPrefs`	A frame defining functions that handle Info button choices. The default frame defines one method, `DoInfoHelp`, that opens the system help book. This function is called if the user selects the Help item from the Info menu. You may want to define the `DoInfoAbout`, `GenInfoAuxItems`, and `DoInfoAux` methods to include your own items on the Info button menu. For details on all these methods that support the Info button, see the description of `protoInfoButton` in Chapter 7, "Controls and Other Protos," in *Newton Programmer's Guide: System Software*.

The following four tables describe the default frames for the `silentPrefs`, `sendPrefs`, `outboxPrefs`, and `inboxPrefs` slots. If you want to override any of the default slots in a frame, you must specify a new frame with all the slots shown.

Table 3-5 Slots in `silentPrefs` frame

Slot	Description
`text`	A string that is the text shown next to the checkbox. The default value is a localized version of the string, "Show status dialogs".
`configuration`	A symbol identifying the slot in the transport's configuration frame in which this user preference item is stored. The default value is `'autoStatus`.

Table 3-6 Slots in sendPrefs frame

Slot	Description
routeText	A string that is the text labeling the preference item that controls when sending occurs. The default value is a localized version of the string, "When sending".
routeChoices	An array of strings that are to be used for the picker that lists choices. The default array is a localized version of this: `["Send now", "Send later", "Specify when"]`.
routeActions	An array of symbols associated with the elements in the `routeChoices` array. When a choice from the send picker is made, the corresponding symbol from this array is stored in the `routeConfig` slot to identify the user's selection. The default array is `['now, 'later, nil]`.
routeConfig	A symbol identifying the slot in the transport's configuration frame in which the user preference item controlling when an item is sent is stored. The default is `'noworlater`.
testMethod	A symbol identifying a method for which to test in the transport object. If this method is not found in the transport object, then the "When sending" view element is not automatically displayed in the preferences view. The default value is `'SendRequest`.
transport	Specify the `appSymbol` of the transport.

Table 3-7 Slots in `outboxPrefs` frame

Slot	Description
`logText`	A string that is the text labeling the Out Box preference item, which controls logging. The default value is a localized version of the string, "After sending".
`logChoices`	An array of strings that are to be used for the picker that lists logging choices. The default array is a localized version of this: `["File", "Log", "Delete"]`.
`logActions`	An array of symbols associated with the elements in the `logChoices` array. When a choice from the logging picker is made, the corresponding symbol from this array is stored in the `logConfig` slot to identify the user's selection. The default array is `['save, 'log, nil]`.
`logConfig`	A symbol identifying the slot in the transport's configuration frame in which the user preference item controlling Out Box logging is stored. The default is `'outboxLogging`.
`testMethod`	A symbol identifying a method for which to test in the transport object. If this method is not found in the transport object, then the view elements controlling Out Box preferences are not automatically displayed in the preferences view. The default value is `'SendRequest`.
`transport`	Specify the `appSymbol` of the transport.

Table 3-8 Slots in `inboxPrefs` frame

Slot	Description
`logText`	A string that is the text shown next to the In Box preference item, which controls where items are filed after being read. The default value is a localized version of the string, "File read items in".
`fileConfig`	A symbol identifying the slot in the transport's configuration frame in which the user preference item controlling filing is stored. The default is `'inboxFiling`.
`testMethod`	A symbol identifying a method for which to test in the transport object. If this method is not found in the transport object, then the view elements controlling In Box preferences are not automatically displayed in the preferences view. The default value is `'ReceiveRequest`.
`transport`	Specify the `appSymbol` of the transport.

Functions and Methods

Utility Functions

This section describes utility functions used in the Transport interface.

RegTransport

`RegTransport(`*symbol*`, `*transport*`)`

Use this global function to register a new transport in the system. You call `RegTransport` from the `InstallScript` method of your Newton package containing the transport.

symbol The transport `appSymbol`.

transport The transport template. This template must be based on `protoTransport`.

The return value of this function is undefined.

`RegTransport` sends the `InstallScript` message to the transport, if this message is defined in the transport. The `InstallScript` message is simply a hook allowing the transport to do other initialization when it is installed.

UnRegTransport

`UnRegTransport(`*symbol*`)`

Use this global function to unregister a transport from the system. Usually you would call this function from the `RemoveScript` function of your transport package.

symbol The transport `appSymbol` passed to `RegTransport`.

The return value of this function is undefined.

DeleteTransport

`DeleteTransport(`*symbol*`)`

Use this global function to remove transport-related information stored in the system, for example, the user preferences for the transport. Usually you would call this function from the `DeletionScript` function of your transport package.

symbol The transport `appSymbol` passed to `RegTransport`.

The return value of this function is undefined.

Note that the `RemoveScript` function in the transport package will also be called, following the `DeletionScript` function.

RegEmailSystem

`RegEmailSystem(`*classSymbol*`, `*name*`, `*internet*`)`

Registers a new type of e-mail system.

classSymbol A symbol identifying the class of the email system. This symbol must be a subclass of `'|string.email|`. An example is `'|string.email.BobsMailSystem|`.

name A string that is the name of the e-mail system. This name will show up in pickers listing e-mail systems throughout the system (in routing slips, the In/Out Box, and the Names application), so it should be short.

internet Either a string or a function object that converts an e-mail address from this system into an internet address. If you specify a string, it is simply appended to the e-mail address to make an internet address. For example, you might specify "@bobsmail.com".

If you specify a function object, it will be used to convert an e-mail address on this system to an internet address. The function will be passed one parameter, a string holding an e-mail address. It should return another string, the internet address for that e-mail address. For example, for Compuserve, commas in the address are changed to periods and "@compuserve.com" is appended.

The transport method `NormalizeAddress` uses the information registered by the *internet* parameter to create internet e-mail addresses from system-specific addresses.

Note that none of the arguments to this function are copied into memory by `EnsureInternal`, so care should be taken to ensure that the application which registers the e-mail service can be removed without causing errors.

To unregister an e-mail system that was registered by `RegEmailSystem`, use the function `UnRegEmailSystem`.

Note
This function may not be defined in ROM—
it may be supplied by NTK. ◆

UnRegEmailSystem

`UnRegEmailSystem(`*classSymbol*`)`

Unregisters an e-mail system registered by `RegEmailSystem`.

classSymbol A symbol identifying the class of the e-mail system to unregister. This is the same symbol you passed to `RegEmailSystem` to register the system.

Note that this function can't be used to unregister e-mail systems that are built-in.

Note

This function may not be defined in ROM—
it may be supplied by NTK. ◆

GetCurrentFormat

`GetCurrentFormat(`*item*`)`

Returns the routing format frame (not the format symbol) for an item from the In Box soup or the Out Box soup, or returns `nil` if a routing format cannot be found for the item.

item The In/Out Box item whose routing format you want to get.

GetGroupTransport

`GetGroupTransport(`*groupSymbol*`)`

Returns a symbol identifying the current (last-used) transport within a transport group. If the current transport is no longer available, this function returns a different one from the same group, if there is one. If there is no current transport and none in the group can be found, `nil` is returned.

groupSymbol A symbol identifying a transport group. The following group symbols are defined: `'print`, `'fax`, `'beam`, and `'mail`.

Application-Defined Method

This section describes a method that can be defined in an application to implement a particular feature.

ItemCompletionScript

app`:ItemCompletionScript(`*item*`)`

This message is sent to the base view of an application when an item's state changes.

item The In/Out Box item whose state changed.

This message is sent only if the `completionScript` slot in the item frame is set to `true`. So if you want to take advantage of this callback mechanism, you must set the `completionScript` slot.

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, // application 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
viewStatus: symbol, // next type of status view to open
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 I/O Box actions
HandleError: function, // displays error alert
HandleThrow: function, // handles exceptions
IgnoreError: function, // screens errors
InstallScript: function, // notification of installation
IOBoxExtensions: function, // extends I/O Box view defs
IsInItem: function, // is item in the In/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 new frame data item frame
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
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
targetView: view, // view containing target
formats: array, // array of routing formats for target
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
...
}
```

protoFormatPicker

```
aFormatPicker := {
_proto: protoFormatPicker, // format picker
labelActionScript: function, // notifies view of new choice
...
}
```

protoSendButton

```
aSendButton := {
_proto: protoSendButton, // proto send button
text: string, // button text
PickActionScript: function, // notifies view of user choice
...
}
```

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 all addresses
...
}
```

protoSenderPopup

```
aSenderPopup := {
_proto: protoSenderPopup, // sender popup/picker
// no other slots needed
}
```

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*`)`
`RegEmailSystem(`*classSymbol*`, `*name*`, `*internet*`)`
`UnRegEmailSystem(`*classSymbol*`)`
`GetCurrentFormat(`*item*`)`
`GetGroupTransport(`*groupSymbol*`)`

Application-Defined Method

app`:ItemCompletionScript(`*item*`)`

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

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 underlying endpoints is allocated from the operating system domain, whereas memory for the endpoints 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" on page 4-35.)

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 the section "Output Spec Frame" on page 4-36.

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 linear 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. 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.
- Force termination of the pending input spec, and 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 numbers.
- 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. 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 Table 4-1.

Table 4-1 Data forms

Data form	Description
`'char`	For receiving data or getting endpoint options, the data is converted to Unicode using the endpoint `encoding` slot and returned as a character. For sending data, the character is converted from Unicode using the `encoding` slot. This is the default data form for sending characters with the `Output` method.
`'number`	For receiving data or getting endpoint options, the data is interpreted as a 30-bit integer using 4 bytes. For sending data or setting endpoint options, the high-order 30 bits are placed into 4 bytes. This is the default data form for sending numbers with the `Output` method.

continued

Table 4-1 Data forms (continued)

Data form	Description
`'string`	For receiving data or getting endpoint options, the data is converted to Unicode using the endpoint `encoding` slot and returned as a NewtonScript character string. The conversion process adds the necessary termination byte to the end of the NewtonScript string; the input stream need not have terminators. For sending data, the NewtonScript character string is converted from Unicode using the `encoding` slot. The termination byte is not sent. This is the default data form for sending strings with the `Output` method, and for receiving data.
`'bytes`	For receiving data or getting endpoint options, the data is returned as an array of unsigned single-byte values. For sending data, the value is squeezed into a single unsigned byte and truncated if necessary.
`'binary`	Use this form anytime you want to receive or send raw binary data. This is the default form for sending binary data. (For details, see the section "Working With Binary Data" beginning on page 4-27.)
`'template`	Use this data form to exchange data with a service that expects C-type data. This is the default value for setting endpoint options. (For more information on how to define this type of data, see the section "Template Data Form" beginning on page 4-8.)
`'frame`	The data is expected to be a frame. For output, the frame is flattened into a stream of bytes prior to being sent, and for input, the byte stream is unflattened and returned as a frame. If you want to flatten and unflatten frames independently of sending and receiving data, you can use the global function `Translate`, described on page 4-57.

Only a subset of data form values is applicable for any particular operation. Table 4-2 enumerates the data forms and their applicability to output specs, input specs, and endpoint option frames.

Table 4-2 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 4-19.

The data types that can be used in the `typelist` array are described in Table 4-3.

Table 4-3 Data types for `typelist` array

Data type	Description
`'long`	signed long integer
`'ulong`	unsigned long integer
`'short`	16-bit unsigned short integer
`'byte`	8-bit unsigned byte
`'char`	8-bit character (translated to/from Unicode)
`'unicodechar`	16-bit Unicode character
`'boolean`	8-bit plain Boolean value
`'struct`	an aggregate structure, padded to a long word (4 bytes)
`'array`	an aggregate array

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 data types given in Table 4-3. And *integer* is an integer specifying the number of bytes to convert. To convert an entire string, including the terminator, specify zero for *integer*. A nonzero value specifies the exact number of bytes 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 may be different in other systems and the byte order may be different as well.

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

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 one of the endpoint methods. 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` (page 4-56) 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 can use the global functions `MakePhoneOption` (page 4-57) and `MakeAppleTalkOption` (page 4-56) to construct `'address` options for the modem and AppleTalk tools.

The slots in an endpoint option frame are described in detail in the section "Endpoint Option Frame" beginning on page 4-33.

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 Table 4-10 on page 4-63. 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 6, "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 5, "Built-in Communication Tools."

Initialization and Termination

Before using an endpoint, you must instantiate it using the `Instantiate` method (page 4-44). This method allocates memory in the system and creates the endpoint object. Then, you must bind the endpoint object to a communication tool by calling the `Bind` method (page 4-44). 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 (page 4-45), then dispose of it using the `Dispose` method (page 4-45).

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 (page 4-45). 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 can use the global function `MakePhoneOption` (page 4-57) to return a proper phone number `'address` option. The AppleTalk tool requires an AppleTalk Name Binding Protocol (NBP) `'address` option. You can use the global function `MakeAppleTalkOption` (page 4-56) to return a proper NBP `'address` option.

To establish a connection where you expect someone else to initiate the connection, you may need to call the `Listen` method (page 4-46). Once the connection is made by using `Listen`, you need to call the `Accept` method (page 4-47) to accept the connection, or the `Disconnect` method (page 4-47) to reject the connection and disconnect.

Sending Data

To send data, use the `Output` method (page 4-48). 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 (page 4-36), 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 (see Table 4-12 on page 4-64) 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`, or you can just send an empty packet last, with the flags `kPacket+kEOP` set.

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 4-4 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 the section "Input Spec Frame" beginning on page 4-37.

After you've connected or accepted a connection, you set up your first input spec by calling `SetInputSpec` (page 4-49). When one input spec terminates, the system automatically posts another input spec for you when the `InputScript` method (page 4-38) 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.

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 (page 4-50) before calling `SetInputSpec` with your new spec.

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

Table 4-4 Input spec slot applicability

Data form	target slot	termination slot	discardAfter slot	filter slot	partialFrequency & partialScript 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

Specifying the Data Form and Target

You can choose how you want the received data formatted by setting the `form` slot (page 4-37) in the input spec. In this slot, you specify one of the standard data forms described in Table 4-1 on page 4-6.

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 (page 4-40) 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," in *Newton Programmer's Guide: System Software*.

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 append the received data to 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.

Specifying Data Termination Conditions

For `'string` and `'bytes` data forms, you must indicate when the input terminates by specifying a `termination` slot (page 4-41). 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 set to 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.

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 (page 4-43). Numbers are converted to single byte; 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 (page 4-37) 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 (page 4-37) 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 (page 4-37) 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 (page 4-40) is sent to the input spec frame.

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 (page 4-42) 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 either a single frame or an array of 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 (page 4-43). 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 (page 4-38) 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` (page 4-49) within your `InputScript` method.

Handling Normal Termination of Input

The `InputScript` message (page 4-38) 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`.

Periodically Sampling Incoming Data

You can sample the incoming data without meeting any of the termination conditions by specifying a `PartialScript` method (page 4-40) 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 (page 4-38) in the input spec. 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 (page 4-50).

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 (page 4-40) 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 (page 4-52).

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 immediately terminate a pending input spec and force the system to send it the `InputScript` message by calling the `Input` method (page 4-49). Note that this method is appropriate to use only when receiving data of the forms `'string` and `'bytes`.

You can look at incoming data outside the scope of your `InputScript` or `PartialScript` method by calling the method `Partial` (page 4-50). 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`.

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` (page 4-53). The purpose of this streaming endpoint is to provide a way to send and receive large frames without having to first flatten or unflatten them. The 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.

The `protoStreamingEndpoint` proto includes a method, `StreamIn` (page 4-53), that allows you to receive streamed data. This method automatically unflattens received data into a frame object in memory or directly on a store. Another method, `StreamOut` (page 4-55), 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 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 4-19.

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 (page 4-36). 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` (page 4-57), 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` (page 4-50) or `Disconnect` (page 4-47). 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 (page 4-50) asynchronously, or calling the `Disconnect` method (page 4-47) 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 (page 4-50) synchronously, or calling the `Disconnect` method (page 4-47) 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 (page 4-51) 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 (page 4-53).

Custom communication tools can return special events to the endpoint object through the `EventHandler` message (page 4-52). This message is sent to the endpoint whenever an event occurs that is not handled by one of the usual endpoint event handlers. You can use this mechanism in whatever way you want to pass events from a custom communication tool up to the endpoint layer.

Error Handling

You can handle exception conditions by specifying an `ExceptionHandler` method (page 4-52) in your endpoint. You can cancel all outstanding requests and throw exceptions by calling the `Cancel` method (page 4-50).

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.

Error codes generated by the Endpoint interface are defined in Table 4-9 on page 4-61.

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 Table 4-10 on page 4-63. 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 4-11.

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 the chapter "New System Services" in *Newton Programmer's Guide: System Software 2.0*.

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.

Endpoint Interface Reference

This section describes the data structures your application uses when interacting with the Endpoint interface, and the routines and protos provided by the interface. It also describes the global functions that are useful in endpoint communications.

Data Structures

This section describes the data structures that your application uses to interact with the Endpoint interface.

Endpoint Option Frame

An endpoint option frame selects the communication tool to use, controls its configuration and operation, and returns result code information from each endpoint method call. Note that multiple option frames can be specified together in an array, but arrays cannot be nested.

Slot descriptions

`type`	The type of option, which can be `'service`, `'option`, or `'address`.
`label`	The option identifier, which is dependent on the communication tool. Usually it is a four-character string that identifies the option. Constants are defined for all the different options for the built-in communication tools. For details, see Chapter 5, “Built-in Communication Tools.”

`opCode` A constant indicating how the tool should handle the option request. Possible values include the following:

- `opSetRequired` Indicates that the request must fail if the service is unable to honor it (for example, setting a bps rate of 1.2 million). Note that other options in the options array are processed even though one or more may fail.
- `opSetNegotiate` Indicates that the service can substitute a "reasonable" value if the requested value is unacceptable.
- `opGetDefault` Indicates that the system is to return the default settings.
- `opGetCurrent` Indicates that the system is to return the current settings.

`form` A symbol identifying the data form to be used in interpreting the `data` slot. Data form symbols are listed in Table 4-1 on page 4-6. You should specify the value `'template` for options.

`result` A result code, set on return from the method. (This slot is ignored if used as a parameter.) The possible result codes are listed in Table 4-10 on page 4-63.

`data` The option data. All the built-in communication tools expect data of the `'template` form. This consists of a frame containing `arglist` and `typelist` arrays. For more information on the template data form, see the section "Template Data Form" beginning on page 4-8.

Callback Spec Frame

A callback spec frame controls whether an endpoint method executes synchronously or asynchronously. It also defines a time-out and contains a `CompletionScript` method that is called when the endpoint operation completes.

Slot descriptions

`async` A Boolean value. If `true`, then the request is posted asynchronously. This slot is optional and defaults to `nil`. It is evaluated only at the time the endpoint method is called.

`reqTimeout` An integer specifying the time, in milliseconds, that the system should allow for the request to complete. If a time-out expires for an asynchronous request, that request and *all* outstanding requests are canceled. This slot is optional, defaults to `kNoTimeout`, and is evaluated only at the time the method is called. This slot is ignored if the callback spec is used with the `Cancel` method, since time-outs don't apply to `Cancel`.

