Newton Programmer's Reference

For Newton 2.0

Printed in the United States of America.

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

Simultaneously published in the United States and Canada.

Table of Contents

Chapter 5 Pickers, Pop-up Views, and Overviews Reference 5-1

Chapter 8 Recognition System Reference 8-1

Chapter 13 Find Reference 13-1

Chapter 14 System Services Reference 14-1

Chapter 15 Intelligent Assistant Reference 15-1

Chapter 18 Routing Interface Reference 18-1

Chapter 19 Transport Interface Reference 19-1

Chapter 20 Endpoint Interface Reference 20-1

Appendix Error Codes A-1

Figures and Tables

About This Book

This book, *Newton Programmer's Reference*, is the definitive reference for Newton programming. It describes all of the protos, methods, functions, data structures, error codes, and other constructs that are part of the Newton application programming interface (API).

This book is a companion to *Newton Programmer's Guide*, which provides conceptual information and instructions for using the Newton application programming interfaces.

Audience

This reference is for anyone who wants to write NewtonScript programs for the Newton family of products.

Before using this reference, 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*.

To make best use of this reference, you should already have a good understanding of the information presented in the companion volume to this book, *Newton Programmer's Guide*.

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*. This companion volume is the definitive guide to Newton programming.
- *Newton Toolkit User's Guide*. This book comes with the Newton Toolkit development environment. It 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 comes with the Newton Toolkit development environment. It describes the NewtonScript programming language.
- *Newton Book Maker User's Guide*. This book comes with the Newton Toolkit development environment. It describes how to use Newton Book Maker and Newton Toolkit to make Newton digital books and to add online help to Newton applications.
- *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 development environment, from Apple Computer, includes many sample code projects. You can examine these samples, learn from them, and experiment with them. 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. The latest sample code is included each quarter on the Newton Developer CD, which is distributed to all Newton Developer Program members and to subscribers of the Newton monthly mailing. Sample code is updated on the Newton Development side on the World Wide Web (`http://dev.info.apple.com/newton`) shortly after it is released on the Newton Developer CD. For information about how to contact Apple Computer regarding the Newton Developer Program, see the section "Developer Products and Support," on page xxxvii.

The code samples in this book show methods of using various routines and illustrate techniques for accomplishing particular tasks. All code samples have been compiled and, in most cases, tested. However, Apple Computer does not intend that you use these code samples in your application.

To make the code samples in this book more readable, only limited error handling is shown. You need to develop your own techniques for detecting and handling errors.

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

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

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

Apple Developer Catalog
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	ORDER.ADC
Internet	order.adc@applelink.apple.com
World Wide Web	http://www.devcatalog.apple.com

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

For Newton-specific information, see 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.

Getting Started Reference

This chapter describes the view classes, protos, and functions useful for creating any application.

View Classes and Protos

clView

The `clView` view class is the base view class. It implements a generic view that has no special characteristics or specific kind of data associated with it. This view class does not support recognition, gestures, or user input of any kind.

When a `clView` is used as the base view of an application, it typically includes many application-specific slots containing global data and methods for use by its child views (which automatically inherit parental slots if they are not overridden). The minimal slots of interest are listed below.

Slot descriptions

viewBounds	Set to the size and location where you want the view to appear.
viewFlags	The default setting is `vVisible`.
viewFormat	Optional. The default setting is `nil`.

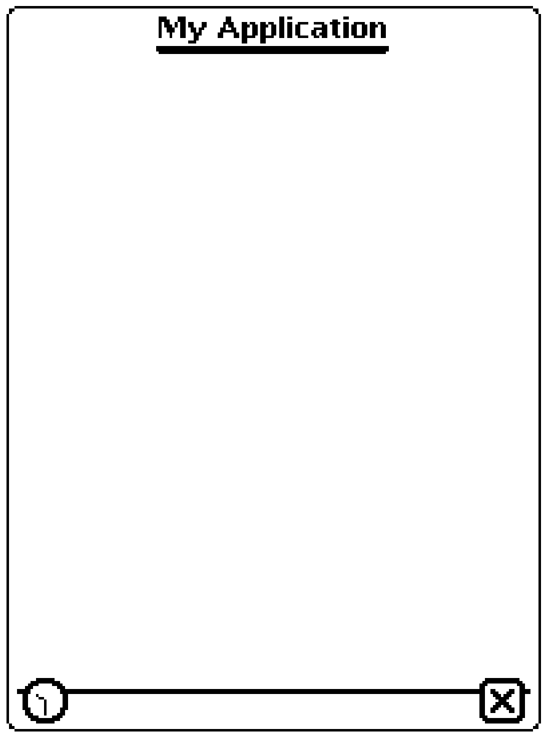
Here is an example of a template defining a view of the `clView` class:

```
sampleApp := {...
   viewClass: clView,
   viewBounds: {left:0, top:0, right:200, bottom:200},
   viewFlags: vApplication+vClickable,
   viewFormat: vfFrameBlack+vfPen1+vfShadow1,
   viewJustify: vjParentCenterH,
   viewEffect: fxUp+fxSteps(8),
   declareSelf: 'base,  // for closebox child

   // methods and other view-specific slots
   ViewSetupFormScript: func()...
...}
```

protoApp

This proto is used to create a simple application base view. It is a view with a title at the top and a status bar at the bottom. The user can tap on the clock icon to see the current time, or on the close box to close the application. Here is an example:



Slot descriptions

`title`	A string that is the title. This title appears in a title bar at the top of the view.
`viewBounds`	Set to the size and location where you want the view to appear. By default it is centered horizontally within its parent view.
`viewFlags`	The default setting is `vVisible + vApplication`. Do not change these flags, but you can add others if you wish.
`viewJustify`	Optional. The default setting is `vjParentCenterH`.

`viewFormat` Optional. The default setting is `vfFillWhite + vfFrameBlack + vfPen(1) + vfInset(1) + vfShadow(1)`.

`declareSelf` Do not change. This slot is set by default to `'base`. This identifies the view to be closed when the user taps the close box.

The `protoApp` has two child views: a title and a status bar.

Here is an example of a template using `protoApp`:

```
myApp := {...
_proto: protoApp,
title: "My Application",
// set bounds relative to screen size
ViewSetupFormScript: func()
   begin
   local b := GetAppParams();
   self.viewBounds.top := b.appAreaTop + 2;
   self.viewBounds.left := b.appAreaLeft;
   self.viewBounds.bottom := b.appAreaHeight - 7;
   self.viewBounds.right := b.appAreaWidth - 21;
   end
...}
```

Functions

Application-Defined Functions

This section describes functions that are called when applications and other parts are installed and removed from the Newton device.

InstallScript

```
InstallScript(partFrame) // for application part
InstallScript(partFrame, removeFrame) // for auto part
```

This function in the application or auto part is executed when the package is activated on the Newton or when the Newton is reset.

partFrame — The part frame. For an application part, this frame contains a slot named `theForm`, which contains a reference to your application's base template. For an auto part, there is no `theForm` slot.

removeFrame — This parameter is passed to this function only if an auto part is being installed, otherwise, only one parameter is passed. The *removeFrame* parameter is the frame that will be passed to the auto part `RemoveScript` function. This frame contains a single slot, `RemoveScript`, which contains a copy of the `RemoveScript` function. Note that you can add additional slots to this frame.

For application parts, the system executes a deep clone of the `InstallScript` function, so you don't normally need to use `EnsureInternal` within it. It's recommended that you keep the `InstallScript` function as small as possible for application parts, because the function is copied into the NewtonScript heap as a result of the deep clone. If you need to execute a lot of code, you might want to make a method in the application base template and send it a message from your `InstallScript`. You can access the base template using the expression *partFrame*`.theForm`. The code in the application method won't be deep cloned since it's not part of the `InstallScript` function.

For auto parts, the `InstallScript` function is not cloned or copied. You must use `EnsureInternal` within this function as appropriate, to prevent the warning to reinsert the card.

DeletionScript

`DeletionScript()`

This function in the part is executed when the package is deleted by the user from the Extras drawer. Typically this function is used to do clean-up operations that you might need to do when the part is deleted.

This function applies to all types of package parts, except for store parts.

After the `DeletionScript` function is executed, the `RemoveScript` function is also executed (for application and auto parts only).

DoNotInstallScript

`DoNotInstallScript()`

This function in the part is executed before the package is first loaded onto a Newton store from some external source. It gives the parts in the package a chance to prevent installation of the package. If any part returns a non-`nil` value from this function, the package is not installed.

You should provide the user with some kind of feedback if package installation is prevented, rather than silently failing.

This function applies to all types of package parts, except for store parts.

RemoveScript

`RemoveScript(`*frame*`)`

This function in the application or auto part is executed when the package is deactivated.

frame For an application part, this parameter is equivalent to the part frame. Note that because the application has been removed, the `theForm` slot contains an invalid reference. For an auto part, this parameter is the same *removeFrame* parameter passed to the `InstallScript` function. Note that the `InstallScript` function can add additional slots to this frame.

Note that the function that is executed is actually a clone of the `RemoveScript` function in your part.

If the application or auto part package is deleted by the user from the Extras drawer, the `DeletionScript` function is executed before the `RemoveScript` function.

Views Reference

This chapter describes the constants, functions, and methods used by the view system interface.

Constants

The following sections contain descriptions of the constants used in the view interface:

- view class constants
- `viewFlags` constants
- `viewJustify` constants
- `viewFormat` constants
- `viewTransferMode` constants
- `viewEffect` constants

View Class Constants

The view class constants are listed and described in Table 2-1.

Table 2-1 View class constants

Constant	Value	Description
`clView`	74	The base view class. This class is used for a generic view that has no special characteristics. A view of this class is generally a container view that encloses other more specialized views. Such a high-level view would include global data and methods shared by its child views. See Chapter 2, "Getting Started," in the *Newton Programmer's Guide* for more information.
`clPictureView`	76	Used for pictures. See Chapter 13, "Drawing and Graphics," in the *Newton Programmer's Guide* for more information.
`clEditView`	77	Used for editing views that can accept both text and graphic user input. This view class typically has child views that are of class `clParagraphView` and `clPolygonView`. See Chapter 8, "Text and Ink Input and Display," in the *Newton Programmer's Guide* for more information.
`clParagraphView`	81	A static or editable text view. When text is recognized, it is displayed in one of these views. Text is grouped into paragraphs so that many words can be shown in a single paragraph view. See Chapter 8, "Text and Ink Input and Display," in the *Newton Programmer's Guide* for more information.

Table 2-1 View class constants (continued)

Constant	Value	Description
`clPolygonView`	82	A graphic view used in an edit view. When a shape is recognized, it is displayed in one of these graphic views. See Chapter 13, “Drawing and Graphics,” in the *Newton Programmer’s Guide* for more information.
`clKeyboardView`	79	Used to define keyboard-like arrays of buttons that can be tapped. No other forms of input recognition are available. See Chapter 8, “Text and Ink Input and Display,” in the *Newton Programmer’s Guide* for more information.
`clMonthView`	80	Used to define a calendar view of a month that lets the user select a date range. See Chapter 6, “Pickers, Pop-up Views, and Overviews,” in the *Newton Programmer’s Guide* for more information.
`clRemoteView`	88	Used for a view that displays another view as its contents. This can be used to show a page preview of a full-page view, for example. This view provides the scaling necessary to display the entire remote view. See Chapter 13, “Drawing and Graphics,” in the *Newton Programmer’s Guide* for more information.
`clPickView`	91	Used to display a list from which you can pick an item. The list can display both text and graphic items. This view class is supported through the `protoPicker` view proto. See Chapter 6, “Pickers, Pop-up Views, and Overviews,” in the *Newton Programmer’s Guide* for more information.

Table 2-1 View class constants (continued)

Constant	Value	Description
`clGaugeView`	92	Used to define a gauge-like view that can display a visual sliding bar indicator. The view can be read-only or changeable. With a changeable view, the user can drag the indicator to a new position. See Chapter 7, "Controls and Other Protos," in the *Newton Programmer's Guide* for more information.
`clOutline`	105	Used for a text outline with expandable headings that have indented subheadings. The user can tap headings to expand and collapse them and to choose items. See Chapter 7, "Controls and Other Protos," in the *Newton Programmer's Guide* for more information.

viewFlags Constants

The `viewFlags` constants are listed and described in Table 2-2. Several additional constants can be specified in the `viewFlags` slot that control what kinds of pen input (taps, strokes, words, letters, numbers, and so on) are recognized and handled by the view. These other constants are described in "Text and Ink Input and Display Reference" (page 7-1).

Table 2-2 `viewFlags` constants

Constant	Value	Description
`vVisible`	1	The view is visible. (Don't set this flag for your application base view, because you don't want it to be shown until the user taps its icon in the Extras Drawer.) If you Show, Hide, Open, Close, or Toggle a view, this flag is changed in the view by the system to reflect the current state of the view.

Table 2-2 `viewFlags` constants (continued)

Constant	Value	Description
`vApplication`	4	Identifies a view that should receive scrolling and other high-level events. For example, when the user taps the scroll arrows, the system searches all views to find the frontmost view that has this bit set, and then sends the scroll event to that view. Generally, this flag is set for the application base view. Views with this flag set can be found with the special view symbols `'viewFrontMost` or `'viewFrontMostApp`.
`vCalculateBounds`	8	The view bounds are not fixed, but are recalculated and will grow if the user enters more information than the view can hold. Used by views of the class `clParagraphView` and `clPolygonView` only, and only when they are enclosed in a view of the class `clEditView`.
`vClipping`	32	The view's contents, including child views, are clipped to its bounds when it is drawn. Note that the base view of all applications is automatically clipped, whether or not this flag is set.
`vFloating`	64	The view is a floating view; that is, it floats above its non-`vFloating` sibling views. A view without this flag will never come in front of a floating sibling view.
`vReadOnly`	2	The view cannot be changed, but it can be scaled or distorted. It is read-only.
`vWriteProtected`	128	The same as `vReadOnly`, except that this flag propagates automatically to all of the view's child views. Additionally, scaling and distortion of the view are not allowed.

Table 2-2 `viewFlags` constants (continued)

Constant	Value	Description
`vNoScripts`	134217728	Prevents the system from sending in the view any of the system messages described "Application–Defined Methods" (page 2-65) (except for the `ViewChangedScript`, and `ViewSetupFormScript` messages, which are still sent). Setting this flag speeds up the processing for a view if it has no application-defined handling methods, because the system won't bother trying to send it messages. This flag is set internally for views of the classes `clParagraphView`, `clPicutureView`, and `clPolygonView` that are created dynamically as the user writes in a `clEditView`.
`vClickable`	512	Allows the view to receive pen input. The system sends the `ViewClickScript` message to the view once for each pen tap (click) that occurs within the view. See to "Text and Ink Input and Display" (page 8-1) in the *Newton Programmer's Guide* for more information.
`vNoFlags`	0	There are no flag attributes for the view.

viewJustify Constants

The constants used for the `viewJustify` slot are listed and described in Table 2-3.

Table 2-3 `viewJustify` constants

Constant	Value	Description
Horizontal alignment of view contents		
`vjLeftH`	0	Left alignment (default).
`vjCenterH`	2	Center alignment (default for `clPictureView` only).

Table 2-3 `viewJustify` constants (continued)

Constant	Value	Description
`vjRightH`	1	Right alignment.
`vjFullH`	3	Stretches the view contents to fill the entire view width.
Vertical alignment of view contents[1]		
`vjTopV`	0	Top alignment (default).
`vjCenterV`	4	Center alignment (default for `clPictureView` only).
`vjBottomV`	8	Bottom alignment.
`vjFullV`	12	For views of the `clPictureView` class only; stretches the picture to fill the entire view height.
Horizontal alignment of the view relative to its parent or sibling view[2]		
`vjParentLeftH`	0	The left and right view bounds are relative to the parent's left side (default).
`vjParentCenterH`	16	The difference between the left and right view bounds is used as the width of the view. If you specify zero for left, the view is centered in the parent view. If you specify any other number for left, the view is offset by that much from a centered position (for example, specifying left = 10 and right = width+10 offsets the view 10 pixels to the right from a centered position).
`vjParentRightH`	32	The left and right view bounds are relative to the parent's right side, and will usually be negative.
`vjParentFullH`	48	The left bounds value is used as an offset from the left edge of the parent and the right bounds value as an offset from the right edge of the parent (for example, specifying left = 10 and right = –10 leaves a 10-pixel margin on each side).
`vjSiblingNoH`	0	(Default) Do not use sibling horizontal alignment.

Table 2-3 `viewJustify` constants (continued)

Constant	Value	Description
`vjSiblingLeftH`	2048	The left and right view bounds are relative to the sibling's left side.
`vjSiblingCenterH`	512	The difference between the left and right view bounds is used as the width of the view. If you specify zero for left, the view is centered in relation to the sibling view. If you specify any other number for left, the view is offset by that much from a centered position (for example, specifying left = 10 and right = width+10 offsets the view 10 pixels to the right from a centered position).
`vjSiblingRightH`	1024	The left and right view bounds are relative to the sibling's right side.
`vjSiblingFullH`	1536	The left bounds value is used as an offset from the left edge of the sibling and the right bounds value as an offset from the right edge of the sibling (for example, specifying left = 10 and right = –10 indents the view 10 pixels on each side relative to its sibling).
Vertical alignment of the view relative to its parent or sibling view[3]		
`vjParentTopV`	0	The top and bottom view bounds are relative to the parent's top side (default).
`vjParentCenterV`	64	The difference between the top and bottom view bounds is used as the height of the view. If you specify zero for top, the view is centered in the parent view. If you specify any other number for top, the view is offset by that much from a centered position (for example, specifying top = –10 and bottom = height–10 offsets the view 10 pixels above a centered position).
`vjParentBottomV`	128	The top and bottom view bounds are relative to the parent's bottom side.

Table 2-3 `viewJustify` constants (continued)

Constant	Value	Description
`vjParentFullV`	192	The top bounds value is used as an offset from the top edge of the parent and the bottom bounds value as an offset from the bottom edge of the parent (for example, specifying top = 10 and bottom = –10 leaves a 10-pixel margin on both the top and the bottom).
`vjSiblingNoV`	0	(Default) Do not use sibling vertical alignment.
`vjSiblingTopV`	16384	The top and bottom view bounds are relative to the sibling's top side.
`vjSiblingCenterV`	4096	The difference between the top and bottom view bounds is used as the height of the view. If you specify zero for top, the view is centered in relation to the sibling view. If you specify any other number for top, the view is offset by that much from a centered position (for example, specifying top = –10 and bottom = height–10 offsets the view 10 pixels above a centered position).
`vjSiblingBottomV`	8192	The top and bottom view bounds are relative to the sibling's bottom side.
`vjSiblingFullV`	12288	The top bounds value is used as an offset from the top edge of the sibling and the bottom bounds value as an offset from the bottom edge of the sibling (for example, specifying top = 10 and bottom = –10 indents the view 10 pixels on both the top and the bottom sides relative to its sibling).
Text limits		
`noLineLimits`	0	(Default) No limits, text wraps to next line.
`oneLineOnly`	8388608	Allows only a single line of text, with no wrapping.
`oneWordOnly`	16777216	Allows only a single word. (If the user writes another word, it replaces the first.)
Indicate that a bounds value is a ratio		
`vjNoRatio`	0	(Default) Do not use proportional alignment.

Table 2-3 `viewJustify` constants (continued)

Constant	Value	Description
`vjLeftRatio`	67108864	The value of the slot `viewBounds.left` is interpreted as a percentage of the width of the parent or sibling view to which this view is horizontally justified.
`vjRightRatio`	134217728	The value of the slot `viewBounds.right` is interpreted as a percentage of the width of the parent or sibling view to which this view is horizontally justified.
`vjTopRatio`	268435456	The value of the slot `viewBounds.top` is interpreted as a percentage of the height of the parent or sibling view to which this view is vertically justified.
`vjBottomRatio`	-536870912	The value of the slot `viewBounds.bottom` is interpreted as a percentage of the height of the parent or sibling view to which this view is vertically justified.
`vjParentAnchored`	256	The view is anchored at its location in its parent view, even if the origin of the parent view is changed. Other sibling views will be offset, but not child views with this flag set.

[1] For views of the `clParagraphView` class, the vertical alignment constants `vjTopV`, `vjCenterV`, and `vjBottomV` apply only to paragraphs that also have the `oneLineOnly` `viewJustify` flag set.
[2] If you are applying horizontal sibling-relative alignment and the view is the first child, it is positioned according to the horizontal parent-relative alignment setting.
[3] If you are applying vertical sibling-relative alignment and the view is the first child, it is positioned according to the vertical parent-relative alignment setting.

viewFormat Constants

The constants used for the `viewFormat` slot are listed and described in Table 2-4.

Table 2-4 `viewFormat` constants

Constant	Value	Description
`vfNone`	0	There are no format attributes set for the view (default).
View fill color		
`vfFillWhite`	1	Fill view with white.
`vfFillLtGray`	2	Fill view with light gray.
`vfFillGray`	3	Fill view with gray.
`vfFillDkGray`	4	Fill view with dark gray.
`vfFillBlack`	5	Fill view with black.
`vfFillCustom`	14	Fill the view with the custom pattern specified in the `viewFillPattern` slot.
View frame color		
`vfFrameWhite`	16	White frame.
`vfFrameLtGray`	32	Light gray frame.
`vfFrameGray`	48	Gray frame.
`vfFrameDkGray`	64	Dark gray frame.
`vfFrameBlack`	80	Black frame.
`vfFrameMatte`	240	Thick gray frame bordered by a black frame, giving a matte effect.

Table 2-4 `viewFormat` constants (continued)

Constant	Value	Description
`vfFrameDragger`	208	Similar effect to `vfFrameMatte`, except that `vfFrameDragger` includes a small control nub in the top portion of the frame at the center. This nub indicates that the user can tap there and drag the view around.
`vfFrameCustom`	224	Use the custom frame pattern specified in the `viewFramePattern` slot.
View frame thickness		
`vfPen(`*pixels*`)`	*pixels* * 256	Sets the frame width; *pixels* specifies the pen thickness in pixels, from 0 through 15. (Note that this is a compile-time only function.)
View frame roundedness		
`vfRound(`*pixels*`)`	*pixels* * 16777216	Sets the corner radius for a rounded frame. *pixels* specifies the corner radius in pixels, from 0 through 15. (Note that this is a compile-time only function.)
View frame inset		
`vfInset(`*pixels*`)`	*pixels* * 65536	Sets the inset style for the frame; that is, the amount of white space (in pixels) between the view bounds and the frame. *pixels* specifies the inset, from 0 through 3. (Note that this is a compile-time only function.)
View shadow style		
`vfShadow(`*pixels*`)`	*pixels* * 262144	Sets the shadow style for the view; *pixels* specifies the thickness of the shadow in pixels that is shown on the bottom and right sides of the view frame. Specify a number from 0 through 3. (Note that this is a compile-time only function.)
View line style (for `clEditView` and `clParagraphView` view classes only)		
`vfLinesWhite`	4096	Draw horizontal lines in white.