The following method is also defined in a callback spec frame.

CompletionScript

`CompletionScript(`*endpoint*`, `*options*`, `*result*`)`

The `CompletionScript` message is sent to a callback spec frame when an asynchronous request completes.

endpoint The endpoint associated with the request.

options A frame containing the returned options for those requests that support the *options* parameter.

result The result code. If no error occurred, this parameter is set to `nil`.

The `CompletionScript` method's return value is not used.

The `CompletionScript` slot in a callback spec is evaluated every time the `CompletionScript` message is to be sent.

Output Spec Frame

An output spec frame is simply a type of callback spec frame with a few additional slots tailored specifically for the `Output` method. These additional slots allow you to pass flags and to define the output data form. This section describes only slots that are not included in the standard callback spec frame.

Slot descriptions

`sendFlags` Special protocol flags provided for certain communication tools. This slot is optional and defaults to `kMore`. Other possible values include `kPacket` and `kEOP`. (For more details, see the section "Sending Data" beginning on page 4-17.)

`form` A symbol defining how to translate the data being sent. The value can be `'string`, `'bytes`, `'binary`, `'number`, `'frame`, or `'template`. By default, this slot is set to `'string`, `'bytes`, or `'binary`, depending on the embedded NewtonScript type information. For more information, see the section "Data Forms" beginning on page 4-5.

`target` A slot used only when `form` is set to `'binary`. This slot contains a frame with the following two slots:

- `offset` An integer that is the offset from the beginning of the binary object at which to begin sending data.
- `length` An integer specifying the length of the data to send, in bytes.

For more information on sending binary data and using this slot, see the section "Working With Binary Data" beginning on page 4-27.

Input Spec Frame

The input spec frame 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.

Slot descriptions

`form`	A symbol identifying the input data form. This slot defaults to `'string`, and is evaluated when the input spec is set. You can override the default setting by using these other values: `'char`, `'number`, `'bytes`, `'binary`, `'template`, or `'frame`. For more information on these data forms, see the section "Data Forms" beginning on page 4-5.
`target`	A frame defining additional information pertaining to `'template` and `'binary` data forms. This frame is described in the section "Input Spec Target Frame" beginning on page 4-40.
`termination`	A frame defining input termination conditions. This frame is described in the section "Input Spec Termination Frame" beginning on page 4-41.
`discardAfter`	An integer that sets the input buffer size. If this buffer overflows, then the oldest bytes are discarded. The default value of this slot is 1024. Note that if you have set the `termination.byteCount` slot, or if the byte count is determined automatically, the value of this slot is ignored. This slot is evaluated only at the time the input spec is set.
`rcvFlags`	Certain communication tools require framed receiving. To use framed receiving, you must set this slot to `kPacket`; otherwise, set this slot to `nil` or don't include it at all.
`reqTimeout`	An integer specifying 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. This slot is optional, defaults to `kNoTimeout`, and is evaluated only at the time the `SetInputSpec` method is called.

`filter` A frame defining how incoming data is to be processed. This frame is described in the section "Input Spec Filter Frame" beginning on page 4-42.

`rcvOptions` An array of one or more communication tool options associated with the receive request. If you have just one option frame, you can specify it directly, without enclosing it in an array.

`partialFrequency`
An integer specifying 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. You must set this slot if you wish to use the `PartialScript` method, as the default value is 0. This slot is evaluated only at the time the input spec is set.

In addition to the slots listed here, you can define the following methods in the input spec frame:

- `InputScript`, which is called when one of the data input termination conditions is met (required method)
- `PartialScript`, which is called periodically at the frequency defined by the `partialFrequency` slot to allow you to sample the incoming data
- `CompletionScript`, which is called when the input is terminated unexpectedly

These methods are described in the following subsections.

InputScript

`InputScript(`*endpoint*`,` *data*`,` *terminator*`,` *options*`)`

This message is sent to the input spec frame when one of the data termination conditions has been met.

endpoint The endpoint associated with the receive request.

data The data that meets the input conditions is returned in this parameter, formatted as specified by the `form` slot of the input spec. Note that if you had set the `target` slot, *data* would be the target object.

terminator A frame specifying the condition that caused the input to terminate. Note that this data is irrelevant for the data forms `'frame` and `'template`, since input terminates automatically for them. The following slots are included:

- `condition` A symbol specifying the name of the slot in the input spec `termination` frame that caused the input to terminate (for example, `'byteCount`). If input was terminated by the `Input` method, this slot is set to `nil`.
- `index` The value of the index into the `termination.endSequence` array, if this was the condition that caused termination.
- `byteCount` The number of bytes received.

options The processed options originally set in the `rcvOptions` slot of the input spec. This parameter is `nil` if the `rcvOptions` slot is `nil`. For more information on the `rcvOptions` slot, see the section "Specifying Receive Options" beginning on page 4-24.

The return value of the `InputScript` method is ignored by the system.

In the input spec, the `InputScript` slot is evaluated every time the `InputScript` message is sent.

PartialScript

`PartialScript(`*endpoint*`,` *data*`)`

This message is sent to the input spec frame periodically, at the interval defined by the `partialFrequency` slot.

endpoint	The endpoint associated with the receive request.
data	All of the data currently in the input buffer is returned in this parameter, formatted as specified by the `form` slot of the input spec.

In the input spec, the `PartialScript` slot is evaluated every time the `PartialScript` message is sent.

CompletionScript

`CompletionScript(`*endpoint*`,` *options*`,` *result*`)`

The `CompletionScript` message is sent to an input spec frame when the input spec completes in an unexpected manner (for example, as a result of a time-out expiring or the `Cancel` method).

endpoint	The endpoint associated with the request.
options	This parameter is not used; you can ignore it.
result	The result code.

The `CompletionScript` method's return value is not used.

The `CompletionScript` slot in an input spec is evaluated every time the `CompletionScript` message is to be sent.

Input Spec Target Frame

This section describes in detail the `target` slot of an input spec frame. The `target` slot itself contains a frame defining additional information pertaining to the data form of the input.

Slot descriptions

`arglist`	The `arglist` array specification for the template. This slot must be defined only for `'template` data forms. You provide placeholder values in the array, which is filled in with actual data when it is received.
`typelist`	The `typelist` array specification for the template. This slot must be defined only for `'template` data forms. This slot is evaluated as needed.
`data`	The binary object, virtual binary object, or string into which received data is placed. This slot must be defined for `'binary` data forms only. It is evaluated as needed and is modified based on the received data.
`offset`	An integer specifying the offset within the binary object at which the received binary data is to be written. The offset is 0 by default. This slot is used only for binary data and is evaluated when the input spec is set.

Input Spec Termination Frame

This section describes in detail the `termination` slot of an input spec frame. The `termination` slot itself contains a frame defining input termination conditions.

The slots are listed here in order of precedence. They are evaluated only at the time the input spec is set.

Slot descriptions

`byteCount`	An integer indicating a number of bytes. If you know how many bytes you're expecting, specify that number here. Don't define this slot if you don't want to terminate input after a specified number of bytes.
`endSequence`	One or more objects, known as a termination sequence, to look for in the incoming data stream. This slot can hold a single character, a string, a number, or an array of bytes. Or, you can specify an array of these elements, where each element in the array defines a separate

termination sequence. If you want to specify just an array of bytes and no other sequence, you must specify it inside another array (for example, `[[...]]`).

`useEOP` Set this slot to `true` to terminate input based on a transport-level end-of-packet (EOP) indicator; otherwise, set it to `nil`. If this slot is set to `true`, and an EOP indicator is detected, input is terminated. 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 `useEOP` slot, the system effectively sets `useEOP` to the default value `true`.

Note

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. ◆

Input Spec Filter Frame

This section describes in detail the `filter` slot of an input spec frame. The `filter` slot itself contains a frame defining how incoming data is to be processed, or filtered.

Slot descriptions

`byteProxy` One or more characters or bytes in the input stream to be replaced by zero or one characters. This slot is evaluated only at the time the input spec is set. Specify

either a single frame, or an array of frames. Each frame must have the following slots:

`byte`	The single-byte character or byte to be replaced.
`proxy`	The single-byte character or byte to be used instead. This slot can also be `nil`, meaning that the original byte is to be removed completely from the input stream.

`sevenBit` Set to `true` to specify that the high-order bit of every incoming byte be stripped (“zeroed out”). This slot is evaluated only at the time the input spec is set, and its default value is `nil`.

Protos

This section describes endpoint protos.

protoBasicEndpoint

This is the basic endpoint object that encapsulates the details of a connection and contains methods that perform communication operations.

Slot descriptions

`encoding` A constant specifying a translation table to be used for the translation of all data to and from Unicode (the data representation on Newton). By default, this slot is set to `kMacRomanEncoding`. This slot is evaluated only when the endpoint is instantiated.

The methods in `protoBasicEndpoint` are described in the following subsections.

Instantiate

endpoint`:Instantiate(`*endpoint*`,` *options*`)`

This synchronous method instantiates and opens an endpoint.

endpoint	A reference to the endpoint you've defined.
options	An array containing a complete set of endpoint option frames. For more information, see the section "Endpoint Options" beginning on page 4-11.

The return value of this method is a clone of the array passed in the *options* parameter. The `result` slot in each option frame is set with a result code for the option.

Bind

endpoint`:Bind(`*options*`,` *bindCallback*`)`

Binds the endpoint to its local address and claims the needed system resources. When used synchronously, this method waits for the binding to be made before returning. When used asynchronously, this method posts the binding request and then returns. When the binding is made, the system calls your callback method.

options	An array of one or more option frames.
bindCallback	A callback spec frame (page 4-35) containing a method to be called when the request completes. Both the callback spec and the `async` slot within it must be defined if you want the `Bind` method to complete asynchronously. If you want to use this method synchronously, without a callback, specify `nil` for this parameter.

When this method is called synchronously, its return value is a clone of the array passed in the *options* parameter. The `result` slot in each option frame is set with a result code for the option.

UnBind

endpoint`:UnBind(`*unbindCallback*`)`

Releases the system resources and local address. When used synchronously, this method waits for the unbinding to complete before returning. When used asynchronously, this method posts the unbinding request and then returns. When the unbinding is complete, the system calls your callback method.

unbindCallback A callback spec frame (page 4-35) containing a method to be called when the request completes. Both the callback spec and the `async` slot within it must be defined if you want the `UnBind` method to complete asynchronously. If you want to use this method synchronously, without a callback, specify `nil` for this parameter.

Dispose

endpoint`:Dispose()`

This synchronous method closes the endpoint and deallocates the underlying endpoint structures.

Connect

endpoint`:Connect(`*options*`,` *connectCallback*`)`

Initiates a connection to the remote system. When used synchronously, this method waits for the connection to be made before returning. When used asynchronously, this method posts the connection request and then returns. Then, when the connection is made, the system calls your callback method.

options An array of one or more option frames.

connectCallback A callback spec frame (page 4-35) containing a method to be called when the request completes. Both the callback spec and the `async` slot within it must be defined if you want the `Connect` method to complete

asynchronously. If you want to use this method synchronously, without a callback, specify `nil` for this parameter.

When this method is called synchronously, its return value is a clone of the array passed in the *options* parameter. The `result` slot in each option frame is set with a result code for the option.

Note that if you are connecting to receive data, you must set up your first input spec by calling `SetInputSpec` (page 4-49) after a connection has been established (either by `Connect` or `Accept`).

Listen

endpoint`:Listen(`*options*`,` *listenCallback*`)`

Creates a connection to the remote system. After the connection is created, you must call the `Accept` or `Disconnect` method to accept or reject the connection.

options — An array of one or more option frames.

listenCallback — A callback spec frame (page 4-35) containing a method to be called when the request completes. Both the callback spec and the `async` slot within it must be defined if you want the `Listen` method to complete asynchronously. If you want to use this method synchronously, without a callback, specify `nil` for this parameter.

When this method is called synchronously, its return value is a clone of the array passed in the *options* parameter. The `result` slot in each option frame is set with a result code for the option.

Accept

endpoint`:Accept(`*options*`,` *acceptCallback*`)`

Accepts a connection after a connection was created by the `Listen` method.

options — An array of one or more option frames.

acceptCallback A callback spec frame (page 4-35) containing a method to be called when the request completes. Both the callback spec and the `async` slot within it must be defined if you want the `Accept` method to complete asynchronously. If you want to use this method synchronously, without a callback, specify `nil` for this parameter.

When this method is called synchronously, its return value is a clone of the array passed in the *options* parameter. The `result` slot in each option frame is set with a result code for the option.

Note that if you are accepting a connection to receive data, you must set up your first input spec by calling `SetInputSpec` (page 4-49).

Disconnect

endpoint`:Disconnect(`*cancelPending*`,` *disconnectCallback*`)`

Disconnects a connection.

cancelPending Set to `true` to specify that all outstanding requests should be canceled. Set to `nil` to wait for all pending output requests to complete before disconnecting. Note that if you set this parameter to `nil`, and an input spec is pending after all other requests have completed, the input spec is then canceled.

disconnectCallback A callback spec frame (page 4-35) containing a method to be called when the request completes. Both the callback spec and the `async` slot within it must be defined if you want the `Disconnect` method to complete asynchronously. If you want to use this method synchronously, without a callback, specify `nil` for this parameter.

For more discussion on canceling, see the section “Canceling Operations” beginning on page 4-28.

Note
This method incorporates both the `Disconnect` and `Release` methods from system software version 1. When the *cancelPending* parameter is set to `true`, this method is similar to the old `Disconnect` method. When the *cancelPending* parameter is set to `nil`, this method is similar to the old `Release` method. ◆

Output

endpoint`:Output(`*data*`,` *options*, *outputSpec*`)`

Sends the specified data.

data	The data to be sent.
options	An array of one or more option frames.
outputSpec	An output spec (page 4-36) containing a method to be called when the `Output` method completes, as well as other options. Both the output spec and the `async` slot within it must be defined if you want the `Output` method to complete asynchronously. If you want to use this method synchronously, without a callback, specify `nil` for this parameter.

When this method is called synchronously, its return value is a clone of the array passed in the *options* parameter. The `result` slot in each option frame is set with a result code for the option.

Note that when sending data with the `Output` method, you can take advantage of the default data forms by not explicitly specifying a data form in the output spec. 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. For example, if you are sending string data and you don't specify the data form, the `'string` data form is used by default.

The `Output` method also lets you specify the data as an array. For instance, if you specify a `'number` data form, you can specify the *data* parameter as an array whose elements are numbers. Other forms you can send as arrays are

`'string`, `'template`, `'char`, and `'binary`. (You cannot send arrays of arrays or arrays of the form `'frame`.)

If you do not specify the `form` slot (to use the default form), you can specify the *data* parameter as a heterogeneous array whose elements are characters, strings, numbers, or binary objects. This is a convenient way for you to concatenate similar calls to the `Output` method into a single call.

SetInputSpec

endpoint`:SetInputSpec(`*inputSpec*`)`

Sets the specified input spec as the active input spec.

inputSpec	The input spec frame to be set as active. Specifying `nil` indicates you don't want to post a new input spec. (For details on input spec frames, see the section "Input Spec Frame" beginning on page 4-37.)

If you call the `SetInputSpec` method and an input spec is already active, a `kCommScriptInputSpecAlreadyActive` error results. To prevent this error, you must first call the `Cancel` method to cancel the current input spec.

Input

endpoint`:Input()`

Terminates the current input spec and calls its associated `InputScript` method. All data in the input buffer is formatted and passed to the `InputScript` method, and the input buffer is cleared.

You use this method only when receiving data of the forms `'string` and `'bytes`.

IMPORTANT

An input spec must be active at the time this method is called, or the method throws an exception with the error `kCommScriptNoActiveInputSpec`. ▲

Partial

endpoint`:Partial()`

Returns all data in the input buffer, formatted according to the input data form specified in the input spec. The data is not removed from the input buffer. Use `FlushPartial` if you want the data removed from the input buffer.

You use this method only when receiving data of the forms `'string` and `'bytes`.

IMPORTANT

An input spec must be active at the time this method is called, or the method throws an exception with `kCommScriptNoActiveInputSpec`. ▲

FlushInput

endpoint`:FlushInput()`

Discards all bytes in the input buffer.

FlushPartial

endpoint`:FlushPartial()`

Discards all bytes in the input buffer through the last partial data read (see the `Partial` method on page 4-50).

Cancel

endpoint`:Cancel(`*cancelCallback*`)`

Cancels all pending requests, synchronous or asynchronous.

cancelCallback A callback spec frame (page 4-35) containing a `CompletionScript` method to be called when the request completes. Both the callback spec and the `async` slot within it must be defined if you want the `Cancel` method to complete asynchronously. This callback spec is slightly different from a standard

callback spec in that you cannot set a request time-out—the `reqTimeout` slot is ignored. If you want to use this method synchronously, without a callback, specify `nil` for this parameter.

If the `Cancel` method throws an exception with error -36003, that means that a cancel operation is already in progress. In this case, you can probably ignore the exception, but you might want to re-examine the program logic that caused this double cancel.

For more discussion of canceling, see the section "Canceling Operations" beginning on page 4-28.

Option

endpoint`:Option(`*options*`,` *optionCallback*`)`

Sets and/or returns the specified options, depending on the setting of the `opCode` slot in each of the option frames in the *options* array.

options	An array of one or more option frames.
optionCallback	A callback spec frame (page 4-35) containing a method to be called when the request completes. Both the callback spec and the `async` slot within it must be defined if you want the `Option` method to complete asynchronously. If you want to use this method synchronously, without a callback, specify `nil` for this parameter.

When this method is called synchronously, its return value is a clone of the array passed in the *options* parameter. The `result` slot in each option frame is set with a result code for the option.

You can specify options in the same array that are of the same type and 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.

ExceptionHandler

endpoint`:ExceptionHandler(`*error*`)`

The system sends your endpoint this message (if you provide it) whenever an exception is thrown and a corresponding `CompletionScript` method does not exist.

error — A frame (set by the system) describing the exception. The following slots are included:

- `name` — A string specifying the exception name (usually `|evt.ex.comm|`).
- `data` — An integer error code.
- `debug` — A symbol. This slot is used in the special case where a callback can't be called. It is described in more detail below. This kind of an error usually results in error –48803.

The `debug` slot of the *error* parameter is used in the special case where a callback can't be called. This slot can have one of the following symbol values: `'inputscript`, `'completionscript`, `'eventhandler`, or `'partialscript`. The value corresponds to the type of callback that caused the error. For example, if you defined an `InputScript` method with only one argument (an error), your `ExceptionHandler` method will be called with the `debug` slot of the *error* parameter set to `'inputscript`. Since this kind of error does not cause a break, you should check the `debug` slot for callback errors. This does not apply to the `ProgressScript` method used with the `protoStreamingEndpoint`.

You can think of exceptions as unsolicited events. If no `ExceptionHandler` method is specified, the exception is passed up the handler chain. Exceptions that are not caught are displayed as warning messages to the user.