Table 2-4 `viewFormat` constants (continued)

Constant	Value	Description
`vfLinesLtGray`	8192	Draw widely dotted horizontal lines.
`vfLinesGray`	12288	Draw dotted horizontal lines.
`vfLinesDkGray`	16384	Draw dashed horizontal lines.
`vfLinesBlack`	20480	Draw solid black horizontal lines.
`vfLinesCustom`	57344	Use the custom line pattern specified in the `viewLinePattern` slot.

viewTransferMode Constants

The constants that you can specify for the `viewTransferMode` slot are listed and described in Table 2-5.

Table 2-5 `viewTransferMode` constants

Constant	Value	Description
`modeCopy`	0	Replaces the pixels in the destination with the pixels in the source, "painting" over the screen without regard for what's already there.
`modeOr`	1	Replaces screen pixels under the black part of the source image with black pixels. Screen pixels under the white part of the source image are unchanged.
`modeXor`	2	Inverts screen pixels under the black part of the source image. Screen pixels under the white part of the source image are unchanged.
`modeBic`	3	Erases screen pixels under the black part of the source image, making them all white. Screen pixels under the white part of the source image are unchanged.

Table 2-5 `viewTransferMode` constants (continued)

Constant	Value	Description
`modeNotCopy`	4	Replaces screen pixels under the black part of the source image with white pixels. Screen pixels under the white part of the source image are made black.
`modeNotOr`	5	Screen pixels under the black part of the source image are unchanged. Screen pixels under the white part of the source image are made black.
`modeNotXor`	6	Screen pixels under the black part of the source image are unchanged. Screen pixels under the white part of the source image are inverted.
`modeNotBic`	7	Screen pixels under the black part of the source image are unchanged. Screen pixels under the white part of the source image are made white.
`modeMask`	8	This is a special transfer mode used for drawing views of the `clPictureView` class only. It causes the picture mask image to be erased first and then the picture bit image is drawn over it using the `modeOr` transfer mode.

viewEffect Constants

Table 2-6 lists all of the constants that you can use in the `viewEffect` slot to create custom animation effects.

Table 2-6 `viewEffect` constants

Constant	Integer Value	Description
`fxSteps(`*x*`)`	(*x*-1)* 2097152	Sets the number of steps (*x*) that the animation should take to complete. Specify an integer from 1 to 15.

Table 2-6 `viewEffect` constants (continued)

Constant	Integer Value	Description
`fxStepTime(`*x*`)`	*x**33554432	Sets the amount of time (*x*) to take for each animation step, in ticks. There are 60 ticks per second, or 16.6 milliseconds per tick. Specify an integer from 0 to 15.
`fxColumns(`*x*`)`	*x*-1	Sets the number (*x*) of columns in which to divide the view for animation purposes.
`fxRows(`*x*`)`	(*x*-1)*32	Sets the number (*x*) of rows in which to divide the view for animation purposes.
`fxMoveH`	65536	Indicates that you want the animation to include horizontal movement. (Note that you can also specify `fxMoveV`.)
`fxHStartPhase`	1024	If specified, indicates that you want the first column to begin moving towards the left. If not specified, the first column begins moving towards the right. This flag can be used only if `fxMoveH` is specified.
`fxColAltHPhase`	4096	If specified, the direction of horizontal movement alternates for each column in the view. If not specified, all columns move in the same direction (left or right) as the first column. This flag can be used only if `fxMoveH` is specified.
`fxRowAltHPhase`	16384	If specified, the direction of horizontal movement alternates for each row in the view. If not specified, all rows move in the same direction (left or right) as the first row. This flag can be used only if `fxMoveH` is specified.
`fxMoveV`	131072	Indicates that you want the animation to include vertical movement. (Note that you can also specify `fxMoveH`.)
`fxVStartPhase`	2048	If specified, indicates that you want the first row to begin moving upwards. If not specified, the first row begins moving downwards. This flag can be used only if `fxMoveV` is specified.

Table 2-6 `viewEffect` constants (continued)

Constant	Integer Value	Description
`fxColAltVPhase`	8192	If specified, the direction of vertical movement alternates for each column in the view. If not specified, all columns move in the same direction (up or down) as the first column. This flag can be used only if `fxMoveV` is specified.
`fxRowAltVPhase`	32768	If specified, the direction of vertical movement alternates for each row in the view. If not specified, all rows move in the same direction (up or down) as the first row. This flag can be used only if `fxMoveV` is specified.
`fxLeft`	66560	Indicates that motion should be towards the left. (This flag is the same as specifying `fxHStartPhase+fxMoveH`.)
`fxRight`	65536	Indicates that motion should be towards the right. (This flag is the same as specifying `fxMoveH` and not specifying `fxHStartPhase`.)
`fxUp`	133120	Indicates that motion should be towards the top. (This flag is the same as specifying `fxVStartPhase+fxMoveV`.)
`fxDown`	131072	Indicates that motion should be towards the bottom. (This flag is the same as specifying `fxMoveV` and not specifying `fxVStartPhase`.)
`fxRevealLine`	262144	If specified, causes a line to be drawn at the edge(s) from which the animation is being revealed. For some types of animation, this setting improves the effect.
`fxWipe`	524288	If specified, causes the view to be revealed in place rather than actually moved into place. In other words, the view is revealed just like a window is revealed by rolling a shade away. Without this flag, the view is actually moved into place.

Table 2-6 `viewEffect` constants (continued)

Constant	Integer Value	Description
`fxFromEdge`	1048576	If specified, causes the animation to begin at the edge of the screen, ending up at the ultimate view location. Without this flag, the entire animation occurs within the bounds of the view being animated.
`fxCheckerboardEffect`	155879	Reveals a view using a checkerboard effect, where adjoining squares move in opposite (up and down) directions.
`fxBarnDoorOpenEffect`	627713	Reveals a view from center towards left and right edges, like a barn door opening where the view is the inside of the barn.
`fxBarnDoorCloseEffect`	626689	Reveals a view from left and right edges towards the center, like a barn door closing where the view is painted on the doors.
`fxVenetianBlindsEffect`	131296	Reveals a view so that it appears behind venetian blinds that open.
`fxIrisOpenEffect`	1023009	Changes the size of an invisible “aperture” covering the view, revealing an ever-increasing portion of the full-size view as the aperture opens.
`fxIrisCloseEffect`	986145	Like `fxIrisOpenEffect`, but decreases the size of an invisible “aperture” covering the view, as the aperture closes.

Table 2-6 `viewEffect` constants (continued)

Constant	Integer Value	Description
`fxPopDownEffect`		
	393216	Reveals a view as it slides down from its top boundary.
`fxDrawerEffect`	133120	Reveals a view as it slides up from its bottom boundary.
`fxZoomOpenEffect`		
	236577	Expands the image of the view from a point in the center until it fills the screen; that is, the entire view appears to grow from a point in the center of the screen.
`fxZoomCloseEffect`		
	199713	Opposite of `fxZoomOpenEffect`. This value shrinks the image of the view from a point in the center until it disappears or closes on the screen.
`fxZoomVerticalEffect`		
	165920	The view expands out from a horizontal line in the center of its bounds. The top half moves upward and lower half moves downward.

Functions and Methods

The following sections describe view functions and methods.

Getting References to Views

The following sections describe the functions and methods used to get references to views.

ChildViewFrames

view:`ChildViewFrames()`

Returns an array of views that correspond to the child views of the view to which this message is sent. The views are returned in the same order they appear in the view hierarchy, from back to front. The most recently opened views (which appear on top of the hierarchy) will be later in the list. Views with the `vFloating` flag will be located at the end of the array.

IMPORTANT
Use this method to get to the child views of a view. If you just reference the `viewChildren` or `stepChildren` slots in the view, you get references to the child templates, not the views. Of course, you can also directly reference any declared child view. ▲

Parent

view:`Parent()`

Returns the parent view of the view to which this message is sent. This is the recommended method of getting a reference to a view's parent view, rather than directly referencing the `_parent` slot.

GetRoot

`GetRoot()`

Returns the system root view.

All applications are normally declared in the root view under their application symbol. This means there is a slot in the root view whose name is the application symbol and whose value is that view. You can use this code to test if an application is open:

`GetRoot().`*applicationSymbol*`.viewCObject;`

If the application is open, this function returns a non-`nil` value; otherwise, `nil` is returned. This reference is always present as long as a view is open, and `nil` when a view is closed.

GetView

`GetView(`*symbol*`)`

Returns the first view found that corresponds to the specified symbol. If no view is found, `nil` is returned.

symbol A symbol identifiying a view template you want to get. Besides a view template name, you can pass in the following special symbols (which are evaluated at run time):

- `'viewFrontMost`, to return the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot.
- `'viewFrontMostApp`, to return the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot, but not including floating views (those with `vFloating` set in their `viewFlags` slot).
- `'viewFrontKey`, to return the view on the screen that accepts keys (there can be only one view that is the key receiver) See "Text and Ink Input and Display" (page 8-1) in the *Newton Programmer's Guide* for more information on key receivers.

Displaying, Hiding, and Redrawing Views

The methods and functions described in the following subsections describe how to display, hide, and redraw views.

Open

view`:Open()`

Creates the graphic representation of the view. This method then plays the "show" sound (stored in the `showsound` slot), brings the view to the front, and shows it and all of its child views.

The view receives the following system messages: `ViewSetupFormScript`, `ViewSetupChildrenScript`, `ViewSetupDoneScript`, `ViewShowScript`, and `ViewDrawScript`. Note that these same system messages (except for `ViewShowScript`) are sent to all visible child views of the view as they are created and shown as well. For information about these system messages, refer to “Application–Defined Methods” (page 2-65).

This method always returns `true`.

Note that this message must be sent to a view, not to a template. To ensure that a view exists for the template, you must have declared it. For details on declaring a view, see “View Instantiation” (page 3-26) in *Newton Programmer’s Guide*.

You can use this code to test if a view is open:

```
view.viewCObject;
```

If the view is open, this code returns a non-`nil` value; otherwise, `nil` is returned. This reference is always present as long as a view is open, and is always `nil` when a view is closed.

Close

view:`Close()`

Closes the specified view. This means that if the view is currently visible, this method plays the “hide” sound (stored in the `hidesound` slot), calls `ViewHideScript`, hides the view and all of its child views, calls `ViewQuitScript`, and then deletes the view from memory. This method always returns non-`nil`.

Note that if the view is hidden (it was opened and then sent the `Hide` message), and you send it the `Close` message, it will be closed. This is because the view is still considered open even when it is hidden. You won’t see anything change on the screen since the view is already not visible, but the view will be deleted from memory. Also, in this case, the “hide” sound is not played and the `ViewHideScript` message is not sent.

If the view has already been closed, nothing happens.

If the view is a declared view, the view memory object is not deleted as a result of the `Close` message, as long as the view it is declared in is still open. Only the graphic representation of the view is deleted. If you want to reopen the view, send it an `Open` or a `Toggle` message.

Note
If you need to close a view from a method within the view itself, you may want to send the `Close` message using the function `AddDeferredCall` so that the `Close` message is delayed until after the currently executing method finishes. For example, you could use code like this:

```
begin
local me := self;
AddDeferredCall (func() me:close(), '[]);
end
```

◆

Toggle

view`:Toggle()`

If the view is currently closed, this method performs the same operations as if the view had been sent the `Open` message.

If the view is currently open, this method performs the same operations as if the view had been sent the `Close` message.

Note that if the view is hidden (it was opened and then sent the `Hide` message), and you send it the `Toggle` message, it will be closed. This is because the view is still considered open even when it is hidden. You won't see anything change on the screen since the view is already invisible, but the view will be deleted from memory. Also, in this case, the "hide" sound is not played.