EventHandler

endpoint`:EventHandler(`*event*`)`

The system sends your endpoint this message (if you provide it) whenever an event occurs that is not handled by the default endpoint event handlers.

Generally, you can catch events specific to a particular communication tool by using this method.

event A frame (set by the system) describing the event. The following slots are included:

`eventCode`	An integer event code.
`data`	An integer representing event data.
`serviceId`	A string representing the communication tool that originated the event. For example, `"mods"` identifies the modem tool.
`time`	An integer representing the time when the event occurred. This is the number of ticks since the system was last restarted, not including time when it was turned off.

State

endpoint`:State()`

This synchronous method returns the state of an endpoint. The possible return values are listed in Table 4-11 on page 4-62.

protoStreamingEndpoint

The `protoStreamingEndpoint` proto uses the `protoBasicEndpoint` as its proto. Besides all of the slots and methods included in `protoBasicEndpoint`, `protoStreamingEndpoint` includes two additional methods: `StreamIn` and `StreamOut`. These methods are described in this section.

StreamIn

`StreamIn(`*streamSpec*`)`

This synchronous method posts a receive request to the communication tool. As data arrives, it is unflattened into a frame and collected in memory or on a store. If you know that the frame will be large, you are advised to specify the `target.store` slot in the *streamSpec* parameter to receive the data in to

a store rather than in to the NewtonScript heap. Otherwise, you may get a heap overflow error during the receive operation.

Note that this method does not return until after the receive operation terminates.

streamSpec You may specify a frame that controls the receive operation, or if you don't need to, specify `nil`.

The *streamSpec* frame can have the following slots:

`form` Optional. This slot must be set to the symbol `'frame`, which is the default setting.

`reqTimeout` Optional. An integer specifying the time, in milliseconds, that the system should allow for each chunk to be received. If a time-out expires, the receive operation and *all* outstanding requests are canceled. This slot defaults to `kNoTimeout` and is evaluated only at the time the method is called.

`rcvFlags` Optional. This slot can contain flags provided for certain transport-level protocols. For more information, see the section "Specifying Flags for Receiving" beginning on page 4-22.

`target` Optional. If you are receiving a large frame that might overflow the NewtonScript heap, you can specify this slot to force it to be created directly on a store as a virtual object. This slot must contain a frame with a single slot, `store`. The `store` slot must contain a reference to the store on which the virtual object is to be created.

The `ProgressScript` method (page 4-55) can also be defined in the *streamSpec* frame.

StreamOut

`StreamOut(`*data*`, `*streamSpec*`)`

This synchronous method takes a frame, flattens it, and sends it in chunks.

Note that this method does not return until after the send operation completes.

data	The data to send. This object must be a frame.
streamSpec	You may specify a frame that controls the send operation, or if you don't need to, specify `nil`.

The *streamSpec* frame has the following slots:

`form`	Optional. This slot must be set to the symbol `'frame`, which is the default setting.
`reqTimeout`	Optional. An integer specifying the time, in milliseconds, that the system should allow for each chunk to be sent. If a time-out expires, the send operation and *all* outstanding requests are canceled. This slot defaults to `kNoTimeout` and is evaluated only at the time the method is called.
`sendFlags`	Optional. This slot can contain protocol flags provided for certain communication tools. For more details, see the section "Sending Data" beginning on page 4-17.

The `ProgressScript` method, described next, can also be defined in the *streamSpec* frame.

ProgressScript

`:ProgressScript(`*bytes*`, `*totalBytes*`)`

The system sends this message periodically to your *streamSpec* frame during the sending (`StreamOut`) or receiving (`StreamIn`) process to inform your application of progress.

bytes	The number of bytes that have been sent (or received) so far.

totalBytes The total number of bytes that are to be sent (or received).

A value of `nil` in either of these parameters signifies that the number is unknown.

This method must return a Boolean value. A return value of non-`nil` tells the system to continue sending (or receiving), and `nil` tells it to cancel the send (or receive) operation. Stopping the operation in this way is a "clean" cancel; that is, no errors are returned and no exceptions occur.

Functions and Methods

Utility Functions

This section includes a description of some global functions applicable to endpoint communications.

MakeAppleTalkOption

`MakeAppleTalkOption(`*NBPaddressString*`)`

Places the specified NBP (Name Binding Protocol) address string in an option frame that is usable by the `Connect` method. The option frame is returned.

NBPaddressString A string containing an AppleTalk NBP address.

MakeModemOption

`MakeModemOption()`

Returns an option frame created using the `'template` data form. This frame contains the modem `kCMOModemDialing` option, and the values are extracted from the user preferences stored in the system soup. The option frame is usable by the endpoint `Option` method or as an argument to any other endpoint method that takes an option frame as an argument.

MakePhoneOption

`MakePhoneOption(`*phoneString*`)`

Places the specified phone number string in an option frame of the `'address` type that is usable by the `Connect` method. The option frame is returned.

phoneString A string containing a phone number.

Translate

`Translate(`*data*`, `*translator*`, `*store*`, `*progressScript*`)`

This function translates data, returning an object that contains the translated data.

data The data to be translated. The type of this object depends on the translator used.

translator A symbol indicating the type of translator to use. Table 4-5 lists the translators available, and the corresponding type of the data object to be used with each.

store Specifies the store on which you want the translated object to be created. If you specify a valid store, the translated object is created as a virtual binary object on that store. This is recommended for large objects. If you specify `nil`, a normal object is created on the NewtonScript heap.

progressScript A method that may be called periodically during the translation process to inform your application of progress. This method is passed two parameters: the first is the number of bytes that have been translated so far, and the second is the total number of bytes that are to be translated. If either of these parameters is `nil`, that signifies that the number is unknown.

This callback method must return a Boolean value. A return value of non-`nil` tells the system to continue

translation, and `nil` tells the system to cancel the translation.

Note that this callback method is not implemented by either of the two existing translators, so this parameter is currently ignored.

Table 4-5 Data translators

Translator	Data type	Description
`'flattener`	Frame	Translates a frame into a binary object containing a flattened frame.
`'unflattener`	Binary object	Translates a binary object containing a flattened frame into a frame.

Summary of the Endpoint Interface

Constants and Symbols

Table 4-6 Data form symbols

Data form	Description
`'char`	Data is converted to or from Unicode, using the `encoding` slot.
`'number`	For receiving data or getting endpoint options, the data is interpreted as a 30-bit integer using 4 bytes. For sending data or setting endpoint options, the high-order 30 bits are placed into 4 bytes.
`'string`	For receiving data or getting endpoint options, the data is converted to Unicode using the endpoint `encoding` slot and returned as a NewtonScript character string with a termination byte. For sending data, the NewtonScript character string is converted from Unicode using the `encoding` slot. The termination byte is not sent.
`'bytes`	For receiving data or getting endpoint options, the data is returned as an array of unsigned single-byte values. For sending data, the value is squeezed into a single unsigned byte and truncated if necessary.
`'binary`	No conversion done.
`'template`	Used to exchange data with a service that expects C-type data.
`'frame`	For output, the frame is flattened into a stream of bytes prior to being sent, and for input, the byte stream is unflattened and returned as a frame.

Table 4-7 `Typelist` data types

Data type	Description
`'long`	Signed long integer
`'ulong`	Unsigned long integer
`'short`	16-bit unsigned short integer
`'byte`	8-bit unsigned byte
`'char`	8-bit character (translated to/from Unicode)
`'unicodechar`	16-bit Unicode character
`'boolean`	8-bit plain Boolean value
`'struct`	An aggregate structure, padded to a long word
`'array`	An aggregate array

Table 4-8 Option opcode constants

Constant	Value	Description
`opSetNegotiate`	256	Sets the option, but the system may substitute different values
`opSetRequired`	512	Sets the option, but fails if not possible
`opGetDefault`	768	Gets the default option value
`opGetCurrent`	1024	Gets the current option value

Table 4-9 Endpoint error codes

Constant	Value	Description
not defined	–16002	Bad communication tool command
not defined	–16008	Invalid call when connected
not defined	–16009	Invalid call when not connected
not defined	–16014	Call not supported by tool
not defined	–18003	Buffer overrun
not defined	–36003	Cancel is in progress
not defined	–36006	Operation not supported in the current tool state
not defined	–36008	System error
not defined	–36030	There's already a synchronous call pending
`kCommScriptNoActiveInputSpec`	–54000	An active input spec is required
`kCommScriptBadForm`	–54001	Error in the `form` slot of an input spec
`kCommScriptZeroLengthData`	–54002	Trying to send zero-length data
`kCommScriptExpectedSpec`	–54003	An input spec is required
`kCommScriptInvalidOption`	–54004	The option you tried to set was missing
`kCommScriptInvalidEndSequence`	–54005	Error in the `endSequence` slot of an input spec
`kCommScriptInappropriatePartial`	–54006	Used the `Partial` method with a bad input spec, or unable to do a partial input
`kCommScriptInappropriateTermination`	–54007	Error in `termination` slot of input spec

continued

Table 4-9 Endpoint error codes (continued)

Constant	Value	Description
`kCommScriptInappropriateTarget`	–54008	Error in `target` slot of input spec
`kCommScriptInappropriateFilter`	–54009	Error in `filter` slot of input spec
`kCommScriptExpectedTarget`	–54010	Attempted to receive binary data with no target object specified
`kCommScriptExpectedTemplate`	–54011	Attempted to send or receive template data without a template specified
`kCommScriptInputSpecAlreadyActive`	–54012	Tried to set an input spec when one was already active
`kCommScriptInvalidProxy`	–54013	Invalid value in `filter.byteProxy.proxy` slot of input spec
`kCommScriptNoEndpointAvailable`	–54014	Endpoint object is missing
`kCommScriptInappropriateCall`	–54015	Method not supported, or called inappropriately
`kCommScriptCharNotSingleByte`	–54016	The character specified in the `filter.byteProxy.proxy` slot of the input spec is more than a single byte

Table 4-11 Endpoint state constants

Constant	Value	Description
`kUninit`	0	Uninitialized
`kUnbnd`	1	Unbound
`kIdle`	2	Idle

Table 4-10 Option error codes

Constant	Value	Description
`kCommOptionFailure`	–54021	Operation failed
`kCommOptionPartSuccess`	–54022	Option set, but set value is different from requested value
`kCommOptionReadOnly`	–54023	Set attempted on read-only option
`kCommOptionNotSupported`	–54024	Option not supported
`kCommOptionBadOpCode`	–54025	`Opcode` value is invalid
`kCommOptionNotFound`	–54026	Option not found
`kCommOptionTruncated`	–54027	One or more requested options missing

Table 4-11 Endpoint state constants

Constant	Value	Description
`kOutCon`	3	Outgoing connection pending
`kInCon`	4	Incoming connection pending (`Listen` method has completed but you have not yet called `Accept` or `Disconnect`)
`kDataXfer`	5	Data transfer
`kOutRel`	6	Outgoing release pending
`kInRel`	7	Incoming release pending (connection released by the remote side)
`kInFlux`	8	State is changing
`kOutLstn`	9	`Listen` method pending

Data Structures

Option Frame

```
myOption := {
type: symbol, // option type
```

Table 4-12 Other endpoint constants

Constant	Value	Description
`kNoTimeout`	0	Set no time-out for a request
`kEOP`	0	Send or receive flag; marks the last packet
`kMore`	1	Send or receive flag; more data is coming
`kPacket`	2	Send or receive flag; data is in packets (framed)

```
label: string, // 4-char option identifier
opCode: integer, // an opCode constant
form: 'template, // standard 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: { // can be a single frame or an array
      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: // // terminates input spec and returns data
   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 virtual binary
            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*`)`

Built-in Communication Tools

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

- Serial
- Modem
- Infrared
- AppleTalk

These communication tools are accessed and used through the Endpoint interface. This chapter describes the options available for each of these tools and other information that applies to their use. For basic information on using communication endpoints, see Chapter 4, "Endpoint Interface."

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

Here's an example of how to create a standard asynchronous serial endpoint:

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

The remainder of this section describes options you can use to configure the serial communication tool. Table 5-1 summarizes the standard serial options.

You can get or set most options in the endpoint method that established the state, as appears in Table 5-1. The endpoint options are set by passing an argument to the communication tool when calling one of the endpoint methods such as `Instantiate`, `Bind`, `Connect`, and the like. Passing the option to the `Bind` method, for example, causes the system to set the option and then do the binding.

Take the first option in Table 5-1, `kCMOSerialHWChipLoc`. It should be used after the endpoint has been instantiated and until the binding is made, but not after the binding is made. That means you could use it in the `Instantiate` and `Bind` methods, but not in a `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, if you do use an option, you must specify a value for each field within it—the defaults do not apply partially.

Table 5-1 Summary of serial options

Label	Value	Use When	Description
`kCMOSerialHWChipLoc`	"schp"	Before binding	Sets what serial hardware to use
`kCMOSerialChipSpec`	"sers"	Any time	Sets what 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 connecting	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
`kCMOSerialBreak`	"sbrk"	After connecting	Sends a break
`kCMOSerialDiscard`	"sdsc"	After connecting	Discards data in input and/or output buffer
`kCMOSerialEventEnables`	"sevt"	After connecting	Configures the serial tool to complete an endpoint event on particular state changes

continued

Table 5-1 Summary of serial options (continued)

Label	Value	Use When	Description
`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

Serial Chip Location Option

This option specifies which serial hardware in the system to use for the endpoint. This option must be set before the endpoint is connected.

Here is an example of this option:

```
local option := {
   type: 'option,
   label: kCMOSerialHWChipLoc,
   opCode: opSetRequired,
   form: 'template,
   result: nil,// not needed; returned
   data : {
      arglist: [
         kHWLocExternalSerial,
         0
      ],
      typelist: [
         'struct,
         ['array, 'char, 4], // location label
         'ulong // service ID
      ]
   }
};
```

The possible values for the location label field within the `data` slot are listed in Table 5-2. Note that these hardware locations are hardware platform dependent.

Table 5-2 Serial chip location labels

Constant	Value	Description
`kHWLocExternalSerial`	"extr"	Use the external serial port (typical default)
`kHWLocBuiltInIR`	"infr"	Use the built-in infrared port
`kHWLocBuiltInModem`	"mdem"	Use the built-in modem
`kHWLocPCMCIASlot1`	"slt1"	Use the application card in slot 1
`kHWLocPCMCIASlot2`	"slt2"	Use the application card in slot 2

Note that the default setting is typically the external serial port, but this may vary since the default is set by the communication tool.

The service ID field within the `data` slot specifies a four-character string identifying a communication tool. If the location label slot is `nil`, the default serial chip location for the specified communication tool is used, regardless of whether or not this is the current tool.

The service ID field and the location label field are mutually exclusive. You should specify an identifier in only one of these fields. If you specify both fields, the location label field takes precedence.

Serial Chip Specification Option

This option is used to specify or return information about the current serial chip. It can be used to select the serial hardware with which to bind and is especially useful for selecting serial hardware on a application card device. This option is a superset of the serial chip location option.

If you use this option to select the serial hardware with which to bind, it must be set before the endpoint `Bind` method is called.

The serial chip specification option also returns information about the serial hardware. When using this option to return information, you can call it any time.

Here is an example of this option used to return serial hardware information:

```
local option := {
   type: 'option,
   label: kCMOSerialChipSpec,
   opCode: opSetRequired,
   form: 'template,
   result: nil,// not needed; returned
   data : {
      arglist: [
         0, // chip location
         0, // features supported by this chip
         0, // output signals supported by chip
         0, // input signals supported by chip
         0, // parity supported
         0, // data and stop bits supported
         0, // serial chip type
         nil, // chip in use
         0, // reserved
         0, // reserved
         0, // application card CIS manufacturer ID
         0 // application card CIS manufacturer ID info
      ],
      typelist: [
         'struct,
         'ulong, // fHWLoc
         'ulong, // fSerFeatures
         'byte, // fSerOutSupported
```

```
            'byte, // fSerInSupported
            'byte, // fParitySupport
            'byte, // fDataStopBitSupport
            'byte, // fUARTType
            'boolean, // fChipNotInUse
            'byte, // reserved
            'byte, // reserved
            'short, // fCIS_ManFID
            'short // fCIS_ManFIDInfo
        ]
    }
};
```

The fields in the serial chip specification option frame appear in Table 5-3.

Table 5-3 Serial chip specification option fields

Option Field	Description
`fHWLoc`	Specifies the serial chip location. Default is 0.
`fSerFeatures`	Features supported by this chip. Default is 0.
`fSerOutSupported`	Output signals supported by this chip. Default is 0.
`fSerInSupported`	Input signals supported by this chip. Default is 0.
`fParitySupport`	Parity supported by this chip. See Table 5-4 for the constants you can specify. Default is 0.
`fDataStopBitSupport`	Number of data and stop bits supported by this chip. See Table 5-4 for the constants you can specify. Default is 0.
`fUARTType`	Type of serial chip. See Table 5-4 for the constants you can specify. Default is 0.

continued

Table 5-3 Serial chip specification option fields (continued)

Option Field	Description
`fChipNotInUse`	A Boolean specifying whether or not the chip is in use (default is `true`, which means the chip is not in use).
`fCIS_ManFID`	Application card CIS manufacturer ID. Default is 0.
`fCIS_ManFIDInfo`	Application card CIS manufacturer ID information. Default is 0.

The constants you can use to specify various field values in the serial chip specifications option are listed in Table 5-4.

Table 5-4 Serial chip specification option constants

Constant	Value	Description
Parity Support Constants		
`kSerCap_Parity_Space`	0x00000001	No parity
`kSerCap_Parity_Mark`	0x00000002	Mark parity
`kSerCap_Parity_Odd`	0x00000004	Odd parity
`kSerCap_Parity_Even`	0x00000008	Even parity
Data and Stop Bits Support Constants		
`kSerCap_DataBits_5`	0x00000001	5 data bits
`kSerCap_DataBits_6`	0x00000002	6 data bits
`kSerCap_DataBits_7`	0x00000004	7 data bits
`kSerCap_DataBits_8`	0x00000008	8 data bits
`kSerCap_StopBits_1`	0x00000010	1 stop bit
`kSerCap_StopBits_1_5`	0x00000020	1.5 stop bits

continued

Table 5-4 Serial chip specification option constants (continued)

Constant	Value	Description
kSerCap_StopBits_2	0x00000040	2 stop bits
kSerCap_StopBits_All	0x00000070	Supports all stop bit choices
kSerCap_DataBits_All	0x0000000F	Supports all data bit choices
Serial chip types		
kSerialChip8250	0x00	8250 UART
kSerialChip16450	0x01	16450 UART
kSerialChip16550	0x02	16550 UART
kSerialChip8530	0x20	8530 UART (SCC chip)
kSerialChip6850	0x21	6850 UART (Brick ASIC modem port UART)
kSerialChip6402	0x22	6402 UART (Brick ASIC infrared port UART)
kSerialChipVoyager	0x23	Cirrus Voyager UART chip
kSerialChipUnknown	0x00	Unknown type of UART

Serial Circuit Control Option

This option controls usage of the serial control lines. This option must be set after the endpoint is connected.

Note that in the external serial port, DTR and RTS signals are combined on the HSKo line, and the RTS line is used for hardware input flow control.

IMPORTANT

The RTS line should not be set or cleared if hardware input flow control is enabled. ▲

Here is an example of the serial circuit control option:

```
local option := {
   type: 'option,
   label:kCMOSerialCircuitControl,
   opCode:opSetRequired,
   form: 'template,// not needed
   result: nil,// not needed; returned
   data : {
      arglist: [
         kSerOutDTR, // set DTR
         0,       // use 1K byte receive buffer
         0,       // will be set on return
         0        // will be set on return
      ],
      typelist: [
         kStruct,
         kUByte, // fSerOutToSet
         kUByte, // fSerOutToClear
         kUByte, // fSerOutState
         kUByte // fSerInState
      ]
   }
};
```

The fields in serial circuit control option frame appear in Table 5-5.

Table 5-5 Serial circuit control option fields

Option Field	Description
`fSerOutToSet`	Output lines to assert. Add together the values from Table 5-6 for each output line you want to assert. Default is 0.
`fSerOutToClear`	Output lines to negate. Add together the values from Table 5-6 for each output line you want to negate. Default is 0.
`fSerOutState`	Current output line state. This field is returned after any lines you specify are set or cleared.
`fSerInState`	Current input line state. This field is returned.

The constants you can use to specify the various serial control lines are listed in Table 5-6.

Table 5-6 Serial circuit control option constants

Constant	Value	Description
Serial Output Lines		
`kSerOutDTR`	0x01	DTR line
`kSerOutRTS`	0x02	RTS line (also known as HSKo on the external serial port)
Serial Input Lines		
`kSerInDSR`	0x02	DSR line
`kSerInDCD`	0x08	DCD line (also known as GPi on the external serial port)

continued

Table 5-6 Serial circuit control option constants (continued)

Constant	Value	Description
`kSerInRI`	0x10	RI line (also known as GPi on the external serial port)
`kSerInCTS`	0x20	CTS line (also known as HSKi on the external serial port)
`kSerInBreak`	0x80	A "break" condition

When the `kSerInBreak` bit is a one, it means that the serial chip has detected a "break" condition on the receive data line. Normally the line is logically high when characters are not being sent and in between characters ("marking", binary 1, less than -3 volts). It drops low ("spacing", binary 0, greater than +3 volts) at the start of a character (start bit), and is high for a bit time at the end of a character (stop bit). If the line is held low for more than a byte time, the serial chip reports a "break" condition, and a consequent interrupt on it.
You can ask for an "event" by means of the `TCMOSerialEventEnables` option on start and end of break. You can use this in terminal programs as a kind of user-initiated interrupt.

You can also send a break for a specified amount of time by using the `TCMOSerialBreak` option.

Serial Buffer Size Option

This option let's you increase the size of the buffers used by the serial tool. Buffers larger than 4 KB are not supported; an error results if you specify too large a buffer. Also note that an out-of-memory error may return at connect time if the serial tool cannot allocate the buffers.

This option is often useful—appropriate buffer size can increase performance and decrease overrun errors. A good buffer size is a few bytes larger than the typical packet size, for communications using packet-oriented protocols.

For streamed communications, output buffer size is not as important as input buffer and can be left at the typical output size. The input buffer can be

increased, especially for data rates above 9600 bps. If no flow control is operating, input buffer size may be the only way to control overruns.

In addition to setting the size of the input and output buffers, this option sets the number of receive error characters to remember. The specification of receive markers can be left at a small number like 8, since multiple errors typically mean something is wrong with the link, and buffering more than 8 error characters won't provide much more interesting information (data is often flushed after errors anyway). The total size of the input buffer is limited to 4 KB, which includes about 8 bytes per marker. Typical input buffer size is 256 to 1024 bytes.

Note that the usable size of a buffer is usually one-to-four bytes less than the buffer size because of DMA boundary constraints and other considerations.

The serial buffer size option must be set before or at connect time.

Here is an example of this option:

```
local option := {
   type: 'option,
   label:kCMOSerialBuffers,
   opCode:opSetRequired,
   form: 'template,// not needed
   result: nil,// not needed; returned
   data : {
      arglist: [
         256,     // use 256 byte transmit buffer
         1024, // use 1K byte receive buffer
         8,       // remember up to 8 error characters
      ],
      typelist: [
         'struct,
         'ulong, // output buffer size in bytes
         'ulong, // input buffer size in bytes
         'ulong, // num of error characters to remember
      ]
```

```
   }
};
```

The default output buffer size is 512 bytes; the default input buffer size is 512 bytes.

Serial Configuration Option

This option specifies the bps rate, stop bits, data bits, and parity options for the serial tool. This option is typically specified at endpoint open or connect time, but you can also use it to change the data format after connection. Changing after connection may require resetting some serial chips, which can cause problems with the serial inputs.

Here is an example that shows the use of this option:

```
local option := {
   type: 'option,
   label:kCMOSerialIOParms,
   opCode:opSetRequired,
   form: 'template,// not needed
   result: nil,// not needed; returned
   data : {
      arglist: [
         k1StopBits, // 1 stop bit
         kNoParity, // no parity bit
         k8DataBits,// 8 data bits
         k57600bps, // date rate 57600 bps
      ],
      typelist: [
         'struct,
         'long, // stop bits
         'long, // parity
         'long, // data bits
         'long, // bps
```

```
        ]
    }
};
```

In the stop bits field, you can use the following constants:

Constant	Value	Description
`k1StopBits`	0	1 stop bit (default)
`k1pt5StopBits`	1	1.5 stop bits
`k2StopBits`	2	2 stop bits

In the parity field, you can use the following constants:

Constant	Value	Description
`kNoParity`	0	no parity (default)
`kOddParity`	1	odd parity
`kEvenParity`	2	even parity

In the data bits field, you can use the following constants:

Constant	Value (Number of Data Bits)
`k5DataBits`	5
`k6DataBits`	6
`k7DataBits`	7
`k8DataBits`	8 (default)

In the bps field, you can use the following constants to specify the interface speed:

Constant	Value
`kExternalClock`	1
`k300bps`	300
`k600bps`	600
`k1200bps`	1200

Constant	Value
k2400bps	2400
k4800bps	4800
k7200bps	7200
k9600bps	9600 (default)
k12000bps	12000
k14400bps	14400
k19200bps	19200
k38400bps	38400
k57600bps	57600
k115200bps	115200
k230400bps	230400

Serial Data Rate Option

This option is used for changing the bps rate after a connection has already been made. If data bit, stop bit, and parity options don't have to be changed, this is a relativelym trouble-free way of changing the data rate.

Here is an example of this option:

```
local option := {
   type: 'option,
   label:kCMOSerialBitRate,
   opCode:opSetRequired,
   form: 'number,
   result: nil,// not needed; returned
   data: k19200bps,// change to 19200
};
```

In the `data` slot, you can specify the same constants as for the bps field in the serial configuration option, given on page 5-15. The default is 9600 bps.

Serial Flow Control Options

The two serial flow control options configure software and hardware flow for input and output. Software flow control uses XON and XOFF characters to control data flow. Hardware flow control uses the RTS line for input flow control and the CTS line for output flow control.

The two serial flow control options are `kCMOInputFlowControlParms` for input and `kCMOOutputFlowControlParms` for output.

Here is an example of setting the `kCMOOutputFlowControlParms` option. The `kCMOInputFlowControlParms` option is set in an identical way.

```
local option := {
   type: 'option,
   label:kCMOOutputFlowControlParms,
   opCode:opSetRequired,
   form: 'template,// not needed
   result: nil,// not needed; returned
   data : {
      arglist: [
         unicodeDC1, // xonChar
         unicodeDC3, // xoffChar
         true, // useSoftFlowControl
         nil,  // useHardFlowControl
         0,       // returned
         0,       // returned
      ],
      typelist: [
         'struct,
         'char, // XON character
         'char, // XOFF character
         'boolean, // software flow control
         'boolean, // hardware flow control
         'boolean, // hardware flow blocked
         'boolean, // software flow blocked
```

```
        ]
    }
};
```

The fields in the serial flow control option frame are described in Table 5-7.

Table 5-7 Serial flow control option fields

Option Field	Description
XON character	Specifies the XON character to use for software flow control (default is DC1, `0x11`).
XOFF character	Specifies the XOFF character to use for software flow control (default is DC3, `0x13`).
software flow control	To enable software flow control, specify `true`. To disable it, specify `nil` (default).
hardware flow control	To enable hardware flow control, specify `true`. To disable it, specify `nil` (default).
hardware flow blocked	Read-only. Returns `true` if hardware flow control is blocked.
software flow blocked	Read-only. Returns `true` if software flow control is blocked.

Serial Send Break Option

This option is used to send a break (string of start bits) for the amount of time specified. No synchronization is done with output.

This option is used after the endpoint is connected. Note that you can only set this option; you can't read the current setting, since the option performs an action rather than setting some kind of parameter.

Here is an example that shows the use of this option:

```
local option := {
   type: 'option,
   label:kCMOSerialBreak,
   opCode:opSetRequired,
   form: 'number,
   result: nil,// not needed; returned
   data: 100*kMilliseconds, // send a 100 millsecond break
};
```

In the `data` slot, specify the length of time for the break in milliseconds by specifying an integer multiplied by the constant `kMilliseconds`. The default is 75 milliseconds.

Serial Discard Data Option

This option is used to discard data in the input or output buffers. Discarding is useful after error conditions, or before synchronization is achieved in serial communication.

This option is used after the endpoint is connected. Note that you can only set this option; you can't read the current setting, since this option performs an action rather than setting some kind of parameter.

With modem endpoints, this option works only when MNP is not being used.

Here is an example that shows the use of this option:

```
local option := {
   type: 'option,
   label:kCMOSerialDiscard,
   opCode:opSetRequired,
   form: 'template,// not needed
   result: nil,// not needed; returned
   data : {
      arglist: [
```

```
            true, // discard input chars
            nil,  // but not output
         ],
         typelist: [
            'struct,
            'boolean, // clear input buffer
            'boolean, // clear output buffer
         ]
      }
};
```

The first data field controls the input buffer and the second data field controls the output buffer. Specify `true` to discard data in a buffer, or `nil` to leave it untouched.

The default for the input buffer is `true`, meaning discard characters. The default for the output buffer is `nil`, meaning leave it untouched.

Serial Event Configuration Option

This option configures the serial tool to send the endpoint the `EventHandler` message when particular state changes occur.

Here is an example that shows the use of this option:

```
local option := {
   type: 'option,
   label:kCMOSerialEventEnables,
   opCode:opSetRequired,
   form: 'template,// not needed
   result: nil,// not needed; returned
   data : {
      arglist: [
         kSerialEventHSKiNegatedMask +
         kSerialEventHSKiAssertedMask,
                  // send event on CTS/HSKi changes
```

```
            0,       // no DCD event specified
         ],
         typelist: [
            'struct,
            'ulong, // event masks
            'ulong, // DCD down time, in microseconds
         ]
      }
};
```

The first data field specifies one or more event mask constants, for state changes that you want to trigger an event. Simply add the constants together to specify more than one event. The allowable values are listed in Table 5-8. The default is zero, for no events.

For the `kSerialEventDCDNegatedMask` event, specify in the second data field the amount of time, in microseconds, that DCD must be negated before this event is reported. It's common for the carrier to drop for short periods of time during a connection, and this is a way to mask drops that are not significant. The default is zero.

Table 5-8 Serial event constants

Constant	Value	Description
`kSerialEventBreakStartedMask`	`0x00000001`	A serial line break condition is detected
`kSerialEventBreakEndedMask`	`0x00000002`	A serial line break condition ends
`kSerialEventDCDNegatedMask`	`0x00000004`	The DCD line is negated (DCD is also known as GPi in the external serial port)
`kSerialEventDCDAssertedMask`	`0x00000008`	The DCD line is asserted

continued

Table 5-8 Serial event constants

Constant	Value	Description
`kSerialEventHSKiNegatedMask`	`0x00000010`	The CTS line is negated (CTS is also known as HSKi in the external serial port)
`kSerialEventHSKiAssertedMask`	`0x00000020`	The CTS line is asserted
`kSerialEventExtClkDetectEnableMask`	`0x00000040`	The serial tool detects more than 100 transitions per second on the CTS line, and thus assumes this line is a clock input

When the serial tool sends an `EventHandler` message to the endpoint, it passes two parameters. The first is the integer 1. The second parameter is an integer indicating the event that occurred, using the same mask bits as shown in Table 5-8. Some higher-order bits may be set as well, so don't count on them being zero.

In the endpoint, you must provide an `EventHandler` method (page 4-52) to receive the message from the serial tool.

Serial Bytes Available Option

This read-only option returns the number of bytes waiting to be read from the receive buffer.

This option is used after the endpoint is connected.

Here is an example that shows the user of this option:

```
local option := {
   type: 'option,
   label:kCMOSerialBytesAvailable,
   opCode:opGetCurrent,
   form: 'number,
```

```
   result: nil,// not needed; returned
   data: 0, // returned
};
```

Serial Statistics Option

This read-only option returns various software and hardware statistics related to the serial tool.

This option is used after the endpoint is connected.

Here is an example that shows the use of this option:

```
local option := {
   type: 'option,
   label:kCMOSerialIOStats,
   opCode:opGetCurrent,
   form: 'template,// not needed
   result: nil,// not needed; returned
   data : {
      arglist: [
         0,       // returned
         0,       // returned
         0,       // returned
         0,       // returned
         0,       // returned
         0,       // returned
         nil,     // returned
      ],
      typelist: [
         'struct,
         'ulong, // parity error count
         'ulong, // framing error count
         'ulong, // soft overrun count
         'ulong, // hard overrun count
```

```
        'byte, // GPi state
        'byte, // HSKi state
        'boolean // external clock detect
    ]
  }
};
```

The fields in this option frame are described in Table 5-9.

Table 5-9 Serial statistics option fields

Option Field	Description
parity error count	Number of parity errors encountered. Reading this value resets it to zero.[1]
framing error count	Number of framing errors encountered. Reading this value resets it to zero.[1]
soft overrun count	Number of soft overrun errors encountered. Reading this value resets it to zero.[1]
hard overrun count	Number of hard overrun errors encountered. Reading this value resets it to zero.[1]
GPi state	State of DCD (GPi) line. Zero = negated, one = asserted
HSKi state	State of CTS (HSKi) line. Zero = negated, one = asserted
external clock detect	`True` if an external clock is detected, otherwise `nil`

[1] The count is cumulative from the last time the statistics were read by this option call, or from the time of the endpoint `Open` call if they haven't been read yet.

Parity, framing, and overrun errors can all happen when receiving data. Hard overruns happen when the serial driver doesn't unload the data from the hardware before it is overwritten by subsequent data. Soft overruns

happen when the endpoint doesn't consume data fast enough and the serial tool buffer fills up, resulting in discarded data.

Soft overruns can be avoided by using input flow control, by increasing the serial tool's receive buffer, and by handling the data from the serial tool in a more efficient manner.

Serial External Clock Divide Option

This option controls how the clock rate is divided when using an external clock.

This option can be used during the `Connect` call or after the endpoint is connected.

Here is an example that shows the use of this option:

```
local option := {
   type: 'option,
   label:kHMOSerExtClockDivide,
   opCode:opSetRequired,
   form: 'byte,
   result: nil,// not needed; returned
   data: kSerClk_DivideBy_16,
};
```

In the `data` slot, you can use the following constants:

Constant	Value	Description
`kSerClk_Default`	0x00	Use the default
`kSerClk_DivideBy_1`	0x80	Divide by 1
`kSerClk_DivideBy_16`	0x81	Divide by 16
`kSerClk_DivideBy_32`	0x82	Divide by 32
`kSerClk_DivideBy_64`	0x83	Divide by 64

Serial Tool with MNP Compression

You can create an asynchronous serial endpoint with MNP compression. This endpoint works just like a standard asynchronous serial endpoint, except that it uses MNP data compression.

Here's an example that shows how to create such an endpoint:

```
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 5-10.

Table 5-10 Summary of serial tool with MNP options

Label	Value	Use when	Description
`kCMOMNPCompression`	"mnpc"	After connecting, before running	Sets the data compression type (See page 5-57)
`kCMOMNPDataRate`	"eter"	Any time	Configures internal MNP timers

The first option in Table 5-10, `kCMOMNPCompression`, is described under "MNP Compression Option" on page 5-57. The last option, `kCMOMNPDataRate`, is most often used for serial tools with MNP and so is described here.

Serial MNP Data Rate Option

The serial MNP data rate option is used by MNP to configure its internal timers. This option is required because the serial port speed may be different from the end-to-end speed.

When using a serial MNP endpoint, this option must be set to the correct value for MNP to function correctly, and the option must be set at or before the endpoint `Connect` call.

Here is an example that shows the use of this option:

```
local option := {
   type: 'option,
   label:kCMOMNPDataRate,
   opCode:opSetRequired,
   form: 'number,
   data : 2400,
};
```

The `data` slot must be set to the rate, in bps, of the raw throughput of the serial link used by MNP. The default value is 2400 bps.

Framed Asynchronous Serial Tool

The framed asynchronous serial tool is a superset of the standard asynchronous serial tool, supporting two additional framing options, which are summarized in Table 5-1 and described in detail in the following subsections.

Table 5-11 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

When you use the framed asynchronous serial tool, if framing is not specified for a send or receive operation, this tool works exactly like the standard asynchronous serial tool.

When framing is specified on input, the framed asynchronous serial tool discards characters until a start of frame sequence is detected; input completes with an EOF indication when the end of frame sequence is detected; and an error is reported if a CRC error is detected. When framing is specified on output, the data is prefixed with the start-of-frame sequence, and the end of frame sequence and calculated CRC are sent at the end of the data. The escape character is used for data transparency during framed operations.

Because the framed asynchronous serial tool supports packetized data, an endpoint can include `kPacket` and `kEOP` and `kMore` flags to control the sending and receiving of framed (packetized) data. For more information on these flags (See “Sending Data” beginning on page 4-17 of Chapter 4, “Endpoint Interface.”)

Here’s an example that shows how to create a framed asynchronous serial endpoint:

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

Serial Framing Configuration Option

This option configures data framing parameters. This option applies only to the framed asynchronous serial tool.

Here is an example that shows the use of this option:

```
local option := {
   label:  kCMOFramingParms,
   type:   'option,
   opCode: opSetRequired,
   data : {
      arglist: [
         unicodeDLE, // escape character
         unicodeETX, // EOM character
         true, // syn/dle/stx header
         true, // send crc at end
         true, // check crc on receive
      ],
      typelist: [
         'struct,
         'char,
         'char,
         'boolean,
         'boolean,
         'boolean,
      ]
   }
};
```

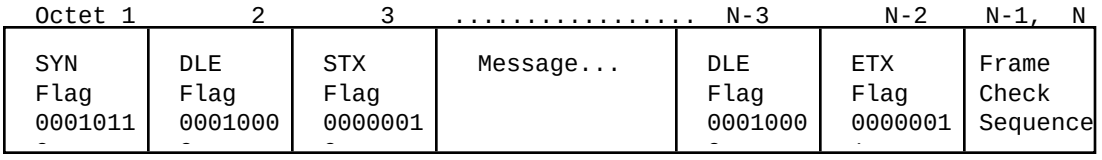
The fields in the serial framing configuration option frame are described in Table 5-12.