`Toggle` returns non-`nil` if the view is to be opened, or `nil` if the view is to be closed, as a result of calling this method.

Note that this message must be sent to a view, not to a template. To ensure that a view exists for the template, you must have declared it. For details on declaring a view, see "View Instantiation" (page 3-26) in *Newton Programmer's Guide*.

Note that `Toggle` actually creates and destroys view objects (like `Open` and `Close`), while `Show` and `Hide` simply make existing views visible or invisible.

Show

view:`Show()`

If the view is currently hidden, this method plays the "show" sound (stored in the `showsound` slot), brings the view to the front, shows it and all of its visible child views, and calls the `ViewShowScript`. Note that you must specify a view. The return value is unspecified.

You can use this method only if the view has previously been opened (you have sent it the `Open` or `Toggle` message) and then hidden (you have sent it the `Hide` message).

Even though all children of the view being shown are also shown, the child views are not sent the `ViewShowScript` message. This message is sent only to the view on which you use the `Show` method directly.

Hide

view:`Hide()`

If the view is currently shown, this method plays the "hide" sound (stored in the `hidesound` slot), calls the `ViewHideScript`, and hides the view and all of its child views. The return values is unspecified.

Even though all children of the view being hidden are also hidden, the child views are not sent the `ViewHideScript` message. This message is sent only to the view on which you use the `Hide` method directly.

To show the view again, send it the `Show` message.

Note that when a view is hidden, the view in memory is not destroyed. All that actually happens is the bits are removed from the screen. The view is still considered open. This allows fast performance when the view is subsequently shown again.

Dirty

view:`Dirty()`

Marks the view as needing redrawing. The view (and its visible child views) will be redrawn the next time the system idle task is executed. This method always returns non-`nil`.

The system tries to handle redrawing only the parts of the view hierarchy that have been dirtied, but it has a limited cache of update nodes (places in the view hierarchy where it starts drawing from). If you dirty several views, the update nodes may merge by remembering a common ancestor of two dirty views and starting the redrawing from there when the time comes to update. To flush out the updates, call `RefreshViews`, which sometimes may be more efficient since the update is more precise.

When a view is redrawn as a result of the `Dirty` method, the system does not necessarily reread all of the slots in the view. For example, slots describing the view contents are not read—the contents are assumed to have not changed. If you were to directly change the `text` slot of a `clParagraphView` and then send it the `Dirty` message, you would not see the text in the view change.

Usually, you want a view to redraw with its new contents, if the contents change. To do this, use the global function `SetValue` (page 2-25) to change the contents of slots in the view. The `SetValue` function causes the system to reread the changed slots in the view before it is redrawn, and it automatically dirties the view so you don't have to send it the `Dirty` message.

If you change the bounds of a view directly, `Dirty` does not cause the view to be redrawn with new bounds. To do that, send the view the `SyncView` (page 2-26) message.

OffsetView

view:`OffsetView(` *dx*, *dy* `)`

Offsets a view by *dx* horizontally and *dy* vertically. The return values is unspecified.

dx The x coordinate of amount you want to offset the view.

dy The y coordinate of amount you want to offset the view.

`OffsetView` does the redraw faster and more easily than `SetOrigin.` `OffsetView` changes where a view is within its parent, `SetOrigin` changes the locatin of the children/contents of a view.

RefreshViews

`RefreshViews()`

Redraws all views immediately, if they need to be updated. This function always returns non-`nil`.

SetValue

`SetValue(`*view*`,` *slotSymbol*`,` *value*`)`

Sets the value of a slot in a view. The view is flagged as dirty, so it will be redrawn using the new information.

view The view in which you want to change a slot value.

slotSymbol A symbol naming the slot whose value you want to change. Note that you must specify a symbol (quoted identifier), for example, `'mySlot`.

value The new value of the slot.

This function always returns `nil`.

You can pass in the following special symbols (which are evaluated at run time) for the *view* parameter:

- `'viewFrontMost`, to indicate the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot.
- `'viewFrontMostApp`, to indicate the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot, but not including floating views (those with `vFloating` set in their `viewFlags` slot).
- `'viewFrontKey`, to indicate the view on the screen that accepts keys (there can be only one view that is the key receiver).

As expected, the view is redrawn immediately with its new settings if you set the value of one of the following slots: `viewBounds`, `viewFormat`, `viewJustify`, `viewFont`, `viewFlags`. Additionally, for these slots, the effect is as if you had sent the `SyncView` message to the view, including calling the `ViewSetupFormScript` method (see the `SyncView` method, next).

If the view exists, any dependent views (see the `TieViews` function on (page 2-55)) are notified, and the `ViewChangedScript` message is sent to the view.

If you specify a slot that does not exist in the view, the slot is created in the view.

Note

`SetValue` now changes the recognition behavior of a view at run time by setting new recognition flags in the `viewFlags` slot. The new recognition behavior takes effect immediately following the `SetValue` call. See the 1.0 *Newton Programmer's Guide* for details on this call's previous behavior. ◆

SyncView

view`:SyncView()`

Redraws a view after you change its `viewBounds` slot. Before the view is redrawn with new bounds, the `ViewSetupFormScript` message is sent to the view. `SyncView` always returns `true`.

MoveBehind

viewToMove`:MoveBehind(`*view*`)`

Moves a view behind another view, redrawing the screen as appropriate.

view	The view identified by *viewToMove* is moved behind this view. If the view parameter is `nil`, *viewToMove* is brought to the front.

If the view is a floating view (has the `vFloating` `viewFlags` bit set), it can be moved behind only another floating sibling view, because floating views cannot appear behind nonfloating views.

The return value of this method is undefined.

Dynamically Adding Views

The following functions are useful for creating and removing views at run time.

AddStepView

`AddStepView(`*parentView*`,` *childTemplate*`)`

Dynamically instantiates a new view based on the specified child template and adds it to the parent's `stepChildren` array. You must send the `Dirty` message to the new view or to its parent view to cause the new view to be drawn. See "Using the AddStepView Function" (page 3-35) in *Newton Programmer's Guide* for information on using this function.

parentView	The parent view to which you want to add the new view.
childTemplate	A template describing the new view you want to add.

This function returns the view if it was successfully created; otherwise, `nil` is returned.

You can pass in the following special symbols (which are evaluated at run time) for the *parentView* parameter:

- `'viewFrontMost`, to indicate the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot.
- `'viewFrontMostApp`, to indicate the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot, but not including floating views (those with `vFloating` set in their `viewFlags` slot).
- `'viewFrontKey`, to indicate the view on the screen that accepts keys (there can be only one view that is the key receiver).

Because this function adds an item to the parent's `stepChildren` array, you must ensure that the array is in RAM, or `AddStepView` will fail. You can use this code:

```
if not HasSlot(parentView, 'stepChildren) then
   parentView.stepChildren := Clone(parentView.stepChildren);
```

The `if` statement checks if the `stepChildren` slot already exists in the parent view (in RAM). If it does not, it is copied out of the template in your package into RAM.

Note that you can add an invisible view; that is, one with its `vVisible` flag not set. You might want to do this if you want the view to show itself with an effect. First add it invisibly, then send it the `Show` message. (If you just add it as a visible view, any view effect you specify is not done when it is first displayed.)

RemoveStepView

`RemoveStepView(`*parentView*`, `*childView*`)`

Removes a child view from its parent view. The child view is closed, if visible.

parentView	The parent view from which you want to remove the child view.
childView	The child view you want to remove.

This function always returns `nil`.

You can pass in the following special symbols (which are evaluated at run time) for either the *parentView* or *childView* parameters:

- `'viewFrontMost`, to indicate the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot.
- `'viewFrontMostApp`, to indicate the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot, but not including floating views (those with `vFloating` set in their `viewFlags` slot).
- `'viewFrontKey`, to indicate the view on the screen that accepts keys (there can be only one view that is the key receiver).

If the specified child view is a root-level view (a child of the root view), this function plays the "hide" sound (stored in the `hidesound` slot in the view), sends the view a `ViewHideScript` message, sends the view a `ViewQuitScript` message, and hides the view (and all of its child views).

If the specified child view is not a child of the root view, the same operations occur, except that the hide sound is not played and the `ViewHideScript` message is not sent.

Note
This function removes the view template from the `stepChildren` array of the parent view. You do not need to remove the template yourself. For a description of how this function worked in the previous release, see "Views" in the *1.0 Newton Programmer's Guide*. ◆

AddView

`AddView(`*parentView*`,` *childTemplate*`)`

Dynamically instantiates a new view based on the specified child template and adds it to the parent's `viewChildren` array. You must send the `Dirty` message to the new view or to its parent view to cause the new view to be drawn.

parentView	The parent view to which you want to add the new view.
childTemplate	A template describing the new view you want to add.

This function returns the view if it was successfully created; otherwise, it returns `nil`.

You can pass in the following special symbols (which are evaluated at run time) for the *parentView* parameter:

- `'viewFrontMost`, to indicate the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot.
- `'viewFrontMostApp`, to indicate the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot, but not including floating views (those with `vFloating` set in their `viewFlags` slot).

- `'viewFrontKey`, to indicate the view on the screen that accepts keys (there can be only one view that is the key receiver).

Because this function adds an item to the parent's `viewChildren` array, you must ensure that the array is in RAM, or `AddStepView` will fail. You can use this code:

```
if not HasSlot(parentView, 'viewChildren) then
   parentView.viewChildren := Clone(parentView.viewChildren);
```

The `if` statement checks if the `viewChildren` slot already exists in the parent view (in RAM). If it does not, it is copied out of the template in your package into RAM.

Note that you can add an invisible view; that is, one with its `vVisible` flag not set. You might do this if you want the view to show itself with an effect. First add it invisibly, then send it the `Show` message. (If you just add it as a visible view, any view effect you specify is not done when it is first displayed.)

RemoveView

`RemoveView(`*parentView*`, `*childView*`)`

Removes a child view from its parent view. The child view is closed, if visible.

parentView — The parent view from which you want to remove the child view.

childTemplate — The child view you want to remove.

This function always returns `nil`.

You can pass in the following special symbols (which are evaluated at run time) for the either the *parentView* or *childView* parameters:

- `'viewFrontMost`, to indicate the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot.
- `'viewFrontMostApp`, to indicate the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot, but not including floating views (those with `vFloating` set in their `viewFlags` slot).

- 'viewFrontKey, to indicate the view on the screen that accepts keys (there can be only one view that is the key receiver).

If the specified child view is a root-level view (a child of the root view), this function plays the "hide" sound (stored in the hidesound slot in the view), sends the view a ViewHideScript message, sends the view a ViewQuitScript message, and hides the view (and all of its child views).

If the specified child view is not a child of the root view, the same operations occur, except that the hide sound is not played and the ViewHideScript message is not sent.

BuildContext

BuildContext(*template*)

Dynamically instantiates a new view based on the specified template and adds it to the root view.

template A template describing the new view you want to add.

This function returns the view that it creates.

To display the newly created view, send it the Open message. The viewFlags slot must not have the vVisible flag set. It's best if you don't set the vVisible flag in the *template*; that way you can display the view with a simple Open message, and this also allows any view effect you specify to be done when the view is first shown.

The parent of the new view is set to the root view. The template is not added to the viewChildren or stepChildren array of any view. The _proto slot of the new view is set to the template that it was created from.

Making Modal Views

The following methods are used to make modal views.