Table 5-12 Serial framing configuration option fields

Option Field	Description
escape character	Specifies the character to use for escape. The default is DLE (`0x10`).
EOM character	Specifies the character to use for end of message. The default is ETX (`0x03`).
syn/dle/stx header	To include the SYN/DLE/STX header, specify `true`. To disable this feature, specify `nil`.
send crc at end	To compute and send a 2-byte CRC at the end of a frame, specify `true`. To disable this feature, specify `nil`.
check crc on receive	To compute and check the 2-byte CRC at the end of each frame, specify `true`. To disable this feature, specify `nil`.

The example above of setting the serial framing configuration option shows the default settings, which implement BSC framing. This kind of framing is shown in Figure 5-1.

Figure 5-1 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.

Serial Framing Statistics Option

This read-only option returns the number of bytes that have been discarded from the receive buffer while looking for a valid frame header. This option applies only to the framed asynchronous serial tool.

Here is an example that shows the use of this option:

```
local option := {
   type: 'option,
   label:kCMOFramedAsyncStats,
   opCode:opGetCurrent,
   form: 'number,
   result: nil,// not needed; returned
   data: 0, // not needed; # bytes discarded is returned
};
```

Modem Tool

The modem tool includes built in support for 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.

Here's an example of how to create a modem endpoint:

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

This remainder of this section describes various options you can use to configure the modem communication tool. Table 5-13 summarizes the modem options described here.

Table 5-13 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

continued

Table 5-13 Summary of modem options (continued)

Label	Value	Use When	Description
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
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"	After connecting, before running	Sets the data compression type
kCMOMNPStatistics	"mnps"	After connecting	Read-only option reporting performance statistics from the current MNP connection

Modem Address Option

You'll need to supply a modem address option that gives an address for use during the connection phase. Set the `kPhoneNumber` constant to establish that you're specifying a phone number.

Here is an example:

```
local option := {
      label:  kCMARouteLabel,
      type:   'address,
      opCode: opSetRequired,
      data: {
```

```
        arglist: [
           kPhoneNumber,        // type
           size,                //phone string length
           phoneStr,            //the phone number
        ],
        typelist: [
           kStruct,
           kLong,
           kULong,
           [kArray, kChar, 0],
        ]
     }
  };
```

Alternatively, you can call the global function `MakeModemOption` to construct an address option. For more information on `MakeModemOption` (See page 4-56 in Chapter 4, "Endpoint Interface.")

Modem Preferences Option

Modem preferences allow the configuration of the modem controller operation. Certain features of the controller can be enabled or disabled using the preferences. This option must be set before the endpoint `Bind` call. For example, setting this option in the endpoint configuration options is appropriate.

Here is an example of this option:

```
local option := {
   type: 'option,
   label:kCMOModemPrefs,
   opCode:opSetRequired,
   form: 'template,// not needed
   data : {
```

```
      arglist: [
         true,    // connect in direct mode
         true,    // id modem
         true,    // require positive id
         true,    // use hardware cd
         true,    // use software cd
         true,    // use config string
         true,    // use dial options
         true,    // hang up at disconnect
         true,    // enable pass thru
         true,    // enable dial out stream
         19200,   // direct mode speed
         3,          // hwcd delay low speed
         15,         // hwcd delay high speed
      ],
      typelist: [
         'struct,
         'boolean, // fConnectInDirectMode
         'boolean, // fIdModem
         'boolean, // fRequirePositiveId
         'boolean, // fUseHardwareCD
         'boolean, // fUseSoftwareCD
         'boolean, // fUseConfigString
         'boolean, // fUseDialOptions
         'boolean, // fHangUpAtDisconnect
         'boolean, // fEnablePassThru
         'boolean, // fEnableDialOutStream
         'ulong, // fDirectModeSpeed
         'ulong, // fHWCDDelayLowSpeed
         'ulong, // fHWCDDelayHighSpeed
      ]
   }
};
```

The fields in this option frame are described in Table 5-14.

Table 5-14 Modem preferences option fields

Option Field	Description
`fConnectInDirectMode`	If `true`, forces modem to connect in direct mode (no speed buffering; DTE-DCE speed is set to match DCE-DCE speed). If `nil`, speed buffering is used if modem profile indicates the modem can support speed buffering. Default is `nil`.
`fIdModem`	If `true`, modem tool executes ID sequence in an attempt to identify the modem which is connected. If the modem is identified, the modem tool configures the active modem profile accordingly. The ID sequence is run when the `Bind` call is made to the modem tool. Note that the modem is reset during the ID sequence using the `AT&F` command. If `nil`, the modem tool skips the ID sequence and configures the active profile to the default. In this case, the modem is not reset. Default is `true`.
`fRequirePositiveId`	If `true`, the modem tool `Bind` will fail if the modem is not identified successfully. If `nil`, and the modem tool can not identify the modem, the default profile is used, and the `Bind` succeeds. Default is `nil`.
`fUseHardwareCD`	If `true`, the modem tool will sense the CD line for determining loss of carrier. External modems must use a cable that connects the CD RS-232 signal to the Newton GPi serial pin (pin 7 on MessagePads). If `nil`, CD is ignored. Default is `true`.
`fUseSoftwareCD`	Ignored.
`fUseConfigString`	If `true`, before initiating a connection, send the current configuration string to the modem (as determined by active modem profile and the connection type). If `nil`, no configuration string is sent. Default is `true`.

continued

Table 5-14 Modem preferences option fields (continued)

Option Field	Description
`fUseDialOptions`	If `true`, before initiating a connection, after the configuration string is sent to modem, set modem dialing configuration according to current option settings. If `nil`, dial configuration string is not sent to modem. Default dial config string: `ATM1L2X4S7=060S8=001S6=003\n`. Default is `true`.
`fHangUpAtDisconnect`	If `true`, when the modem tool disconnects, hang up the modem using the hang up sequence. If `nil`, when the modem tool disconnects, no commands are sent to the modem. Default is `true`.
`fEnablePassThru`	If `true`, modem tool connects/disconnects in pass through mode. In pass through mode, all modem controller functionality is disabled, and the modem tool behaves the same as a serial endpoint. If `nil`, modem tool controller is enabled. Normal modem tool operation. Default is `nil`.
`fEnableDialOutStream`	If `true`, enables dialing of the output stream. After connecting, all data output by modem tool client endpoint is sent to modem as dial commands. This feature can be used for interactive dialing. If `nil`, modem handles client endpoint output as normal data. Default is `nil`.
`fDirectModeSpeed`	Speed in bits per second (bps) at which modem tool begins direct mode connection. Default is 19200 bps.
`fHWCDDelayLowSpeed`	Amount of time, in seconds, which the CD line must be deasserted before considering the line disconnected. This value is used for connection speeds less than 2400 bps. Default is 3 seconds.
`fHWCDDelayHighSpeed`	Amount of time, in seconds, which the CD line must be deasserted before considering the line disconnected. This value is used for connection speeds greater than 2400 bps. Default is 15 seconds.

Modem Profile Option

A modem profile is a collection of information that describes the characteristics of a modem, to be used by the modem controller for configuring and connecting the modem. The modem profile option includes an indication of support for asynchronous speed buffering (if so, CTS flow control must be supported), support for special cellular configuration (e.g., signal attenuation), support for error control, connection speeds supported, the highest speed supported for the DTE-DCE interface, maximum command processing time, maximum characters per command line, minimum delay between commands, and the strings used to configure the modem for various types of connections.

Typically, individual applications do not need to set this option. The modem profile is set by the modem setup, enabling a particular modem to work with all applications that use a modem endpoint. Users pick the appropriate modem setup in the Modem Preferences. See Chapter 6, "Modem Setup Service," for information about how to write a modem setup package. If for some reason your application needs to customize the active modem profile, set this option in the configuration options for the endpoint. Also, set the `kCMOModemPrefs` option in the configuration options and disable the modem ID feature by setting `fIdModem` to `nil`.

Here is an example that shows the use of this option:

```
local modemID := "gonzo";
local ECnone := "ATC1";
local EConly := "ATC1";
local ECfall := "ATC1";
local ECcell := "ATC1";
local ECdirect := "ATC1";
local strSize := StrLen(modemID) + StrLen(ECnone) +
StrLen(EConly) +
   StrLen(ECfall) + StrLen(ECcell) + StrLen(ECdirect);
```

```
local option := {
   type: 'option,
   label:kCMOModemProfile,
   opCode:opSetRequired,
   form: 'template,// not needed
   data : {
      arglist: [
         true,    // supports Cellular
         true,    // supports EC
         true,    // supports LCS
         true,    // direct connect only
         1200,    // connect speeds
         1200,    // config speed
         2000,    // command response timeout
         40,         // max characters per command line
         25,         // inter-command delay
         strSize, // modem strings length
         modemID, // modem id string
         ECnone,  // config string no EC
         EConly,  // config string EC only
         ECfall,  // config string EX & fallback
         ECcell,  // config string EC cellular
         ECdirect,// config string direct connect
      ],
      typelist: [
         'struct,
         'boolean, // fSupportsCellular
         'boolean, // fSupportsEC
         'boolean, // fSupportsLCS
         'boolean, // fDirectConnectOnly
         'ulong, // fConnectSpeeds
         'ulong, // fConfigSpeed
         'ulong, // fCommandResponseTimeOut
```

```
            'ulong, // fMaxCharsPerCmdLine
            'ulong, // fInterCmdDelay
            'ulong, // fModemStringsLen
            ['array, 'char, 0], // fModemIDString
            ['array, 'char, 0], // fConfigStrNoEC
            ['array, 'char, 0], // fConfigStrECOnly
            ['array, 'char, 0], // fConfigStrECAndFallback
            ['array, 'char, 0], // fConfigStrCellular
            ['array, 'char, 0], // fConfigStrDirectConnect
        ]
    }
};
```

The fields in the modem profile option frame are described in Table 5-15.

Table 5-15 Modem profile option fields

Option field	Description
`fSupportsCellular`	If `true`, indicates modem profile contains an `fConfigStrCellular`. This string is used for cellular type data connections (e.g., turn on MNP 10). If `nil`, the modem profile does not contain an `fConfigStrCellular`. In this case, the normal data mode configuration string is used for cellular connections. Default is `nil`.
`fSupportsEC`	If `true`, indicates modem supports built-in error correction, and the profile contains configuration strings for error correction. Default is `nil`.

continued

Table 5-15 Modem profile option fields (continued)

Option field	Description
`fSupportsLCS`	If `true`, indicates modem supports line current sense. LCS is used for determining when a user has lifted the phone handset off hook. Applications take advantage of this feature by allowing the modem to determine when it should release the line for a voice call. If `nil`, the modem does not support LCS. In this case, an application can use a dialog box and user interaction to determine when to tell the modem to release the line (command `ATH`). Default is `nil`.
`fDirectConnectOnly`	If `true`, indicates modem only supports direct connect mode and can't support speed buffer. In this case, the DTE speed must be adjusted to the modem speed after carrier is established. If `nil`, indicates the modem supports speed buffering, and use of CTS flow control. Default is `true`.
`fConnectSpeeds`	Indicates speeds (in bps) at which modem can connect. This value does not affect modem configuration. The intention is for the application to read this value to determine the modem capabilities. The default value is 255, representing these speeds: 300, 1200, 2400, 4800, 7200, 9600, 12000, and 14400. Here are the bit flags, which are ORed together to yield the final value: `0x00000001` 300 bps `0x00000002` 1,200 bps `0x00000004` 2,400 bps `0x00000008` 4,800 bps `0x00000010` 7,200 bps `0x00000020` 9,600 bps `0x00000040` 12,000 bps `0x00000080` 14,400 bps `0x00000100` 16,800 bps `0x00000200` 19,200 bps `0x00000400` 21,600 bps `0x00000800` 24,000 bps `0x00001000` 26,800 bps `0x00002000` 29,000 bps `0x00004000` 31,400 bps

continued

Table 5-15 Modem profile option fields (continued)

Option field	Description
`fConfigSpeed`	Indicates speed at which to configure modem, in bps. Default is 19200.
`fCommandResponseTimeOut`	Indicates how long (in milliseconds) the modem command response state machine should wait for modem response to a command before timing out. Default is 2000.
`fMaxCharsPerCmdLine`	Indicates maximum number of characters per command line, not counting the `AT` prefix and the ending carriage return. The modem controller uses this number to ensure the dial string does not exceed the modem's capability. If the number of characters in the dial string exceeds this number, the dial string will be split into multiple commands, with a semicolon (;) appended to the intermediate dial string commands. Default is 40.
`fInterCmdDelay`	Indicates minimum amount of delay required between modem commands, in milliseconds. This is the time from last response received to next command sent. Default is 25.
`fModemStringsLen`	Indicates length of modem strings in the remainder of the fields for this option (packed together and null terminated). This value includes the termination characters.
`fModemIDString`	Modem response to the `ATI4` command. If the modem responds with more than one result string, `fModemIDString` should contain only one result string. Default is `unknown`.
`fConfigStrNoEC`	Modem command string used to configure modem for a non-error-corrected connection. Uses speed buffering. This string is used for FAX connections. Default is "ATE0&C1S12=12W2&K3&Q6\n".
`fConfigStrECOnly`	Modem command string used to configure the modem for an error corrected connection. Uses speed buffering. This string should be `nil` for modems that do not support error correction. Default is `nil`.

continued

Table 5-15 Modem profile option fields (continued)

Option field	Description
`fConfigStrECAndFallback`	Modem command string used to negotiate for error correction. If error-correction negotiation fails, the modem falls back to a non-error-corrected connection. Uses speed buffering. This string should be `nil` for modems that do not support error correction. Default is `nil`.
`fConfigStrCellular`	Modem command string used to configure the modem to connect over a cellular connection. This command should be used to turn on MNP 10 and power attenuation. Uses speed buffering. This string should be `nil` for modems that do not support error correction. Default is `nil`.
`fConfigStrDirectConnect`	Modem command string used to configure the modem to connect in direct mode. Speed buffering is disabled. After connecting in data mode, the DTE speed is adjusted to match the modem speed. Default is "ATE0&C1S12=12W2&K0&Q0\n".

Modem Error Control Type Option

This option specifies the type of error control protocol to use in the modem connection. More than one type of error control protocol can be specified, and the modem tool uses precedence to determine which type of error control protocol to use for the connection. The pseudo-code for determining error control precedence is as follows:

```
if (External EC is enabled) then
   begin
      if (No EC is enabled) then
         use fConfigStrECAndFallback
      else
         use fConfigStrECOnly
   end
else
```

```
begin
   /* use internal EC */
   attempt MNP connection
   if (MNP connection fails)
      begin
         if (No EC is enabled)
            fallback to normal connection
         else
            disconnect
      end
   else
      we're connected with MNP
end
```

Note
Cellular connections take precedence over external error control. In other words, if the connection type is `cellular`, as specified by the modem dialing option, see page 5-45, `fConfigStrCellular` is used even if external error control is enabled separately. ◆

This option should be set at or before the endpoint `Connect` call. Here is an example that shows the use of this option:

```
local option := {
   type: 'option,
   label:kCMOModemECType,
   opCode:opSetNegotiate,
   form: 'number,
   data : kModemECProtocolNone,
};
```

The possible values for the `data` slot are listed in Table 5-16. Note that these values can be ORed together to specify multiple error control types. The default is `kModemECProtocolMNP` and `kModemECProtocolNone` ORed together.

Table 5-16 Modem error control type

Constant	Value	Description
`kModemECProtocolNone`	`0x00000001`	No error control
`kModemECProtocolMNP`	`0x00000002`	Use internal MNP class 4
`kModemECProtocolExternal`	`0x00000008`	Use external modem's built-in error control
`kModemECInternalOnly`	`x0x00000010`	Connect with internal error control only; overrides other settings

Note
`kModemECProtocolNone`, `kModemECProtocolMNP`, and `kModemECProtocolExternal` had distinct meanings in previous versions of the system. Although their meanings are now enigmatic, they are maintained for backward compatibility. ◆

If you use "use error control" but don't set "no error control OK", you do not fall back to no error control because if error control can't be negotiated with the remote end, the connect/listen fails.

The Newton's internal error control is always available. If you select "use error control", and the modem you are using has built-in error control, the modem tool reconfigures itself to use the modem's built-in error control.

Modems that support built-in error control also support fall-back. The modem setups have two configuration strings for error control: one with fall-back to no error control, the other with error control only. If you have "use error control" set, one of these two strings is used; which string depends on the value of "no error control OK".

Modem Dialing Option

The modem dialing option is used to control the various parameters associated with dialing. This option should be set at or before the endpoint `Connect` call.

Rather than setting the modem dialing option manually, you should use the global function `MakeModemOption`. This method reads the user preferences and builds the modem dialing option frame for you. The `MakeModemOption` method is described on page 4-56 in Chapter 4, "Endpoint Interface."

Here is an example that shows the use of this option:

```
local option := {
   type: 'option,
   label:kCMOModemDialing,
   opCode:opSetRequired,
   form: 'template,// not needed
   data : {
      arglist: [
         true,    // speakeron
         true,    // detectdialtone
         true,    // detectbusy
         true,    // dtmftonedialing
         nil,     // manualdial
         2,          // speakervolume
         2,          // waitforcarrier in seconds
         2,          // waitforblinddial in seconds
         2,          // commadelay in seconds
         2,          // ringtoanswerafter in rings
         10,         // the country ID
         nil,        // use fConfigStrCellular
      ],
      typelist: [
         'struct,
         'boolean, // fSpeakerOn
         'boolean, // fDetectDialTone
         'boolean, // fDetectBusy
         'boolean, // fDTMFToneDialing
         'boolean, // fManualDial
         'char, // fSpeakerVolume
```

```
            'byte, // fWaitForCarrier
            'byte, // fWaitBeforeBlindDial
            'byte, // fCommaDelay
            'byte, // fRingToAnswerAfter
            'ulong, // fCountryId
            'boolean, // fCellularConnection
        ]
    }
};
```

The fields in the modem dialing option frame are described in Table 5-17.

Table 5-17 Modem dialing option fields

Option Field	Description
`fSpeakerOn`	If `true`, the modem speaker is turned on during the carrier establishment (`ATM1`). If `nil`, the speaker is off (`ATM0`). Default is `true`.
`fDetectDialTone`	If `true`, the modem detects and requires dial tone before dialing (`ATX4` or `ATX2`, depending on `fDetectBusy`). If `nil`, dial tone is not detected or required. In this case, the modem waits `fWaitBeforeBlindDial` seconds and then dials (`ATX3` or `ATX1`, depending on `fDetectDialTone`). Default is `true`.
`fDetectBusy`	If `true`, the modem detects the busy signal and reports this with the BUSY result (`ATX4` or `ATX3`, depending on the value of `fDetectDialTone`). If `nil`, the busy signal is ignored and the BUSY result code is not used (`ATX2` or `ATX1`, depending on value of `fDetectDialTone`). Default is `true`.
`fDTMFToneDialing`	If `true`, modem uses DTMF dialing (`ATDT`…). If `nil`, modem uses pulse dialing (`ATDP`…). Default is `true`.
`fManualDial`	If `true`, the modem goes off-hook to connect without dialing a number (e.g., `ATDT`). If `nil`, a phone number is required to originate a modem connection. Default is `nil`.

continued

Table 5-17 Modem dialing option fields (continued)

Option Field	Description
`fSpeakerVolume`	Modem speaker level. The value is used in the `ATLn` command. The acceptable values are defined as follows: `kSpeakerVolumeLow` "1" `kSpeakerVolumeMedium` "2" `kSpeakerVolumeHigh` "3" Note that these are one-character strings. Default is `kSpeakerVolumeMedium`.
`fWaitForCarrier`	Value used to set modem register S7. Units are seconds. Indicates amount of time modem waits to establish carrier after going off-hook. Default is 55.
`fWaitBeforeBlindDial`	Value used to set modem register S6. Units are seconds. Indicates amount of time modem waits after going off hook until dialing when dial tone is not required (when `fDetectDialTone` is `nil`). Default is 3.
`fCommaDelay`	Value used to set modem register S8. Units are seconds. Indicates length of pause in dialing when a comma occurs in the dial string. Default is 1.
`fRingToAnswerAfter`	Used when modem endpoint `Listen` call is made. The modem tool listens for an incoming call. Value is used to set modem register S0. Default is 2.
`fCountryId`	Indicates the current location of the user. This value is derived from the Time Zones setting. The following values are defined, based on the country codes: `kUSACountryId` 1 `kCanadaCountryId` 10 `kJapanCountryId` 81 Default is `kUSACountryId`.
`fCellularConnection`	Indicates that the `fConfigStrCellular` string from the Modem Profile Option should be used.

Take care because some modem setups are written exclusively for cellular modems, for example the "Moto Cellular" modem setup, see Chapter 6, "Modem Setup Service,". For this reason, set the value of `fCellularConnection`, to `true` only if you are also specifying your own modem profile that includes an `fCellularConnection` string

Modem Connection Type Option

The modem connection type option configures the modem endpoint for the type of connection desired. The modem tool distinguishes among voice connections, fax connections, data connections, and cellular data connections.

For voice connections, the modem tool acts as an auto-dialer. The modem is taken off-hook, the number is dialed, and the modem returns to command mode without attempting to establish the carrier.

For fax connections, the modem tool configures the modem in EIA/TIA 578 Service Class One mode; then Class One commands are used to send a fax.

For data connections, the modem tool configures and connects the modem according to the modem tool's current configuration (for example active modem profile, modem preferences).

When connecting, if more than one type of connection is enabled, the connection type of the highest precedence is initiated. Connection precedence is: voice (highest), then fax, then data (lowest).

When listening for a connection, voice takes precedence. If both data and fax are enabled, the type of connection is determined by the modem handshaking.

This option should be set at or before the endpoint `Connect` call.

Here is an example that shows the use of this option:

```
local option := {
   type: 'option,
   label:kCMOModemConnectType,
   opCode:opSetRequired,
   form: 'template,// not needed
```

```
   data : {
      arglist: [
         nil,     // voice enabled
         nil,     // fax enabled
         true,    // data enabled
         nil,     // reserved
         nil,     // immediate connection
      ],
      typelist: [
         'struct,
         'boolean, // fVoiceEnable
         'boolean, // fFaxEnable
         'boolean, // fDataEnable
         'boolean, // reserved
         'boolean, // fImmediate
      ]
   }
};
```

The fields in this option frame are described in Table 5-18.

Table 5-18 Modem connection type option fields

Option Field	Description
`fVoiceEnable`	If `true`, enables voice connection (auto dial with modem). Default is `nil`.
`fFaxEnable`	If `true`, enables fax connection. Default is `nil`.
`fDataEnable`	If `true`, enables data connection. Default is `true`.
`fImmediate`	If `true`, go off hook immediately after configuring modem. The dialing step (or when listening, the waiting for ring step) is skipped. Default is `nil`.

Modem Connection Speed Option

The modem connect speed option indicates modem-to-modem raw connection speed, in bps. This value is not a measure of throughput, which can vary because of compression, but instead is a measure of the raw bit rate of the modem-to-modem connection. This option is read only. The intended use is for determining modem connection speed, while the modem is connected. This option should be read only when the endpoint is in the connected state.

Here is an example that shows the use of this option:

```
local option := {
   type: 'option,
   label:kCMOModemConnectSpeed,
   opCode:opGetCurrent,
   form: 'number,
   data : 0,
};
```

Modem Fax Capabilities Option

The modem fax capabilities option indicates fax service class capabilities and modem modulation capabilities. This option is valid only after the endpoint `Bind` call.

If you use this option to *set* these capabilities, then the values constrain capabilities present in the modem. If you use this option to *get* the capabilities, then the values returned are the values for the modem ORed by the current values set with this option. By default the capabilities of the modem are not constrained.

Normally you don't need to set this option because the modem setup chosen by the user handles it. For example the "Moto Cellular" modem setup constrains the fax send/receive speed to 4800 baud. See Chapter 6, "Modem Setup Service," for information about how to write modem setup packages.

Here is an example that shows the sue of this option:

```
local option := {
   type: 'option,
   label:kCMOModemFaxCapabilities,
   opCode:opGetCurrent,
   form: 'template,// not needed
   data : {
      arglist: [
         0,
         0,
         0, //Returned
         0, //Returned
         0, //Returned
         0, //Returned
         0, //Returned
      ],
      typelist: [
         'struct,
         'ulong, // fServiceId
         'ulong, // fExtendedResult
         'ulong, // fServiceClass
         'ulong, // fTransmitDataMod
         'ulong, // fTransmitHDLCDataMod
         'ulong, // fReceiveDataMod
         'ulong, // fReceiveHDLCDataMod
      ]
   }
};
```

The fields in the modem fax capabilities option frame are described in Table 5-19.

Table 5-19 Modem Fax Capabilities Option Fields

Option field	Description
`fServiceClass`	Indicates which fax service classes are supported by the modem. The following service classes can be returned: *Constant* / *Value* / *Meaning* `kModemFaxClass0` `0x00000001` no fax service `kModemFaxClass1` `0x00000002` HDLC modulation `kModemFaxClass2` `0x00000004` T.30 modulation kmodemFaxClass2=0 0x0000000008 Approved class 2 spec
`fTransmitDataMod`	Indicates transmit modulations supported by the `AT+FTM=x` command. See Table 5-20 for possible return values. The array of possible values needs to be ORed together.
`fTransmitHDLCDataMod`	Indicates transmit HDLC modulations supported by the `AT+FTH=x` command. See Table 5-20 for possible return values. The array of possible values needs to be ORed together.
`fReceiveDataMod`	Indicates receive modulations supported by the `AT+FRM=x` command. See Table 5-20 for possible return values. The array of possible values needs to be ORed together.
`fReceiveHDLCDataMod`	Indicates receive HDLC modulations supported by the `AT+FRM=x` command. See Table 5-20 for possible return values. The array of possible values needs to be ORed together.

Fax modulation return values are described in Table 5-20.

Table 5-20 Modem Fax Modulation Return Values

Constant	Value	Description
kV21Ch2Mod	0x00000001	V.21 (300 bps)
kV27Ter24Mod	0x00000002	V.27 ter (2400 bps)
kV27Ter48Mod	0x00000004	V.27 ter (4800 bps)
kV29_72Mod	0x00000008	V.29 (7200 bps)
kV17_72Mod	0x00000010	V.17 (7200 bps)
kV17st_72Mod	0x00000020	V.17 short train (7200 bps)
kV29_96Mod	0x00000040	V.29 (9600 bps)
kV17_96Mod	0x00000080	V.17 (9600 bps)
kV17st_96Mod	0x00000100	V.17 short train (9600 bps)
kV17_12Mod	0x00000200	V.17 (12000 bps)
kV17st_12Mod	0x00000400	V.17 short train (12000 bps)
kV17_14Mod	0x00000800	V.17 (14400 bps)
kV17st_14Mod	0x00001000	V.17 short train (14400 bps)

Modem Voice Support Option

The modem voice support option is used to determine if the modem supports line current sense (LCS). If the modem is capable of supporting LCS, it automatically releases the phone line, by going on hook, when the user lifts the handset when a voice connection is made with the modem tool.

A modem which supports LCS ignores the `ATH0` command when auto dialing for a voice connection. Instead, it waits until it senses the current draw when the handset is lifted. If the active modem does not support LCS, the modem goes on hook when the modem endpoint `Disconnect` call

is made. If the user has not lifted the handset when the `Disconnect` call is made, the phone call is terminated. This option is read only and is valid only after the endpoint `Bind` call.

Here is an example of this option:

```
local option := {
   type: 'option,
   label:kCMOModemVoiceSupport,
   opCode:opGetCurrent,
   form: 'template,// not needed
   data : {
      arglist: [
         true,    // supports LCS
      ],
      typelist: [
         'boolean, // fSupportsLCS
      ]
   }
};
```

The single Boolean field in the `data` slot returns `true` if the modem supports LCS and `nil` if it does not.

MNP Speed Negotiation Option

This option controls the MNP speed negotiation. If you use this option before or when connecting, the modem tool negotiates with the remote end to change the data speed to the specified level. After connecting, you can determine the connection speed by getting the current value with the `kCMOMNPDataRate` option.

Generally, `kCMOSerialIOParms` should be used to set up things before connecting, and then `kCMOSerialBitRate` should be used to change speeds during later negotiations. It doesn't make sense to use both in the

same call; if you do, the speed ends up at the rate specified by the latter option in the option array.

The speed shift is negotiated in the link request (LR) packet and is fully backwards compatible; previous implementations that don't support this feature simply ignore the speed negotiation LR parameter.

Note
The MNP link request packets are sent at the original connect speed (set with either the `kCMOSerialIOParms` or `kCMOSerialBitRate` options). When the `kCMOMNPSpeedNegotiation` option is used, it negotiates the MNP data rate speed, and the serial port speed is set to this value. ◆

Here is an example that shows the use of this option:

```
local option := {
   type: 'option,
   label:kCMOMNPSpeedNegotiation,
   opCode:opSetNegotiate,
   form: 'template,// not needed
   data : {
      arglist: [
         57600,   // speed in bps
      ],
      typelist: [
         'struct,
         'long, // fSpeed
      ]
   }
};
```

The single integer field in the `data` slot specifies the desired data rate speed in bps. The default value is 57600.

MNP Compression Option

The MNP compression option is used to configure the data compression options in the modem tool. Data compression can only be supported on MNP connections. The modem tool supports V.42bis compression and MNP Class 5 compression.

The type of compression used during a connection must be negotiated with the remote connection end. If both V.42bis and MNP Class 5 compression types are enabled, the compression used for the connection is negotiated with the remote end. V.42bis compression is given top priority, followed by MNP Class 5. If neither compression can be used, the connection can be made with no compression. This option should be set at or before the endpoint `Connect` call.

Here is an example that shows the use of this option:

```
local option := {
   type: 'option,
   label:kCMOMNPCompression,
   opCode:opSetRequired,
   form: 'number,
   data : kMNPCompressionNone,  // no compression
};
```

The possible values for the `data` slot are listed in Table 5-21. Note that these values can be ORed together to specify multiple compression types. The default is all three values ORed together.

Table 5-21 MNP compression type

Constant	Value	Description
`kMNPCompressionNone`	`0x00000001`	No compression
`kMNPCompressionMNP5`	`0x00000002`	Use MNP class 5 compression
`kMNPCompressionV42bis`	`0x00000008`	Use V.42bis compression

MNP Data Statistics Option

The MNP data statistics option is a read-only option that reports some performance statistics from the current MNP connection. Valid results can be obtained by reading this option while the endpoint is being connected and after it is in the connected state.

Here is an example that shows the use of this option:

```
local option := {
   type: 'option,
   label:kCMOMNPStatistics,
   opCode:opGetCurrent,
   form: 'template,// not needed
   data : {
      arglist: [
         0,          // adapt value
         0,          // lt retrans count
         0,          // lr retrans count
         0,          // total retransmissions
         0,          // rcv broken total
         0,          // force ack total
         0,          // rcv async err total
         0,          // frames received
         0,          // frames transmitted
         0,          // bytes received
         0,          // bytes transmitted
         0,          // write bytes in
         0,          // write bytes out
         0,          // read bytes in
         0,          // read bytes out
         0,          // write flush count
      ],
```

```
      typelist: [
         'struct,
         'ulong, // fAdaptValue
         'ulong, // fLTRetransCount
         'ulong, // fLRRetransCount
         'ulong, // fRetransTotal
         'ulong, // fRcvBrokenTotal
         'ulong, // fForceAckTotal
         'ulong, // fRcvAsyncErrTotal
         'ulong, // fFramesRcvd
         'ulong, // fFramesXmited
         'ulong, // fBytesRcvd
         'ulong, // fBytesXmited
         'ulong, // fWriteBytesIn
         'ulong, // fWriteBytesOut
         'ulong, // fReadBytesIn
         'ulong, // fReadBytesOut
         'ulong, // fWriteFlushCount
      ]
   }
};
```

The fields in this option frame are described in Table 5-22.

Table 5-22 MNP data statistics option fields

Option Field	Description
`fAdaptValue`	Maximum size data packet when the connection supports adaptive packet sizing (Class 4). Default is 196.
`fLTRetransCount`	Number of times current data packet (LT) has been retransmitted. Default is 0.
`fLRRetransCount`	Retransmission count for connect packet (LR - link request). Default is 0.

continued

Table 5-22 MNP data statistics option fields (continued)

Option Field	Description
`fRetransTotal`	Total number of LT frame retransmissions during connection. Default is 0.
`fRcvBrokenTotal`	Total number of broken frames received during connection. Default is 0.
`fForceAckTotal`	Total number of forced acknowledgments during connection. Default is 0.
`fRcvAsyncErrTotal`	Total number of serial driver async errors (overruns) received during connection. Default is 0.
`fFramesRcvd`	Total number of frames received during connection. Default is 0.
`fFramesXmited`	Total number of frames transmitted during connection. Default is 0.
`fBytesRcvd`	Total number of data bytes received during connection. Includes packet header/tail. Default is 0.
`fBytesXmited`	Total number of data bytes transmitted during connection. Includes packet header/tail. Default is 0.
`fWriteBytesIn`	Total number of user data bytes transmitted during connection (before compression). Default is 0.
`fWriteBytesOut`	Total number of user data bytes transmitted during connection (after compression). Default is 0.
`fReadBytesIn`	Total number of user data bytes received during connection (before decompression). Default is 0.
`fReadBytesOut`	Total number of user data bytes received during connection (after decompression). Default is 0.
`fWriteFlushCount`	Number of flush calls to V.42bis compressor during connection. Default is 0.

Infrared Tool

This section describes the infrared (IR) communication tool.

The infrared tool permits only half-duplex communication. If you are using the infrared communication tool, do not activate an input spec and expect to output data.

Because the infrared tool supports packetized data, an endpoint can include `kPacket` and `kEOP` and `kMore` flags to control sending and receiving framed (packetized) data. For more information on these flags, see "Sending Data" beginning on page 4-17 of Chapter 4, "Endpoint Interface."

Here's an example of how to create an IR endpoint:

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

The infrared tool supports some options, which are summarized in Table 5-46 and described in detail in the following subsections.

Table 5-23 Summary of Infrared Options

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

Infrared Connection Option

This option to controls how the infrared connection is made. You can set it in the `Instantiate`, `Bind`, or `Connect` methods.

Here is an example that shows the use of this option:

```
local option := {
   label: kCMOSlowIRConnect,// "irco"
   type: 'option,
   opCode: opSetNegotiate,
   data: {
      arglist: [ connect ],
      typelist: [ 'ulong ]
   }
};
```

The `connect` field is interpreted as a series of bit flags. The `connect` field bit flags are as follows:

Constant	Value	Description
`kNormalConnect`	0	Normal connection, if set
`irSymmetricConnect`	1	Allows symmetric connection, if set
`irActiveConnection`	2	This bit is set by the infrared tool to indicate the type of connection made

The `kNormalConnect` constant indicates that the infrared tool should connect normally as controlled by the use of the `Connect` or `Listen` calls. If you use the `Connect` call, the tool connects in active mode, expecting the remote device to be listening passively. If you use the `Listen` call, the tool connects in passive mode, expecting the remote device to be connecting actively.

The `irSymmetricConnect` constant indicates that the tool should open in symmetric mode; that is, the `Connect` call can act either as a `Connect` or a `Listen`, depending on what the remote side is doing. If the remote side is also attempting to open an active connection (via `Connect`) then the local side opens as if `Listen` had been called instead.

Client applications can determine in which state the tool was actually opened by looking at the second bit, `irActiveConnection`. If this bit is set, the tool opened as the active side (`Connect`). If this bit is cleared, the tool opened as the passive side (`Listen`).

Infrared Protocol Type Option

The infrared protocol type option is a read-only option that reports the protocol and speed of the current infrared connection. Valid results can be obtained by reading this option during or after the endpoint is in the connected state, or while a connection is being negotiated.

Here is an example of this option:

```
local option := {
   label: kCMOSlowIRProtocolType,// "irpt"
   type: 'option,
   opCode: opGetCurrent,
   data: {
      arglist: [
         protocol,
         options
      ],
      typelist: [
         'ulong,
         'ulong
      ]
   }
};
```

The possible values for the `protocol` field are as follows:

Constant	Value	Description
`kUsingNegotiateIR`	0	The tool is negotiating a connection using the negotiation protocol (Sharp protocol with Apple extensions). No connection has been made.
`kUsingSharpIR`	1	A connection has been made to a Sharp OZ/IQ or similar device using the standard Sharp protocol.
`kUsingNewton1`	2	A connection has been made to a Newton 1.x device using the Sharp protocol with Apple extensions.
`kUsingNewton2`	4	A connection has been made to a Newton 2.x device using the Sharp protocol with Apple extensions.