AsyncConfirm

`AsyncConfirm(`*confirmMessage*`, `*buttonList*`, `*fn*`)`

This method creates and displays a slip that the user must dismiss before continuing. The slip is created at a deferred time, so the call to `AsyncConfirm` returns immediately, allowing the currently executing NewtonScript code to finish. `AsyncConfirm`'s return value is unspecified.

confirmMessage	A string to be displayed to the user.
buttonList	A symbol (`'okCancel`, `'yesNo`), an array of strings, for example `["Three", "Two", "One"]`, or an array of frames; each frame has two slots, `'value` and `'text`. The slot `'text` holds the value for the button, a string. The slot `'value` holds the result that tapping the button generates. If a symbol was passed, the result is non-`nil` for the "OK" and "Yes" buttons, and `nil` for the "Cancel" and "No" buttons. If an array of strings was passed, the result is the index into the array of the item that was chosen. If an array of frames was passed, the result is the contents of the value slot for the item that was chosen.
fn	A closure to be called when the slip is dismissed. It is passed as one argument, the value of the button tapped.

ModalConfirm

`ModalConfirm(`*confirmMessage*`, `*buttonList*`)`

This method creates and displays a slip that returns the result of the user's choice. Because this method causes a new task to be spawned, it is less efficient and takes more system overhead, so you should use `AsyncConfirm` in most cases.

For example:

```
if ModalConfirm("Do you want to erase?", 'okCancel) then
     ...
```

confirmMessage A string to be displayed to the user.

buttonList See `AsyncConfirm` for a list of symbols and arrays that you can pass in for the *buttonList*.

FilterDialog

view:`FilterDialog()`

This method opens a view and returns `true` immediately after opening. `FilterDialog` is the same as `Open` except that the view is modal. This means that all taps outside the modal view are ignored while the modal view is open. The modal state is exited when the modal view is closed.

`FilterDialog` is preferred over `ModalDialog` as it does not spawn a new task when it is used.

Like `Open`, the `FilterDialog` method creates the graphic representation of the view. It then plays the "show" sound (stored in the `showsound` slot), brings the view to the front, and shows it (and all of its child views). The view receives the following system messages: `ViewSetupFormScript`, `ViewSetupChildrenScript`, `ViewSetupDoneScript`, `ViewDrawScript`, and `ViewShowScript`. For information about these system messages, refer to "Application–Defined Methods" (page 2-65).

Note that the `FilterDialog` message must be sent to a view, not to a template. To ensure that a view exists for the template, you must have declared it. For details on declaring a view, see "View Instantiation" (page 3-26) in *Newton Programmer's Guide*..

ModalDialog

view:`ModalDialog()`

This method is the same as `FilterDialog`, except that it spawns a separate OS task and doesn't return until after the dialog is closed.

This method always returns `true`.

Note
`ModalDialog` will not work correctly if it is sent to a non-root child. ◆

Setting the Bounds of Views

The following functions and view methods calculate and return a `viewBounds` frame.

RelBounds

`RelBounds(`*left*`, `*top*`, `*width*`, `*height*`)`

Returns a `bounds` frame, if you know the top-left coordinate and the width and height of the view. This function calculates the right and bottom values and returns a bounds frame. The value returned can be used for the value of the `viewBounds` slot in a template.

left	The left coordinate of the view.
top	The top coordinate of the view.
width	The width of the view.
height	The height of the view.

SetBounds

`SetBounds(`*left*`, `*top*`, `*right*`, `*bottom*`)`

Returns a `bounds` frame when supplied with the four bounds values. The value returned can be used for the value of the `viewBounds` slot in a template.

left	The left coordinate of the view.
top	The top coordinate of the view.
right	The right coordinate of the view.
bottom	The bottom coordinate of the view.

GlobalBox

view:`GlobalBox()`

Returns the rectangle, in global coordinates, of the specified view. The rectangle is returned as a `bounds` frame. If a valid view is not found, this method throws an exception.

Note

If called from the `ViewSetupFormScript` method, `GlobalBox` gets the `viewBounds` and `viewJustify` slots from the view, calculates the effects of the sibling and parent alignment on the view bounds, and then returns the resulting bounds frame in global coordinates. ◆

GlobalOuterBox

view:`GlobalOuterBox()`

Returns the rectangle, in global coordinates, of the specified view, including any frame that is drawn around the view. The rectangle is returned as a `bounds` frame. If a valid view is not found, this method returns `nil`.

This method is just like `GlobalBox`, except that `GlobalOuterBox` includes the frame around the view.

Note

If called from the `ViewSetupFormScript` method, `GlobalOuterBox` gets the `viewBounds` and `viewJustify` slots from the view, calculates the effects of the sibling and parent alignment on the view bounds, and then returns the resulting bounds frame in global coordinates. ◆

LocalBox

view:`LocalBox()`

Returns a `viewBounds` frame containing the view bounds relative to the view itself. That is, the top-left coordinates are both zero, the right coordinate is the width of the view, and the bottom coordinate is the height of the view. If a valid view is not found, this method throws an exception.

Note

If called from the `ViewSetupFormScript` method, `LocalBox` gets the `viewBounds` and `viewJustify` slots from the view, calculates the effects of the sibling and parent alignment on the view bounds, and then returns the resulting bounds frame in local coordinates. ◆

DirtyBox

view:`DirtyBox(`*boundsFrame*`)`

Marks a portion of a view (or views) as needing redrawing. The view (and its visible child views) is redrawn the next time the system idle task is executed.

boundsFrame A bounds frame describing the area of the screen to be dirtied, in global coordinates.

The return value of this method is undefined.

This method may save screen update time if only a portion of a view needs redrawing, rather than the whole view.

You can use the `DirtyBox` method anywhere you would use the `Dirty` method.

GetDrawBox

view:`GetDrawBox()`

Returns the bounds of the area on the screen that needs redrawing (the area marked as dirty). The dirty area is always non-`nil`. This method returns a bounds frame containing global coordinates.

Note
`GetDrawBox` will provide meaningful results only when called from `ViewDrawScript`. ◆

ButtonBounds

`ButtonBounds(`*width*`)`

Returns a `viewBounds` frame when supplied with the width of a button to be placed in the status bar. You can use this return value for the value of the button `viewBounds` slot.

width The width of the button to place in the status bar.

For the first button you place in the status bar, specify the width as a negative number. For example, if you want the button to be 30 pixels wide, specify –30. This signals that this is the first button, and the bounds are calculated to place it at a standard offset (36 pixels) from the left side of the status bar.

For subsequent buttons that you place in the same status bar, specify the width as a positive number. For subsequent buttons, you must also use the `viewJustify` flag `vjSiblingRightH`.

Note
This function is available in the Newton Toolkit development environment at compile time only. It is not available at run time. ◆

StdButtonWidth

`StdButtonWidth(`*str*`)`

Returns the button size necessary to fit a string of specified text.

str A string that contains the button name.

This function internally calls `StrFontWidth`.

PictBounds

`PictBounds(`*name*`,` *left*`,` *top*`)`

Returns a `viewBounds` frame for views containing pictures. This function opens the picture resource, finds the width and height of the picture, and returns the proper bounds frame. The value returned is used for the value of the `viewBounds` slot in a template.

name	A string that is the name of a PICT resource.
left	The left coordinate of the view.
top	The top coordinate of the view.

Note
This function is available in the Newton Toolkit development environment at compile time only. It is not available at run time. ◆

Animating Views

There are four view methods that perform special animation effects on views.

Effect

view`:Effect(`*effect*`,` *offScreen*`,` *sound*`,` *methodName*`,` *methodParameters*`)`

Posts a message to the specified view to redraw it with an animation. However, the system does not actually do the animation until after it calls the method that you specify, in which you can do any operations required before the animation is done. For example, you might want to animate a view as you change its contents.

effect	Specifies an animation effect. You can specify any of the effect constants that are used in the `viewEffect` slot (see “Opening and Closing Animation Effects” (page 3-23) in *Newton Progammer’s Guide*).
offScreen	Specifies whether or not the view should appear to animate off or onto the screen. Specify non-`nil` to make

the animation appear as if the view is moving off the screen (for example, closing). Specify `nil` to make the animation appear as if the view is moving onto the screen (for example, opening).

sound — A sound frame containing a sound that you want played concurrently with the animation. (If you don't want a sound, specify `nil`.)

methodName — This method changes the state of your view (the two states that the effect transitions between). You must specify a symbol (for example, `'myScript`). Do not change the state of your view before calling `Effect`. This method must be accessible from the view to which the `Effect` message is sent; that is, this method must reside in that view or be accessible from that view through inheritance.

methodParameters — An array of parameters that are passed to your method.

The `Effect` method always returns `nil`.

Here is an example using this method:

```
aView := {...
doEffect: func()
   begin
   view1:Effect(fxZoomVerticalEffect, nil, ROM_plunk,
               'effectScript,[]);
   end,
...}

view1 := {...
text: "",
effectScript: func()
   begin
   SetValue(view1, 'text, "This is a paragraph view...");
```

```
   end,
...}
```

SlideEffect

view:`SlideEffect(`*contentOffset*`,` *viewOffset*`,` *sound*`,` *methodName*`,` *methodParameters*`)`

Posts a message to the specified view to perform a vertical sliding animation on it. However, the system does not actually do the animation until after it calls the method that you specify, in which you must do any operations that change the state of your view.

contentOffset The number of pixels to animate the view contents scrolling in a vertical direction. A positive number makes the view contents appear to move downwards. A negative number makes the view contents appear to move upwards. Note that only the bits on the screen are moved; the location of the actual view data is not affected.

viewOffset The number of pixels to animate the whole view moving up or down on the screen. Specify a positive number to make the view appear to move up on the screen. To make the view appear to move down, specify a negative number.

If you don't want to make the view appear to move, but just want to scroll its contents, specify zero.

sound A sound frame containing a sound that you want played concurrently with the animation. (If you don't want a sound, specify `nil`.)

methodName The method that you want called before the animation occurs. You must specify a symbol (for example, `'myScript`). This method must be accessible from the view to which the `SlideEffect` message is sent; that is, this method must reside in that view or be accessible from that view through inheritance.

methodParameters An array of parameters that are passed to your method.

The `SlideEffect` method always returns `nil`.

Note that this method does not actually change the bounds of the view or the position of its contents. The bits are moved on the screen, but that is all that occurs.

If you want to change the bounds or the position of the contents, you must do so in the method that you supply, appropriately to correspond to the visual effect that you specified in this call.

To animate a view scrolling in place, without changing its size, specify a positive or negative *contentOffset* and zero for *viewOffset* (for example, -50, 0). To slide a view up from the bottom, showing more of it, but keeping the data that was near the top still near the top, specify a negative *contentOffset* and a *viewOffset* that is the same as *contentOffset*, but positive (for example, -50, 50). To shrink the view back down, specify a positive *contentOffset* and a negative *viewOffset* (for example, 50, -50).

Here is an example of this method:

```
aView := {...
slideUp: func()
   begin
   local amount := 100;
   view1:SlideEffect(-amount, amount, ROM_flip,
                     'myEffect, ['up, amount]);
   end,
slideDown: func()
   begin
   local amount := 100;
   view1:SlideEffect(amount, -amount, ROM_flip,
                     'myEffect, ['down, -amount]);
   end,
...}
```

```
view1 := {...
myEffect: func(direction, amount)
   begin
   local bounds := self.viewbounds; //copy viewbounds
   If direction = 'up then
      begin  // only top needs changing
      bounds.top := bounds.top-amount;
      SetValue(view1, 'viewbounds, bounds);
      end
   Else // direction is down
      begin  // only top needs changing
      bounds.top := bounds.top-amount;
      SetValue(view1, 'viewbounds, bounds);
      end
   end,
...}
```

RevealEffect

view:`RevealEffect(`*distance*, *bounds*, *sound*, *methodName*, *methodParameters*`)`

Posts a message to the specified view to perform a revealing animation on it. However, the system does not actually do the animation until after it calls the method that you specify, in which you must perform any operations required before the animation is done.

distance The number of pixels to animate a portion of the view moving up or down on the screen. Specify a positive number to make the view portion appear to move upward on the screen this number of pixels. To make the view portion appear to move downward, specify a negative number. The *distance* parameter should be the height of the view content you want to reveal (or hide).

bounds The partial area of the view that you want to animate moving up or down. You should specify a `viewBounds` frame using coordinates local to the view to which you are sending this message. The portion of the view that you specify is copied above or below its present position, depending on the setting of *distance*.

sound — A sound frame containing a sound that you want played concurrently with the animation. (If you don't want a sound, specify `nil`.)

methodName — The method that you want called before the animation occurs. You must specify a symbol (for example, `'myScript`). This method must be accessible from the view to which the `RevealEffect` message is sent; that is, this method must reside in that view or be accessible from that view through inheritance.

methodParameters — An array of parameters that are passed to your method.