The possible values for the `options` field are as follows:

Constant	Value	Description
`kUsing9600`	1	Connection speed is 9600 bps
`kUsing19200`	2	Connection speed is 19200 bps
`kUsing38400`	4	Connection speed is 38400 bps

The infrared tool uses the Sharp Infrared protocol. Because of the characteristics of this protocol, we suggest setting `sendFlags` to `kPacket` and to `kEOP` every time you send data. For more information on `sendFlags` see, "Output Spec Frame" beginning on page 4-36 of Chapter 4, "Endpoint Interface." If you don't set `sendFlags` to this value, the protocol only sends after 512 bytes of data are queued up. This queuing means input scripts do not terminate when you expect them to. For the receiving side, the queuing means you will terminate after every output if you set `useEOP` to `true`. If you are using `byteCount`, you should set `useEOP` to `nil` if you want to trigger on `byteCount` instead of `EOP`. For more information on `useEOP` and `byteCount`, see "Input Spec Termination Frame" beginning on page 4-41 of Chapter 4, "Endpoint Interface."

Infrared Statistics Option

The infrared statistics option is a read-only option that reports various statistics on the current infrared connection. Valid results can be obtained by reading this option during or after the endpoint is in the connected state.

Here is an example that shows the use of this option:

```
local option := {
   label: kCMOSlowIRStats,// "irst"
   type: 'option,
   opCode: opGetCurrent,
   data: {
```

```
      arglist: [
         dataPacketsIn,
         checkSumErrs,
         dataPacketsOut,
         dataRetries,
         falseStarts,
         serialErrs,
         protocolErrs
      ],
   typelist: [
      'ulong,
      'ulong,
      'ulong,
      'ulong,
      'ulong,
      'ulong,
      'ulong
      ]
   }
};
```

The fields in the infrared statistics option frame are described in Table 5-24.

Table 5-24 Infrared statistics option fields

Option Field	Description
`dataPacketsIn`	Number of data packets received
`checkSumErrs`	Number of checksum errors in received packets
`dataPacketsOut`	Number of data packets sent
`dataRetries`	Number of retries performed while sending
`falseStarts`	Not used

continued

Table 5-24 Infrared statistics option fields (continued)

Option Field	Description
`serialErrs`	Number of bytes with parity or framing errors or serial chip buffer overruns
`protocolErrs`	Number of unexpected or out-of-sequence packets. These can occur because a packet was garbled in transmission and the two sides became unsynchronized.

AppleTalk Tool

The AppleTalk tool enables access to the ADSP (Apple Data Stream Protocol), ZIP (Zone Information Protocol), and NBP (Name Binding Protocol) components of the AppleTalk protocol stack.

Here's 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, 4]
                  ]
   } ];
results := myATalkEP:Instantiate(myATalkEP, myOptions);
```

Note that you do not need to call `OpenAppleTalk` or `CloseAppleTalk` to use AppleTalk. The AppleTalk endpoint automatically opens and closes the drivers for you when the endpoint methods `Bind` and `UnBind` are called, respectively.

The AppleTalk tools supports some options, which are summarized in Table 5-46 and described in detail in the following subsections.

Table 5-25 Summary of AppleTalk options

Label	Value	Use When	Description
`kCMARouteLabel`	"rout"	When connecting	Sets an AppleTalk NBP address
`kCMOAppleTalkBuffer`	"bsiz"	When connecting	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

AppleTalk Address Option

During the connection phase you'll need to supply an address option that gives an AppleTalk NBP address.

Here is an example that shows the use of this option:

```
local NBPStr := "PrinterName:Laserwriter@zone";// address
local size := StrLen(NBPStr);

local opt := {
   label:kCMARouteLabel,
   type: 'address,
   opCode:opSetRequired,
   data: {
      arglist: [
         kNamedAppleTalkAddress, // type
         kNamedAppleTalkAddress, // named addr type
         kDefaultLink,           // = "sltk"
         size,                   // length
         NBPStr,                 // NBP string
      ],
      typelist: [
         'struct,
         'long,
         'long,
         ['array, 'char, 4],
         'ulong,
         ['array, 'unicodeChar, 0],
      ]
   }
};
```

Note you need to set the size and NBPSX's where "size" is the length in bytes of the "NOP string" you are passing.

Note also that you must pass the `kCMARouteLabel` option in to ADSP's `Connect` and `Listen` options. For `Connect`, it's who you're connecting to; for Listen, it's who you are.

Alternatively, to construct an address option, you can call the global function `MakeAppletalkOption` instead. For more information on `MakeAppletalkOption`, see page 4-56 in Chapter 4, "Endpoint Interface."

AppleTalk Buffer Size Option

You can supply a buffer size option to specify the sizes of the send, receive, and attention buffers. You must specify a separate option per buffer type.

This option can be set in conjunction with the `Connect` and `Listen` methods only.

The buffer types are identified by integers, as follows:

Buffer type	Identifier	Default size
Send	`kSndbuffer`	511
Receive	`kRevBuffer`	511
Attention	`kAtnBuffer`	0

For example:

```
local opt := {
   label:kCMOAppleTalkBuffer,
   type: 'option,
   opCode:opSetRequired,
   data: {
      arglist: [
         kBufferType, // buffer type (kSndbuffer,
                        kRevBuffer, or kAtnBuffer)
         kSize, // buffer size in bytes
      ],
```

```
      typelist: [
         'struct,
         'ulong,
         'long,
      ]
   }
};
```

AppleTalk Bytes Available Option

This read-only option returns the number of bytes waiting to be read from the receive buffer.

The AppleTalk bytes available option is used after the endpoint is connected.

Here is an example that shows the use of this option:

```
local option := {
   type: 'option,
   label:kCMOSerialBytesAvailable,
   opCode:opGetCurrent,
   form: 'number,
   result: nil,// not needed; returned
   data: 0, // returned
};
```

AppleTalk Functions

A number of global functions are provided for obtaining the addresses of other devices on the network.

Opening and Closing the AppleTalk Drivers

It is not necessary to call `OpenAppleTalk` in order to access zone information because all other functions open AppleTalk if necessary, but you may want to do so for efficiency's sake. In that case you can do many more

lookups or make other functions calls without the overhead of repeated openings and closings of AppleTalk.
In order to access zone information, you need to first open the AppleTalk drivers with `OpenAppleTalk`. When you're finished, be sure to close the drivers with `CloseAppleTalk`.

Note that if you are using an AppleTalk endpoint, the AppleTalk drivers are opened automatically when the endpoint `Bind` method is called, and are closed when the endpoint `UnBind` method is called.

OpenAppleTalk

```
OpenAppleTalk()
```

Opens the AppleTalk drivers, returning zero if successful. You may call `OpenAppleTalk` as many times are you like, but you must call `CloseAppleTalk` at least as many times as you call `OpenAppleTalk`. If AppleTalk is open, `OpenAppleTalk` simply increments a counter and returns.

CloseAppleTalk

```
CloseAppleTalk()
```

Closes the AppleTalk drivers, returning zero if successful. You can call `CloseAppleTalk` as many times as you like. If AppleTalk is not open, this call does nothing. If the open counter is 1, this call closes the AppleTalk drivers; otherwise it decrements the open count. It is not necessary to call `CloseAppleTalk` in order to access zone information because all other functions open AppleTalk if necessary.

AppleTalkOpenCount

```
AppleTalkOpenCount()
```

Returns the open count for the AppleTalk drivers. A return value of zero means the drivers are closed.

Obtaining Zone Information

There are several functions for obtaining zone information, which are described in the following sections.

HaveZones

```
HaveZones()
```

Returns `true` if a connection exists and zones are available. Returns `nil` if there are no zones. This function automatically opens and closes the AppleTalk drivers as needed, so you don't need to call `OpenAppleTalk` or `CloseAppleTalk`.

GetMyZone

```
GetMyZone()
```

Returns a string naming the current AppleTalk zone. A return value of "*" identifies the default zone, usually meaning that no AppleTalk router was found. This function automatically opens and closes the AppleTalk drivers as needed, so you don't need to call `OpenAppleTalk` or `CloseAppleTalk`.

GetZoneList

```
GetZoneList()
```

Returns an array containing strings of all the existing zone names, or returns `nil` if no zones are available (which usually means no router is visible). This function automatically opens and closes the AppleTalk drivers as needed, so you don't need to call `OpenAppleTalk` or `CloseAppleTalk`.

GetNames

`GetNames(`*fromWhat*`)`

Returns a string or an array of names based on the *fromWhat* parameter.

fromWhat A network address in the form `name:type@zone`.

If *fromWhat* is a string, `GetNames` returns a string; if *fromWhat* is an array, `GetNames` returns an array of names.

Here's an example that shows the use of this function:

```
userconfiguration.currentPrinter.printerName
#4415501  "Idiot Savante:LaserWriter@RD1/
NewHaven-LocalTalk"

GetNames(userconfiguration.currentPrinter.printerName)
#4417791  "Idiot Savante"
```

GetZoneFromName

`GetZoneFromName(`*fromWhat*`)`

Returns the zone name as a string based on the *fromWhat* parameter

fromWhat A network address in the form `name:type@zone`.

For example:

```
GetZoneFromName(userconfiguration.currentPrinter.
printerName)
#44184A9  "RD1/NewHaven-LocalTalk"
```

NBPStart

`NBPStart(`*entity*`)`

Begins a lookup of network entities, as specified by the *entity* parameter. This function automatically opens the AppleTalk drivers, so you don't need to first call `OpenAppleTalk`.

This function returns a lookup ID that is used with the other NBP functions described in this section. This function returns `nil` if the lookup can't be started.

entity A string specifying the type of entity to search for and the zones in which to search. The entity string must have the form "*name*`:`*type*`@`*zone*". You can use the wild card characters "*" to identify the local zone, "=" to match all strings, and "≈" (Option–x) to match a

partially specified string. For example "=:≈Laser≈@*" searches for all entities whose type contains the string "Laser" in the local zone. The characters ":" and "@" are reserved, to separate the name from the type, and the type from the zone, respectively.

To get the names of the entities that are found, use the `NBPGetNames` function. For example, to look for LaserWriters in the current zone, use this code:

```
lookupID := NBPStart("=:LaserWriter@*");
NBPGetNames(lookupID);
```

NBPGetCount

`NBPGetCount(`*lookupID*`)`

Returns the number of entities the currently running NBP lookup has found.

lookupID The lookup ID returned by the `NBPStart` function used to start this lookup.

Here is an example of using this function:

```
lookupID := NBPStart("=:LaserWriter@*");
NBPGetCount(lookupID);
```

NBPGetNames

`NBPGetNames(`*lookupID*`)`

Returns an array of strings that are the names found by `NBPStart`.

lookupID The lookup ID returned by the `NBPStart` function used to start this lookup.

For an example of using this function, see `NBPStart`.

NBPStop

`NBPStop(`*lookupID*`)`

Terminates a lookup started by `NBPStart`, returning zero if successful, or -10067 if not. `NBPStop` automatically closes the AppleTalk drivers.

lookupID The lookup ID returned by the `NBPStart` function used to start this lookup.

NetChooser Function

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

NetChooser:OpenNetChooser

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

This function displays the chooser view.

zone A string identifying the pre-defined AppleTalk zone; specify `nil` for the current zone.

lookupName A string identifying the name of the entity to be looked up.

startSelection A string identifying the name of an entity to be selected as the default.

who An identifier naming the context that does the notification; `self` specifies the current context.

connText The string to be placed in the button that will select the service.

headerText The string to be placed in the title string, and also in possible notifications.

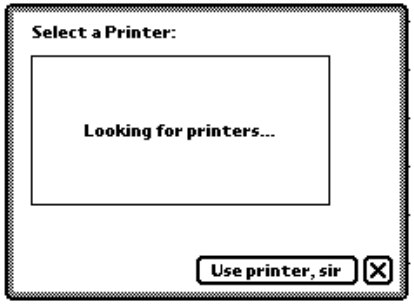
lookforText A string that informs the user what the chooser is trying to find. The *lookforText* is appended to the string `"Looking for"`. This string appears while the list is being assembled, or when the user chooses Change Zone.

Here's an example that shows the use of this function:

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

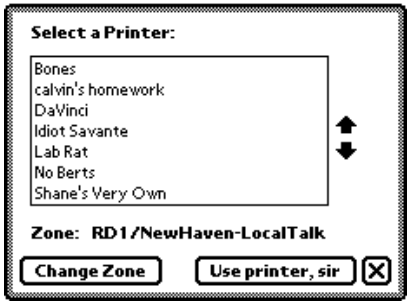
This example opens the NetChooser view and displays the *lookforText* string while the search is in progress, see Figure 5-2.

Figure 5-2 NetChooser view while searching



When the search has been completed, the NetChooser fills in the available choices, see Figure 5-3.

Figure 5-3 NetChooser view displaying printers



NetworkChooserDone

To obtain the user's selection, you need to provide a method called `NetworkChooserDone`. This method should have the following format:

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

currentSelection Returns the selected entity

currentZone Returns the currently selected AppleTalk zone.

Here's an example that shows the use of this function:

```
ChooserSample := {
// open our network connection
openNetworkScript: func()
begin
GetRoot().NetChooser:openNetChooser(nil,"=:LaserWriter@",
nil, 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
};

ChooserSample:OpenNetworkScript()
#1A       TRUE

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

Resource Arbitration Options

You can construct a communication tool to share its resource with other communication 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 communication tool.

The communication 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 communication tool is claiming its resources passively or actively. If this value is `true`, the communication tool is claiming its resources passively and will allow another tool to claim it. If this value is `false`, the communications tool is claiming its resources actively and will not allow another application to claim it.
- The resource-passive state option (`kCMOPassiveState`) has a Boolean value that specifies whether or not the current state of the communication 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 `false`, the communication tool is not willing to relinquish use of its passively claimed resources.

Table 5-26 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

Here are two examples that show the use of these options. The first demonstrates claiming the tool passively. Passive claiming must take place before binding. By default all tools are claimed actively.

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

The second is an example of using the tool passively. For instance, if you are listening on a tool you may pass this option down specifying an `arglist` value of `true`. By default tools are in an active state.