A revealing effect is like a slide effect, except that it slides just a portion of the view either up or down, while leaving the rest of the view in place. This can be used to create an effect that reveals new information where the portion of the view moved from. The method you specify as a parameter should set up the new information to be revealed so that when the view is redrawn, the new information is visible.

The `RevealEffect` method always returns `nil`.

Here is an example of this method:

```
aView := {...
revealMore: func() // move view portion downwards
   begin
   local vb := view1:LocalBox();
   vb.top := 60; vb.bottom := 80;
   view1:RevealEffect(40,vb,ROM_flip,'myEffect,['dn]);
   end,
closeUp: func() // move view portion upwards
   begin
   local vb := view1:LocalBox();
   vb.top := 60; vb.bottom := 120;
```

```
   view1:RevealEffect(-40,vb,ROM_flip,'myEffect,['up]);
   end,
...}

view1 := {...
myEffect: func(direction)
   begin
   If direction = 'up then // revealing less
      begin
      // Here you would change the view contents so it
      // removes that portion being hidden ...
      end
   Else // revealing more
      begin
      // Here you would change the view contents so it
      // includes the "revealed" information ...
      end
   end,
...}
```

Delete

view:`Delete(`*methodName*`,` *methodParameters*`)`

Posts a message to the specified view to perform an animation on it that crumples the view and tosses it into a trash can that appears on the screen. The view is not actually deleted—only the animation is done.

methodName	The method that actually removes the view or changes it to make it appear deleted. You must specify a symbol (for example, `'myScript`). This method must be accessible from the view to which the `Delete` message is sent; that is, this method must reside in that view or be accessible from that view through inheritance.
methodParameters	An array of parameters that are passed to your method.

The `Delete` method always returns `nil`.

If you want to delete the view or remove the data shown in it, you must do these things yourself in the method you supply. For example, the view may be showing an item from a soup. When the `Delete` animation is performed, you would typically want to clear the data from the view and possibly delete the data from the soup also. Alternatively, you might want to close the view.

Here is an example of this method:

```
aView := {...
// call Delete method
doDeleteEffect: func(whatData)
   textView:Delete('myDelete, [whatData]);
...}

parent_of_textView := {...
myDelete: func(what)
   begin
   //remove data from soup
   EntryRemoveFromSoupXmit(what, kAppSymbol);
   textview:Close(); // close the view being deleted
   end,
...}
```

Dragging a View

Dragging a view means allowing the user to move the view by tapping it, holding the pen down, and dragging it to a new location on the screen. To drag a view, send the view a `Drag` message.

Drag

view`:Drag(`*unit*`,` *dragBounds*`)`

This method is typically called from within a `ViewClickScript` method. It tracks the pen on the display, and drags the view to follow it.

unit The current stroke unit passed by the `ViewClickScript` message.

dragBounds A `bounds` frame describing the area, relative to the root view, within which the view can be dragged. If *dragBounds* is `nil`, the bounds of the entire screen limit the dragging area.

The return values is unspecified.

The display of electronic ink is turned off during the dragging operation.

Here is an example of this view method:

```
draggableView :={...
viewFlags: vVisible + vClickable,
viewClickScript: func(unit)
   begin
   local limits;
   limits := SetBounds(5,50, 230, 305);
   :Drag(unit, limits);

   true;  // return true because we've handled the tap
   end,
...}
```

Dragging and Dropping a Item

The following method is used to drag and drop an item.

DragAndDrop

view`:DragAndDrop(`*unit*`,` *bounds*`,` *limitBounds*, *copy*`,` *dragInfo*`)`

This method is typically sent from the `ViewClickScript`. It starts the drag and drop process and returns when the dragged item(s) is dropped into a view or into the clipboard.

unit The stroke unit received by the `ViewClickScript` method.

bounds The bounds of the item to be dragged, in global coordinates. The bitmap enclosed by the bounds is the bitmap used by the clipboard.

limitBounds Lets you pass in a `bounds` frame, in global coordinates, whose boundaries limit the dragging, so the object cannot be dragged outside of the specified bounds. *limitBounds* has a value of `nil` or a `bounds` frame. A value of `nil` means don't limit the bounds. A `bounds` frame specifies the bounds limits.

copy A Boolean value indicating whether to drag a copy or the original items. Specify non-`nil` to drag a copy or `nil` to move the original items.

dragInfo An array of frames (one frame per dragged item). Each frame has the following slots:

`types`	An array of symbols of the types to which an item can be converted.
`view`	A view object type if the dragged item is a view with a symbol type of `'paragraph`, `'polygon`, `'picture`, and so on).
`dragRef`	Any value that will be passed to other methods.
`label`	An optional string used when the drop is to the Clipboard; it is used as the Clipboard label. If this slot is missing and the item has a `'text` type, the text data is used as the label; otherwise a default label is used.

`DragAndDrop`'s return value can be one of the following:

- `kDragNot = 0` indicates whether the item was actually dragged at all.
- `kDragged = 1` indicates that the item was dragged, but was rejected by the destination.
- `kDragNDropped = 2` indicates that the view was dropped into another container (view).

If you want other views to be able to accept data, these views must implement all of the destination methods. If you have more than one view that can receive a drop, it is easier if you make one drop-aware proto and use it for your other views.

The `DragAndDrop` method sends several messages to both the source view (the view to which `DragAndDrop` was sent) and the destination view (the view that will receive the items). These messages are documented in "Application–Defined Methods" (page 2-65).

Scrolling View Contents

The following methods are used to scroll a view's contents.

SetOrigin

view`:SetOrigin(`*originX*`, `*originY*`)`

Changes the view bounds offset to reflect the new origin point, if it is different from the current origin, and "dirties" the view (so you don't have to send it the `Dirty` message). `SetOrigin` works only on view children.

originX The x coordinate of the new view origin.

originY The y coordinate of the new view origin.

This method always returns `nil`.

This method scrolls the child views of the view to which you send the `SetOrigin` message. The following table shows what parameters to pass to `SetOrigin` to scroll the child views in different directions:

originX	***originY***	**Visual direction**	**Scroll direction**
zero	positive	Up	Down
zero	negative	Down	Up
positive	zero	Left	Right
negative	zero	Right	Left

This method sets the `viewOriginX` and `viewOriginY` slots in the view to the new values you specify.

The view origin determines where, within the view bounds, the actual view contents (child views) are displayed. Initially, the view origin is set to (0, 0). This means that the top-left corner of the view contents (point (0, 0)) is positioned at the top-left corner of the view bounds. If you change the view origin, the view contents are positioned so that the point you specify as the origin is placed at the top-left corner of the view bounds. Thus, the contents are offset within the view. The current view origin coordinates are stored in the slots `viewOriginX` and `viewOriginY` within the view.

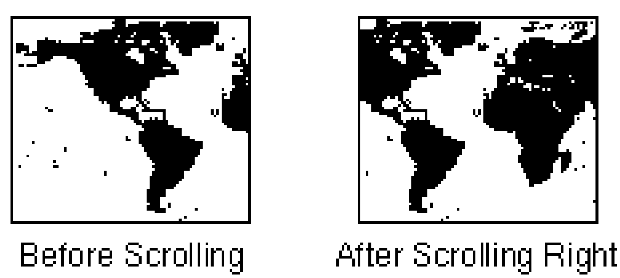
When using `SetOrigin` to scroll a view, you typically want the contents of the view to be clipped to some particular area. For example, you might want to scroll a large map around within a view so that the user can see different parts of the map within the same view. To get this effect, make the parent view smaller than the child (the map, for example) that you want to scroll. The parent view should be as big as the part of the child you want to show at one time.

Set the `vClipping` flag in the `viewFlags` slot of the parent view. When you send the `SetOrigin` message to the parent view, the child view will scroll and be clipped to the bounds of its parent view.

Figure 2-1 shows an example of a world map before and after it has been scrolled. The map is enclosed in a parent view, which is the rectangle around the map. The map was scrolled to the right with this code:

```
parentView:SetOrigin(40,0)
```

Figure 2-1 `SetOrigin` example



Here is an example of using this view method:

```
ParentView := {...
viewFlags: vVisible+vClipping,
viewOriginX: 0,
viewOriginY: 0,
...}
ScrollRightButton := {...
buttonPressedScript: func()
   begin
   parentView:SetOrigin(parentView.viewOriginX+20, 0);
   RefreshViews();
   end,
...}
```

SyncScroll

view:SyncScroll(*what*, *index*, *upDown*)

Scrolls the child views of a view vertically the increment of one child view in the direction indicated.

what — You can specify either an array of view templates or a soup cursor, depending on what kind of data is contained in the view you want to scroll. If all view children are contained in an array, specify the array. If your view data consists of child views created from soup entries, specify the soup cursor.

index — Only used if you specify an array of view templates for *what*. This is the index of the child view template that is currently displayed at the top of the parent view.

upDown — Set to –1 to scroll up (visually, the views move downward on the screen), or set to 1 to scroll down (visually, the views move upward on the screen).

This method has different return values, depending on what you specify for *what*. If you specify an array, this method returns a new array of the child views that are visible within the parent view after scrolling; or, if there is nothing to scroll, `nil` is returned. If you specify a cursor, this method always returns `nil`.

This method plays a "scroll up" or a "scroll down" sound effect, depending on which way the views are scrolling. The sound effect should be stored in the `scrollUpSound` or `scrollDownSound` slot of the view, respectively.

A slot named `height` is required in each of the child views (or soup entries, if you are working with a cursor). This slot should contain the height of the view in its normal (expanded) state.

A slot named `index` is required in the view that receives the `SyncScroll` message (the parent view). Initialize the `index` slot to the index of the child template that is at the top of the parent view when the view is first displayed. Pass the `index` slot for the *index* parameter to `SyncScroll`. The `SyncScroll` method modifies this slot when it scrolls the views, so you don't need to keep track of the index. On each subsequent call to `SyncScroll`, pass the `index` slot for the *index* parameter.

The following information applies only if you specify an array for *what*.

- This method uses two optional slots in the parent view: `allCollapsed` and `collapsedHeight`. These slots control scrolling when the child views have both expanded and collapsed modes. The `allCollapsed` slot should hold a `true` value if all child views are in the collapsed mode, or a `nil` value if all child views are not collapsed. The `collapsedHeight` slot holds the standard, height, in pixels of a collapsed view.
- This method also uses one specific slot in each of the child views: `collapsed`. If there is a `collapsed` slot in a child view, and it holds a `true` value, the individual child view is assumed to be in the collapsed state.

The following information applies only if you specify a soup cursor for *what*.

- This method may or may not move the cursor forward or backward in the soup. Scrolling does not always require advancing to the next or previous view, in which case the cursor would not be changed. For example, a single data item may be longer than the screen space allocated for it in a view, and so tapping the scroll arrow should scroll the view rather than advance to the next data item. In this case, the soup cursor would not be advanced since a new item need not be retrieved from the soup as a result of scrolling.

- Before the scrolling animation is done and the views are redrawn, the `ViewSetupChildrenScript` message is sent to the view that is being scrolled. The view being scrolled must use the `ViewSetupChildrenScript` method to recalculate its `stepChildren` array so that the correct views are displayed when they are redrawn by the `SyncScroll` method.

Working With View Highlighting

These methods and functions are used to highlight a view.

Hilite

view`:Hilite(`*on*`)`

Highlights or unhighlights a view.

on	If non-`nil`, the view is highlighted if it is not already highlighted; if `nil`, the view is unhighlighted.

This method always returns `true`.

HiliteUnique

view`:HiliteUnique(`*on*`)`

Highlights or unhighlights a single view in a group of views.

on	If non-`nil`, highlights the view; if `nil`, the view is unhighlighted.

This method always returns `true`.

The view you specify will be the only view highlighted in its sibling group. That is, any other child views of the same parent that happen to be highlighted are unhighlighted, so that only a single view is highlighted at a time.

TrackHilite

view:`TrackHilite(`*unit*`)`

This method is typically called from within a `ViewClickScript` method. It tracks the pen on the display, highlighting the view when the pen is within its bounds, and unhighlighting the view when the pen is outside it.

unit The current stroke unit passed to the `ViewClickScript` method.

This method returns `true` if the pen is lifted within the view bounds or `nil` if the pen is lifted outside the view bounds.

This method repeatedly sends the `ButtonPressedScript` message to the view while the pen is down and within the view bounds.

The display of electronic ink is turned off while the pen is tracked.

TrackButton

view:`TrackButton(`*unit*`)`

Performs the same operations as `TrackHilite`, but protects against leaving the button highlighted if an error occurs. (The button is unhighlighted if an error occurs during the tracking.)

unit The current stroke unit passed to the `ViewClickScript` method.

This function internally calls `TrackHilite`. It returns non-`nil` if the pen is lifted within the view bounds or `nil` if the pen is lifted outside the view bounds.

Unlike `TrackHilite`, however, this function sends the `ButtonClickScript` message to the view if the pen is lifted within the view bounds of the button.

HiliteOwner

`HiliteOwner()`

Returns the view containing highlighted data. If there is more than one view containing highlighted data, the common parent of those views is returned. However, only one application at a time can have highlighted data. This function returns `nil` if no views contain highlighted data. See “Determining Which View Item Is Selected” (page 3-37) in *Newton Programmer’s Guide* for information on using this function.

This function works only returns views of the class `clEditView` or `clParagraphView`.

GetHiliteOffsets

`GetHiliteOffsets()`

Returns an array of arrays, containing information about views that have highlighted data, even if only text from a single paragraph is selected. If you have a mixed selection; that is, some shapes or sketches and some paragraphs, this function returns nil.

The format is as follows:

`[[`*view1*`, `*startpos1*`, `*endpos1*`], [`*view2*`, `*startpos2*`, `*endpos2*`], ...]`

In the above example, text from the first two paragraphs *view1* and *view2* have been selected. The views in this array are always `clParagraphViews`. In addition, you don’t need to use `HiliteOwner` in conjunction with `GetHiliteOffsets`.

A view can have only one range of highlighted characters. Discontiguous highlighting within a view is not supported. Only one application at a time can have views with highlighted data; so all views returned by this function belong to the same application.

This function works only with views of the class `clParagraphView`. Other kinds of views containing highlighted data (views of the class `clPolygonView`, for example) are not returned.

SetHilite

view:`SetHilite(`*start*`, `*end*`, `*unique*`)`

Highlights some or all of the text in a view of the class `clParagraphView`.

start — The starting character position of the highlighting. A character position of zero indicates the beginning of the view, a position of 1 is after the first character, and so on.

end — The ending character position of the highlighting.

unique — A Boolean value. Specify non-`nil` to make the specified text the only highlighted text in the view; any other highlighted text is unhighlighted. Specify `nil` to allow previously highlighted text to stay highlighted. In the later case, the highlighting is extended to include the newly specified highlighted text. Discontiguous highlighting is not allowed.

This function returns `true`, unless *view* is invalid, in which case `nil` is returned.

Creating View Dependencies

The following functions are used to make one view dependent on another.

TieViews

`TieViews(`*mainView*`, `*dependentView*`, `*methodName*`)`

Makes one view dependent on another so that when the main view changes, it notifies the dependent view by sending a message to the dependent view.

mainView — The main view.

dependentView — The view that you want to be notified when *mainView* changes.

methodName — A symbol that is the name of the method to call in *dependentView* when *mainView* changes. This method is passed two parameters when it is called. The first parameter is a reference to the view that changed and the second parameter is a symbol that is the name of the slot that changed.

This function returns non-`nil` if it successfully registers the dependent view with the main view; otherwise, it returns `nil`.

You can pass in the following special symbols (which are evaluated at run time) for either the *mainView* or *dependentView* parameters:

- `'viewFrontMost`, indicates the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot.
- `'viewFrontMostApp`, indicates the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot, but not including floating views (those with `vFloating` set in their `viewFlags` slot).
- `'viewFrontKey`, indicates the view on the screen that accepts keys (there can be only one view that is the key receiver).

Here is an example of two views of the `clParagraphView` class. Any text entered in the first view is duplicated in the second:

```
mainView := {...
viewClass: clParagraphView,
viewFlags: vVisible+vClickable+vStrokesAllowed+
           vGesturesAllowed+vCharsAllowed,
ViewSetupFormScript: func()
   begin
   TieViews(mainView, tieView, 'ItChanged);
   end,
...}
tieView := {...
viewClass: clParagraphView,
viewFlags: vVisible,
ItChanged: func(view, slot)
   begin
   local newtext := view.text;
   setvalue(self, 'text, newtext);
   end,
...}
```

Synchronizing Views

The following two methods are used to synchronize views.

RedoChildren

view:`RedoChildren()`

Closes, then reopens and redraws, all of a view's child views. This method always returns `true`.

As a result of the `RedoChildren` message, the following system actions occur:

1. The child views are sent `ViewQuitScript` messages, and then they are closed.
2. The parent view (the view to which you sent the `RedoChildren` message) is sent the `ViewSetupChildrenScript` message, and the child templates are reread from the `viewChildren` and `stepChildren` slots of the parent view.
3. The child views are reopened, and in this process are sent the following messages: `ViewSetupFormScript`, `ViewSetupChildrenScript`, `ViewSetupDoneScript`.
4. The parent view, and then the child views, are drawn and sent the `ViewDrawScript` message.

For more information about system messages, refer to "Application–Defined Methods" (page 2-65).

Note that because the `RedoChildren` method closes child views, any new data that you have stored in those views during run time will be lost. For example, if you have created a slot in a child view and stored a value in it, that slot and value will be lost when the view is closed and reopened. The view is reopened directly from its template, so of course, any data that was in the view memory object in RAM is lost.

However, if a child view is declared in a view that is still open (typically the parent view), then, even though the child view is closed, its view memory object is not destroyed and any data stored in the view is preserved. This is the same as when you send the `Close` message to a declared view. For more information about declared views, see "View Instantiation" (page 3-26) in *Newton Programmer's Guide*.

Because the `RedoChildren` method closes and reopens all child views, it is relatively slow. If you know that some of the child views are still visible within the parent, you can use `SyncChildren` instead, which gives better performance since it doesn't close views that are still visible.

SyncChildren

view`:SyncChildren()`

Redraws all of a view's child views, with their new bounds, if the bounds have changed. This method always returns `true`.

As a result of the `SyncChildren` message, the following system actions occur:

1. The `ViewSetupChildrenScript` message is sent to the view to which the `SyncChildren` message was sent.
2. The child views are synchronized with the `stepChildren` and `viewChildren` arrays of the parent view to which this message was sent. If a view is no longer listed in the `stepChildren` or `viewChildren` array, then the `ViewQuitScript` message is sent to it and it is closed. If a new view template is listed in one of these arrays, the new child view is created and opened. As a result of its opening, the new view is sent the usual messages: `ViewSetupFormScript`, `ViewSetupChildrenScript`, and `ViewSetupDoneScript`.
3. Internally, the system does a `SyncView` for each of the child views. As a result, the `ViewSetupFormScript` message is sent to each child view, and each view whose bounds has changed is redrawn.

Note that if a new child view is created, it receives the `ViewSetupFormScript` message twice, once in step 2 and once in step 3.

The view to which you send the `SyncChildren` message is not dirtied. Usually this is not a problem, except in one case, in which you must send the view the `Dirty` message to cause it to be redrawn. If a child view is closed in step 2 and another child view is not drawn completely over it, the old child view will still be visible.

Here is an example of using the `SyncChildren` method:

```
{...
addOneChild: func(childTemplate)
   begin
   // ensure that stepChildren array is in RAM
   if not HasSlot(self, 'stepChildren) then
      self.stepChildren := Clone(self.stepChildren);
   // add new template into the array
   AddArraySlot(self.stepChildren, childTemplate);
   // sync up the views
   self:SyncChildren();
   end
...}
```

Laying Out Multiple Child Views

The following methods are used to layout multiple child views.

LayoutTable

view`:LayoutTable(`*tableDefinition*`, `*columnStart*`, `*rowStart*`)`

Generates a table where each cell is a child of the parent view to which this message is sent. This method essentially calculates the bounds for each child view so that the children are laid out in a table-like format in the parent.

tableDefinition	A frame describing the table. The slots are described later in this method description.
columnStart	The column number of the cell that should be placed in the upper-left corner of the parent view. Specify an integer from zero (for the first column) to one less than the total number of columns.
rowStart	The row number of the cell that should be placed in the upper-left corner of the parent view. Specify an integer from zero (for the first row) to one less than the total number of rows.

This method returns an array of child templates that can be used as the value of the `stepChildren` slot in the parent template.

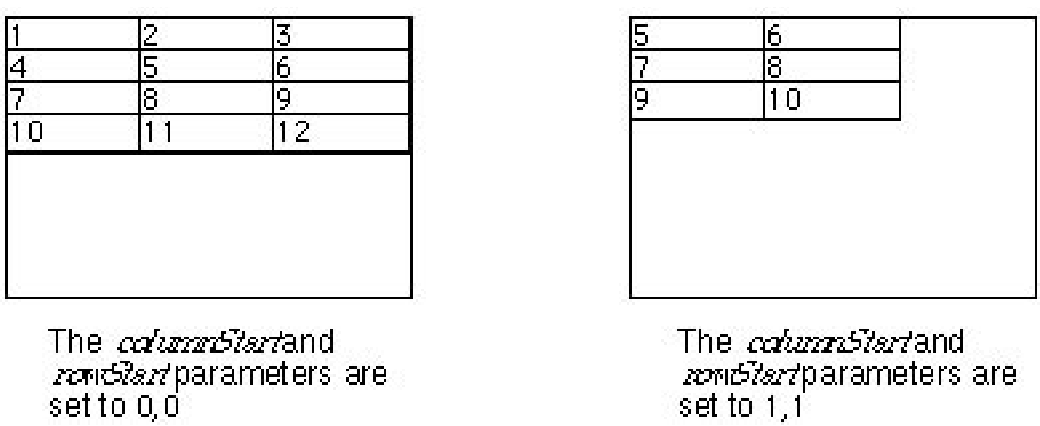
The `viewBounds` slots of the children are calculated so that the first child is placed in the upper-left corner of the parent view. You can use the *columnStart* and *rowStart* parameters to change which child is the first child. By using these parameters to specify a different upper-left cell, you can display just a portion of the entire table.