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

],
```

Summary

Constants and Variables

Table 5-27 Serial chip specification option constants

Constant	Value	Description
Parity Support Constants		
`kSerCap_Parity_Space`	0x00000001	No parity
`kSerCap_Parity_Mark`	0x00000002	Mark parity
`kSerCap_Parity_Odd`	0x00000004	Odd parity
`kSerCap_Parity_Even`	0x00000008	Even parity

continued

Table 5-27 Serial chip specification option constants

Constant	Value	Description
Data and Stop Bits Support Constants		
`kSerCap_DataBits_5`	0x00000001	5 data bits
`kSerCap_DataBits_6`	0x00000002	6 data bits
`kSerCap_DataBits_7`	0x00000004	7 data bits
`kSerCap_DataBits_8`	0x00000008	8 data bits
`kSerCap_StopBits_1`	0x00000010	1 stop bit
`kSerCap_StopBits_1_5`	0x00000020	1.5 stop bits
`kSerCap_StopBits_2`	0x00000040	2 stop bits
`kSerCap_StopBits_All`	0x00000070	Supports all stop bit choices
`kSerCap_DataBits_All`	0x0000000F	Supports all data bit choices
Serial chip types		
`kSerialChip8250`	0x00	8250 UART
`kSerialChip16450`	0x01	16450 UART
`kSerialChip16550`	0x02	16550 UART
`kSerialChip8530`	0x20	8530 UART (SCC chip)
`kSerialChip6850`	0x21	6850 UART (Brick ASIC modem port UART)
`kSerialChip6402`	0x22	6402 UART (Brick ASIC infrared port UART)
`kSerialChipVoyager`	0x23	Cirrus Voyager UART chip
`kSerialChipUnknown`	0x00	Unknown type of UART

Table 5-28 Serial circuit control option constants

Constant	Value	Description
Serial Output Lines		
`kSerOutDTR`	0x01	DTR line
`kSerOutRTS`	0x02	RTS line (also known as HSKo on the external serial port)
Serial Input Lines		
`kSerInDSR`	0x02	DSR line
`kSerInDCD`	0x08	DCD line (also known as GPi on the external serial port)
`kSerInRI`	0x10	RI line (also known as GPi on the external serial port)
`kSerInCTS`	0x20	CTS line (also known as HSKi on the external serial port)
`kSerInBreak`	0x80	

Table 5-29 Stop bits field constants

Constant	Value	Description
`k1StopBits`	0	1 stop bit (default)
`k1pt5StopBits`	1	1.5 stop bits
`k2StopBits`	2	2 stop bits

Table 5-30 Parity field constants

Constant	Value	Description
`kNoParity`	0	No parity (default)
`kOddParity`	1	Odd parity
`kEvenParity`	2	Even parity

Table 5-31 Data bits constants

Constant	Value (Number of data bits)
`k5DataBits`	5
`k6DataBits`	6
`k7DataBits`	7
`k8DataBits`	8 (default)

Table 5-32 Field interface speed constants

Constant	Value
`kExternalClock`	1
`k300bps`	300
`k600bps`	600
`k1200bps`	1200
`k2400bps`	2400
`k4800bps`	4800
`k7200bps`	7200
`k9600bps`	9600 (default)

continued

Table 5-32 Field interface speed constants

Constant	Value
`k12000bps`	12000
`k14400bps`	14400
`k19200bps`	19200
`k38400bps`	38400
`k57600bps`	57600
`k115200bps`	115200
`k230400bps`	230400

Table 5-33 Serial event constants

Constant	Value	Description
`kSerialEventBreakStartedMask`	0x00000001	A serial line break condition is detected
`kSerialEventBreakEndedMask`	0x00000002	A serial line break condition ends
`kSerialEventDCDNegatedMask`	0x00000004	The DCD line is negated (DCD is also known as GPi in the external serial port)
`kSerialEventDCDAssertedMask`	0x00000008	The DCD line is asserted
`kSerialEventHSKiNegatedMask`	0x00000010	The CTS line is negated (CTS is also known as HSKi in the external serial port)
`kSerialEventHSKiAssertedMask`	0x00000020	The CTS line is asserted
`kSerialEventExtClkDetectEnableMask`	0x00000040	The serial tool detects more than 100 transitions per second on the CTS line, and thus assumes this line is a clock input

Table 5-34 Data slot constants:

Constant	Value	Description
kSerClk_Default	0x00	Use the default
kSerClk_DivideBy_1	0x80	Divide by 1
kSerClk_DivideBy_16	0x81	Divide by 16
kSerClk_DivideBy_32	0x82	Divide by 32
kSerClk_DivideBy_64	0x83	Divide by 64

.

Table 5-35 Modem error control type

Constant	Value	Description
kModemECProtocolNone	0x00000001	No error control
kModemECProtocolMNP	0x00000002	Use internal MNP class 4
kModemECProtocolExternal	0x00000008	Use external modem's built-in error control

Table 5-36 Modem service type constants

Constant	Value	Description
kModemFaxClass0	0x00000001	No fax service
kModemFaxClass1	0x00000002	HDLC modulation
kModemFaxClass2	0x00000004	T.30 modulation

Table 5-37 Modem fax modulation return values

Constant	Value	Description
kV21Ch2Mod	0x00000001	V.21 (300 bps)
kV27Ter24Mod	0x00000002	V.27 ter (2400 bps)
kV27Ter48Mod	0x00000004	V.27 ter (4800 bps)
kV29_72Mod	0x00000008	V.29 (7200 bps)
kV17_72Mod	0x00000010	V.17 (7200 bps)
kV17st_72Mod	0x00000020	V.17 short train (7200 bps)
kV29_96Mod	0x00000040	V.29 (9600 bps)
kV17_96Mod	0x00000080	V.17 (9600 bps)
kV17st_96Mod	0x00000100	V.17 short train (9600 bps)
kV17_12Mod	0x00000200	V.17 (12000 bps)
kV17st_12Mod	0x00000400	V.17 short train (12000 bps)
kV17_14Mod	0x00000800	V.17 (14400 bps)
kV17st_14Mod	0x00001000	V.17 short train (14400 bps)

Table 5-38 MNP compression type

Constant	Value	Description
kMNPCompressionNone	0x00000001	No compression
kMNPCompressionMNP5	0x00000002	Use MNP class 5 compression
kMNPCompressionV42bis	0x00000008	Use V.42bis compression

Table 5-39 The protocol field constants

Constant	Value	Description
`kUsingNegotiateIR`	0	The tool is negotiating a connection using the negotiation protocol (Sharp protocol with Apple extensions). No connection has been made.
`kUsingSharpIR`	1	A connection has been made to a Sharp OZ/IQ or similar device using the standard Sharp protocol.
`kUsingNewton1`	2	A connection has been made to a Newton 1.x device using the Sharp protocol with Apple extensions.
`kUsingNewton2`	4	A connection has been made to a Newton 2.x device using the Sharp protocol with Apple extensions.

Table 5-40 The options field constants:

Constant	Value	Description
`kUsing9600`	1	Connection speed is 9600 bps
`kUsing19200`	2	Connection speed is 19200 bps
`kUsing38400`	4	Connection speed is 38400 bps

Functions and Methods

AppleTalk Functions

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

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*`)`

Registration Methods

`GetCommConfig(`*name*`, `*type*`, `*serviceId*`)`
`RegCommConfig(`*id*`, `*configuration*`)`
`UnRegCommConfig(`*id*`)`

Options

Table 5-41 Summary of serial options

Label	Value	Use When	Description
`kCMOSerialHWChipLoc`	"schp"	Before binding	Sets which serial hardware to use
`kCMOSerialChipSpec`	"sers"	Any time	Sets which serial hardware to use and returns information about the serial hardware
`kCMOSerialCircuitControl`	"sctl"	After connecting	Controls usage of the serial interface lines

continued

Table 5-41 Summary of serial options (continued)

Label	Value	Use When	Description
kCMOSerialBuffers	"sbuf"	Before connecting	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
kCMOSerialBreak	"sbrk"	After connecting	Sends a break
kCMOSerialDiscard	"sdsc"	After connecting	Discards data in input and/or output buffer
kCMOSerialEventEnables	"sevt"	After connecting	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

Table 5-42 Summary of serial with MNP options

Label	Value	Use When	Description
kCMOMNPCompression	"mnpc"	After connecting, before running	Sets the data compression type (see page 5-57)
kCMOMNPDataRate	"eter"	Any time	Configures internal MNP timers

Table 5-43 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

Table 5-44 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
`kCMOModemVoiceSupport`	"mvso"	After bind, before connecting	Read-only option indicating if the modem supports line current sense (LCS)

continued

Table 5-44 Summary of modem options (continued)

Label	Value	Use When	Description
`kCMOMNPSpeedNegotiation`	"mnpn"	Any Time	Sets MNP data rate speed
`kCMOMNPCompression`	"mnpc"	After connecting, before running	Sets the data compression type
`kCMOMNPStatistics`	"mnps"	After connecting	Read-only option reporting performance statistics from the current MNP connection

Table 5-45 Summary of infrared options

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

Table 5-47 Resource arbitration options

Label	Value	Use When	Description
`kCMOPassiveClaim`	"cpcm"	Any time	Specifies whether your tool claims resources actively or passively
`kCMOPassiveState`	"cpst"	Any time	Specifies whether your tool releases resources

Table 5-46 Summary of AppleTalk options

Label	Value	Use When	Description
`kCMARouteLabel`	"rout"	When connecting	Sets an AppleTalk NBP address
`kCMOAppleTalkBuffer`	"bsiz"	When connecting	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

Modem Setup Service

This chapter contains information about the modem setup capability in Newton system software. You should read this chapter if you need to define a modem setup package for your application. You will also want to refer to chapter 5 “Built-in Communication Tools” for information on the Modem Tool constants discussed here.

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, methods, and functions you use in defining a modem setup.

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 device. 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. See "Modem Preferences Option" beginning on page 5-34 for full details.
- **A modem tool profile option.** This part of the package describes the characteristics of the modem—for example, whether the modem supports cellular data connections, error correction protocols, and so forth. For

more information on this option, see the section “Modem Profile Option” beginning on page 5-38)

- **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.

Because of the way the modem setups are used and the way parameters are defined, separate profiles need to be made for cellular and landline operations if a modem supports both—unless the modem automatically senses and configures itself. 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.

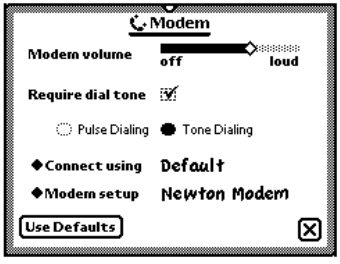
Note

The constants and code shown in this chapter apply to the NTK project which is provided by 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 6-1. The Modem Setup item is a picker, when tapped it, displays all of the modem setups installed in the system. The chosen modem setup is the default used by all applications.

Figure 6-1 Modem preferences view



The Modem Setup Process

All communication applications that use a modem endpoint make use of the modem setup service. When a modem endpoint `Instantiate` call is made, but before the `Bind` and `Connect` methods have completed, the current modem setup is invoked.

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. Next, the modem endpoint is instantiated and connected in pass-through mode.
3. The system sets the modem preferences (`kCMOModemPrefs`) and modem profile (`kCMOModemProfile`) 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 `Bind` and `Connect` calls.

"Defining a Modem Setup" on page 6-6 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.
- Hardware flow control is expected 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. 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 at least 19200.

Defining a Modem Setup

The parts of a modem setup are specified in an 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 subsections 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 version of the modem setup package. The constant `kOrganization` indicates the source of the modem setup package. See Table 6-3 on page 6-10 for complete details on these constants.

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;
```

Table 6-4 on page 6-11 describes these constants. For additional information about these items, see the section "Modem Preferences Option" beginning on page 5-34.

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 kConfigStrECAndFallback :=
   "ATE0&A0&B1&C1&H1&M4S12=12\n";
constant kConfigStrDirectConnect :=
   "ATE0&A0&B0&C1&H0&M0S12=12\n";
```

Table 6-5 on page 6-12 describes these constants. For additional information about these items, see the section "Modem Profile Option" beginning on page 5-38.

When the modem tool establishes communication with the modem through an endpoint, a configuration string is normally sent 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 6-1 summarizes when each kind of configuration string is used:

Table 6-1 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.
`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.
`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.
`kConfigStrDirectConnect`	The default configuration used for data connections when `kDirectConnectOnly` is `true`.

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;
```

on page 6-19 describes these constants.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.

The speed at which faxes are sent and received are specified by configuration strings. Table 6-2 lists the strings available for these two constants.

Table 6-2 Available fax speeds

Configuration string	Value	Bits per second
`kV21Ch2Mod`	0x00000001	300
`kv27Ter24Mod`	0x00000002	2400
`kV27Ter48Mod`	0x00000004	4800
`kV29_72Mod`	0x00000008	7200
`kV17_72Mod`	0x00000010	7200
`kV17st_72Mod`	0x00000020	7200
`kV29_96Mod`	0x00000040	9600
`kV17_96Mod`	0x00000080	9600
`kV17st_96Mod`	0x00000100	9600
`kV17_12Mod`	0x00000200	12000
`kV17st_12Mod`	0x00000400	12000
`kV17st_14Mod`	0x00001000	14400

Modem Setup Service Reference

This section describes the constants and methods used by the modem setup service.

Constants

The following sections contain descriptions of the constants you use in the general information, preferences, and profile parts of the modem setup.

Modem Setup General Information

The following constants specify general information about the modem setup.

Table 6-3 Constants for modem setup general information

Constant	Description
`kModemName`	A string consisting of the name that identifies this modem setup shows up in the Modem Preferences picker. Typically this is the name of the modem.
`kVersion`	An integer identifying the version number of this modem setup package. The system prevents a modem setup package with an equivalent or lower version number from overwriting one with a higher version number that is already installed on a Newton.
`kOrganization`	A string indicating the developer of the modem setup package.

Modem Setup Preferences

The following constants specify the modem setup preferences for configuring the modem controller.

Table 6-4 Constants for modem setup preferences

Constant	Description
`kidModem`	Set to `nil` to prevent the modem tool from executing a modem ID sequence and automatically setting the modem profile.
`kuseHardwareCD`	This is generally set to `true` for PCMCIA modems. For serial modems, a setting of `true` requires a special cable that connects the CD signal from the modem to the GPi serial pin on the Newton. A setting of `true` causes the modem tool to sense the CD line to detect loss of carrier. If this constant is set to `nil`, the CD line is ignored.
`kuseConfigString`	Set this to `true`, unless the modem happens to be configured correctly when it is reset, which is very unlikely. A setting of `true` means that a modem configuration string is to be sent to the modem before initiating a connection. The modem configuration string is defined in the modem profile option and depends on the connection type. If this constant is set to `nil`, no modem configuration string is sent.
`kuseDialOptions`	Set this to `true` to send the default dialing configuration string to the modem, following the configuration string. The default dialing configuration string is `ATM1L2X4S7=060S8=001S6=003\n`. If you specify `nil`, the dialing configuration string is not sent to the modem.
`khangUpAtDisconnect`	Set this to `true`. This setting causes a “clean” hang-up sequence to occur when the modem disconnects. If this constant is set to `nil`, no hang-up commands are sent to the modem on disconnect.

Modem Setup Profile Constants

The modem profile constants describe the modem characteristics, which are used by the modem controller.

Note:
Where the backslash (\) is used in a configuration string, you must specify two of them together (\\), since a single backslash is used as the escape character in NewtonScript. ◆

Table 6-5 Constants for the modem setup profile

Constant	Description
`kSupportsEC`	Specify `true` if the modem supports any error correction protocols (such as MNP 5, V.42, LAPM) and the profile contains configuration strings for error correction. Note that `kdirectConnectOnly` must also be `nil`. Specify `nil` if the modem does not support error correction.
`kSupportsLCS`	Specify `true` if the modem supports LCS (Line Current Sense); otherwise, specify `nil` otherwise. LCS is used for determining when a user has lifted the telephone handset off the hook. Applications can take advantage of this feature by allowing the modem to determine when it should release the line for a voice telephone call.
`kDirectConnectOnly`	Normally this is set to `nil`. Set to `true` if the modem does not support error correction or buffering.
`kConnectSpeeds`	An array indicating the speeds (in bps) at which the modem can connect. This array is not used, except by application that want to determine the modem capabilities.

continued

Table 6-5 Constants for the modem setup profile (continued)

Constant	Description
`kCommandTimeout`	Indicates how long (in milliseconds) the modem tool should wait for a modem response to a command before timing out. A setting of 2000 ms is usually sufficient, though some modems may require 3000 or 4000 ms.
`kMaxCharsPerLine`	Indicates the maximum number of command line characters that the modem can accept, not counting the `AT` prefix and the ending carriage return.
`kInterCmdDelay`	Indicates the minimum amount of delay required between modem commands, in milliseconds. This is the time from the last response received to the next command sent. A setting of 25 ms is usually sufficient, though you can adjust this to up to 40 ms if necessary. This setting should be kept as low as possible.
`kModemIDString`	Normally set this to the string `"unknown"`. This string is used if the modem tool attempts to identify the modem using the `ATI4` command. It should be set to the same string with which the modem responds.

continued

Table 6-5 Constants for the modem setup profile (continued)

Constant	Description
kConfigStrNoEC	The configuration string used for non–error–corrected data connections when kDirectConnectOnly is true, and for FAX connections. This configuration string must enable speed buffering. The default string is as follows: E0 Echo off (always required) &C1 DCD indicates the true state of the remote carrier S12=12 Escape guard time is 240 ms (12*20). Modems usually set S12 to 50. W2 Report connection in "CONNECT *bps*" format. Not all modems accept this command. An alternative is to use Q0 with X1 or X4, and V1. &K3 Enables bidirectional RTS/CTS flow control. The modem uses CTS to control flow from the Newton, and the Newton uses RTS to control flow from the modem. This does not work on all modems. An alternate form is \Q3\X0. It is possible that &R0 and \D1 will be required as well. &Q6 Use normal buffered mode. Again, this does not work on all modems. An alternate form is to use \N0, or on some modems \N7. Without hardware flow control (kdirectConnectOnly is true), software flow control should be used for FAX connections. In this case, instead of &K3, use the following commands: &K4 Enables bidirectional XON/XOFF flow control. The modem and Newton halt data flow when they receive XOFF (DC3) and resume data flow when they receive XON (DC1). This does not work on all modems. An alternate form is \Q1\X0. &R1 Assume RTS is always asserted. This does not work on all modems. \D0 Force CTS on at all times. This does not work on all modems.

continued

Table 6-5 Constants for the modem setup profile (continued)

Constant	Description
`kConfigStrECOnly`	The configuration string used for data connections that require error correction. This configuration string must enable speed buffering and can be used only if hardware flow control can be enabled. The default string is `nil`. Here is an example: `E0` Echo off (always required). `&C1` DCD indicates the true state of the remote carrier. `S12=12` Escape guard time is 240 ms (12*20). Modems usually set `S12` to `50`. `W2` Report connection in "CONNECT *bps*" format. Not all modems accept this command. An alternative is to use `Q0` with `X1` or `X4`, and `V1`. `&K3` Enables bidirectional RTS/CTS flow control. The modem uses CTS to control flow from the Newton, and the Newton uses RTS to control flow from the modem. This does not work on all modems. An alternate form is `\Q3\X0`. It is possible that `&R0` and `\D1` are required as well. `&Q5` Use reliable mode. Again, this does not work on all modems. An alternate form is to use `&M4` or `\N6`. `\N6` Try to establish a reliable LAPM link; if that fails, try to establish an MNP link, and if that fails, disconnect. You could also try `\N4`, especially for cellular connections. `%C1` Enable bilateral MNP 5 or V.42bis data compression. (Note that this can be interpreted differently on different modems.) `\M1` Enable V.42 detection phase.

continued

Table 6-5 Constants for the modem setup profile (continued)

Constant	Description
`kConfigStrECAndFallback`	The configuration string used for data connections that allow error–corrected communication, and if error correction negotiation fails, the modem falls back to a non–error corrected connection. This configuration string must enable speed buffering and can be used only if hardware flow control can be enabled. The default string is `nil`. Here is an example: `E0` Echo off (always required). `&C1` DCD indicates the true state of the remote carrier. `S12=12` Escape guard time is 240 ms (12*20). Modems usually set `S12` to `50`. `W2` Report connection in "CONNECT *bps*" format. Not all modems accept this command. An alternative is to use `Q0` with `X1` or `X4`, and `V1`. `&K3` Enables bidirectional RTS/CTS flow control. The modem uses CTS to control flow from the Newton, and the Newton uses RTS to control flow from the modem. This does not work on all modems. An alternate form is `\Q3\X0`. It is possible that `&R0` and `\D1` are required as well. `&Q5` Use reliable mode and fall back depending on the value in register S36. Again, this does not work on all modems. An alternate form is to use `&Q9`, `&M4`, or `\N7`. `%C1` Enable bilateral MNP 5 or V.42bis data compression. (Note that this can be interpreted differently on different modems.) `\M1` Enable V.42 detection phase.

continued

Table 6-5 Constants for the modem setup profile (continued)

Constant	Description
`kConfigStrDirectConnect`	The configuration string used for data connections for modems that have no speed buffering, and have no error correction or compression built in (`kdirectConnectOnly` is set to `true`). The default string is as follows: `E0` Echo off (always required) `&C1` DCD indicates the true state of the remote carrier `S12=12` Escape guard time is 240 ms (12*20). Modems usually set `S12` to 50. `W2` Report connection in "CONNECT *bps*" format. Not all modems accept this command. An alternative is to use `Q0` with `X1` or `X4`, and `V1`. `&K0` Disable serial port flow control. The Newton must be dynamically configured to match speeds with the modem's negotiated speed. This does not work on all modems. An alternate form is `\Q0\X0`. `&Q0` Use direct connect mode. Again, this does not work on all modems. An alternate form is to use `\N1`. `%C0` Disable data compression. (Note that this can be interpreted differently on different modems.)

Fax Profile Option

The following constants specify the fax setup preferences for configuring the modem controller.

Table 6-6 Constants for the fax profile

Constant	Description
`kTransmitDataMod`	Specifies the set of speeds at which the fax can be sent. If this constant isn't defined, then the fax send speed isn't restricted
`kReceiveDataMod`	Specifies the set of speeds at which the fax can be received. If this constant isn't defined, then the fax receive speed isn't restricted

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

Glossary

Action button	The small envelope button used in applications to send data to other places. 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 is used to save space, since the alias object is small, and it can be used to reference very large objects. Resolving an alias refers to retrieving the object that the alias references. See also **entry alias.**
callback spec	A frame that is passed as an argument to an endpoint method. The callback spec frame contains slots that control how the endpoint method executes, and it contains a completion method that is called when the endpoint operation completes.
class	A symbol that describes the data referenced by an object. Arrays, frames, and binary objects can have user-defined classes.
cursor	An object returned by the `Query` method. The cursor contains methods used to 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 that data. The soup entries which it defines are to be used by an application and stored in its soup. A data definition is registered with the system. The shortened term dataDef is sometimes used.
data form	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.
endpoint	An object created from the proto `protoBasicEndpoint`, or one of its derivative protos. This object encapsulates and maintains the details of the specific connection, and it allows you to control the underlying communication tool.
endpoint option	An endpoint option is specified in a frame that is 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 alias	An object that provides a standard way to save a reference to a soup entry. Entry aliases themselves may be stored in soups.
EOP	End of packet indicator.
In/Out Box	The application that serves as a central repository for all incoming and outgoing data handled by the Routing and Transport interfaces.
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.

item frame	The frame that encapsulates a routed (sent or received) object and that is stored in the In/Out Box soup.
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.
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.
output spec	A special type of **callback spec** that is 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.
routing format	A frame that describes how an object to be sent (routed) is to be formatted. Some 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.
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.**
target	The object being sent. Sometimes the target consists of multiple items encapsulated in a single frame, for example, when multiple items are selected from an overview for sending.

transport	A special type of Newton application that is used to send and/or receive data. Transports communicate with the In/Out Box on one end and to an endpoint object on the other end. They provide user interface features such as routing slips, routing headers, status slips, and preferences slips.
view definition	A view template that defines how data from a particular data definition is to be displayed. 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.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

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
Christopher Bey, Yvonne Tornatta, Dirk van Nouhuys

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 and Gavin Peacock.