For example, to generate templates for all cells in a table, specify 0, 0 for *columnStart* and *rowStart*. This places the top-left cell in the table in the top-left corner of the parent view. This is illustrated in the first view shown in Figure 2-2.

To offset the table upward and to the left, specify 1, 1. This places the second cell in the second row in the top-left corner of the parent view. This is illustrated in the second view shown in Figure 2-2. Note, however, that cells are laid out sequentially beginning with the indicated cell. That is, cells 5 through 10 are all shown. The table isn't simply shifted up and to the right.

Templates are not generated for cells that precede the starting cell. The first template in the array returned by `LayoutTable` is the template for the first cell indicated by *columnStart* and *rowStart*.

Figure 2-2 `LayoutTable` results



TableDefinition slots

`tabAcross` The number of columns in the table.

`tabDown` The number of rows in the table.

`tabWidths` An integer giving the fixed width of the columns, in pixels, or an array of column widths.

`tabHeights` An integer giving the fixed height of the rows, in pixels, or an array of row heights.

`tabProtos` A reference to a template used in creating the child views, or an array of references to templates. The array elements are mapped to the table of views beginning at the top-left cell of the table and continuing down the first column, and then down the second column, and so on. If there are fewer array elements than table cells, after the last array element is mapped, the mapping continues with the first element.

`tabValues` A value that is used as the value of each of the child views. Alternately, an array of values that are mapped to table cells as above.

`tabValueSlot` A symbol naming the slot in each of the child views where its view value (specified in `tabValues`) is stored. (Remember to quote the symbol; as with `'text`.) For example, if the table consists of child views based on the `clParagraphView` class, you would specify `'text` for this slot, since the value of a `clParagraphView` is stored in the `text` slot.

`tabSetup` A method that is called before each of the child views is instantiated. It is passed three parameters: a reference to the child template, its column number in the table, and its row number in the table. This allows you to do special initialization operations to each child view before it is instantiated. This method must be passed the context with the call.

The following example of `LayoutTable` method shows the code used to generate the first table in Figure 2-2:

```
{...
viewclass: clView,
viewBounds: {left: 42, top: 26, right: 193, bottom: 129},

tabAcross: 3,
tabDown: 4,
tabWidths: nil,
tabHeights: nil,
tabProtos:{viewclass: clParagraphView,
   viewBounds: nil,
   viewJustify: vjLeftH+vjCenterV+oneLineOnly,
   viewFlags: vVisible+vClickable,
   viewFormat: vfFillWhite+vfFrameBlack+vfPen(1),
   text:nil,
   viewFont: simpleFont10},
tabValues: nil,
tabValueSlot: nil,

ViewSetupChildrenScript: func()
   begin
   local box, cells;
   box := self:localBox();
   viewWidth := box.right - box.left;
   tabWidths := viewWidth DIV tabAcross;
   tabHeights := FontHeight(tabProtos.viewFont);
   tabValues := ["1", "2", "3", "4", "5", "6", "7", "8",
                 "9", "10", "11", "12"];
   tabValueSlot := 'text;
   self.stepChildren := self:LayoutTable(self, 0, 0);
   end,
...};
```

LayoutColumn

view:`LayoutColumn(`*childViews*`,` *index*`)`

In the view to which this message is sent (the main view), `LayoutColumn` displays a subset of views from a larger array of views.

childViews — The array of views from which you want to display a subset.

index — The index of the view in the *childViews* array that you want to display at the top of the view to which you send this message.

This method returns a reference to an array of child views that fill the bounds of the main view, beginning with the view at *index* and containing as many subsequent views as it takes to fill the main view to the bottom. Each child view must have a height slot that is set to the height of the view in pixels.

Miscellaneous View Operations

This section describes other miscellaneous view methods and functions.

SetPopup

view:`SetPopup()`

After a view is shown, call this method to make the view a pop-up view (a picker); that is, a view that gets closed on the next pen tap (whether inside or outside of it). An example of using this feature is in the `protoPicker` view proto (page 5-13).

This method always returns `nil`.

Here's how you would typically call this method in your view template:

```
viewSetupDoneScript: func()
   self:SetPopup();
```

GetViewFlags

`GetViewFlags(`*template*`)`

Returns the value of the `viewFlags` slot in the view corresponding to the specified template, or in the template itself, if its view has not yet been instantiated.

template The template or view whose `viewFlags` slot you want to get.

You can pass in the following special symbols for the *template* parameter:

- `'viewFrontMost`, indicates the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot.
- `'viewFrontMostApp`, indicates the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot, but not including floating views (those with `vFloating` set in their `viewFlags` slot).
- `'viewFrontKey`, indicates the view on the screen that accepts keys (there can be only one view that is the key receiver).

These symbols are evaluated at run time.

Visible

`Visible(`*view*`)`

This function tests a view to see if it is visible or not. This function returns non-`nil` if the view is visible or `nil` if the view is not visible. Note that a view can be open but not visible, so this function is not a valid test of whether a view is open.

view The view that should be tested to see if it is visible.

You can pass in the following special symbols for the *view* parameter:

- `'viewFrontMost`, indicates the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot.
- `'viewFrontMostApp`, indicates the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot, but not including floating views (those with `vFloating` set in their `viewFlags` slot).

- `'viewFrontKey`, indicates the view on the screen that accepts keys (there can be only one view that is the key receiver).

These symbols are evaluated at run time.

ViewIsOpen

`ViewIsOpen(`*view*`) //platform file function`

Returns `true` if the view is open and `nil` if it is not.

IMPORTANT

This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

`call kViewIsOpenFunc with (`*view*`);`

▲

view	The view you wish to check.

Note that a view can be open but not visible (if it is hidden).

This function is a better way to check if a view is open, rather than checking if the `viewCObject` slot is non-`nil`.

Application–Defined Methods

The following subsections describe application-defined methods. When using any of these methods, always call `inherited:?ViewXXXScript` when using protos or in case the present or future system software provides such a method.

ButtonToggleScript

view`:ButtonToggleScript(`*frontmostApp*`)`

Lets the application perform special handling when its icon is tapped in the Extras Drawer.

frontmostApp	The base view of the application that is frontmost on the screen.

The value that the application returns from the `ButtonToggleScript` method is important. It can return either `nil` or non-`nil`. A return value of `nil` means that the system should proceed with the normal operations that it does when an icon is tapped. A value of non-`nil` means that the system should do nothing — the assumption being that the application handled the icon tap in whatever way it wanted to itself.

ViewSetupFormScript

view`:ViewSetupFormScript()`

During view creation, this message is sent before any slots in the view template are read. In this method, you can do any special initialization that your view needs, including setting the value of any slots other than the `viewClass` slot. For example, you can dynamically change the `viewBounds` slot, the `viewFlags` slot, the `viewFont` slot, and so on. Note that you cannot perform any operations involving child views of your view since they haven't yet been instantiated at this point. (However, you can manipulate the `stepChildren` array at this point.) The return values is unspecified.

This message is also sent during execution of the system view method `SyncView`, before it begins its operations. It is sent during execution of the global function `SetValue` (it calls `SyncView` internally), if you set the value of one of these slots: `viewBounds`, `viewFormat`, `viewJustify`, or `viewFont`.

Here is an example of using this method:

```
ViewSetupFormScript: func()
   begin
   self.viewBounds := SetBounds(0, 15, 200, 180);
   end
```

Note

The system searches for this method only in the current view and its protos. The parent chain is not searched for the method. ◆

ViewSetupChildrenScript

view:`ViewSetupChildrenScript()`

This message is sent after a view is created but before its children are instantiated. In this method, you can do any special initialization that you need to do before the child views are instantiated. For example, you might want to dynamically set up the `stepChildren` array, which controls what child views are to be created. The return values is unspecified.

This message is also sent during execution of the following system view methods before the child views are redrawn: `SyncChildren`, `RedoChildren`, and `SyncScroll` (only if you pass a soup cursor for the first parameter in `SyncScroll`).

Here is an example of using this method:

```
ViewSetupChildrenScript: func()
   begin
   self.stepChildren := [pg4, pg5]; // child templates
   end
```

Note
The system searches for this method only in the current view and its protos. The parent chain is not searched for the method. ◆

ViewSetupDoneScript

view:`ViewSetupDoneScript()`

This message is sent after all of the child views of the view are instantiated, just before the view is displayed. `ViewSetupDoneScript` is sent for children before it is sent for the parents of the children. The return values is unspecified.

Here is an example of using this method:

```
ViewSetupDoneScript: func()
   begin
   self:SetPopup();
   end
```

Note
The system searches for this method only in the current view and its protos. The parent chain is not searched for the method. ◆

ViewQuitScript

view:`ViewQuitScript()`

This message is sent just before the view is closed. It gives you a chance to do any processing or clean-up that you need to just before the view is closed.

Note that an undeclared view is destroyed when it is closed. A declared view still exists, if the view in which it is declared is still open. A view can get control after all of its children have been destroyed.

When a view is closing, this message is sent to the topmost view that is closing as well as to all of the children of that view, since they too are closing with it. That is, the first child view receives this message, then all of its children, in order, and then the second child view receives this message, and so on. For each child view, the message is sent recursively to all of its children before the next top-level child is notified.

The child views are closed in reverse order. That is, the views at the bottom of the hierarchy are closed first, then those above them, and so on, until the original view receiving the `ViewQuitScript` message is closed last.

If you return the symbol `'postQuit` from the `ViewQuitScript` method of a view, that same view will then be sent the `ViewPostQuitScript` message after all of its child views have been destroyed. This allows you an opportunity to do extra clean-up, if necessary. See `ViewPostQuitScript` (page 2-69) for additional details.

Note that you can't send any view messages to a view whose `ViewQuitScript` has already executed. If you do, the system throws an exception.

IMPORTANT

If you override the `ViewQuitScript` of any proto, you must return the value of the expression `inherited:?ViewQuitScript`. Otherwise, if there is a `ViewPostQuitScript` method in the proto, it may not be executed. Even if current protos don't use the `ViewPostQuitScript` feature, they may in the future. ▲

Here is an example of this method:

```
ViewQuitScript: func()
   begin
   RemoveSlot(GetRoot(), 'businessFormat);
   RemoveSlot(GetRoot(), 'myAuxFormat);
   inherited:?viewQuitScript();
   end
```

Note

The system searches for this method only in the current view and its protos. The parent chain is not searched for the method. ◆

ViewPostQuitScript

view:`ViewPostQuitScript()`

This message is sent to a view following the `ViewQuitScript` message and after all of the view's child views have been destroyed. This message is not automatically sent to all views, but is sent only if the `ViewQuitScript` method returns the symbol `'postQuit`. See `ViewQuitScript` (page 2-68) for more information.

Note that when a view receives the `ViewPostQuitScript` message, it is not actually a full-fledged view anymore, but only the remnants of its view frame. This means that from within the `ViewPostQuitScript` method, you can't send any view messages to `self`; however, the parent view is still valid, so the children can still send messages to the parent view.

ViewShowScript

view:`ViewShowScript()`

This message is sent when the view is instructed to show itself; it is not sent to any child views. This can occur as a result of the `Show`, `Open`, or `Toggle` messages. When showing a view, the view system first shows the view and then sends this message to allow you to perform any additional operations. The return value is ignored.

Here is an example of using this method:

```
ViewShowScript: func()
   begin
   // idle method will close view after 5 seconds
   :SetupIdle(5000);
   end
```

ViewHideScript

view:`ViewHideScript()`

This message is sent when the view is instructed to hide itself. This can occur as a result of the `Hide`, `Close`, or `Toggle` view methods. When hiding a view, the view system first sends this message, then hides the view and all of its child views. However, this message is not sent to any of the child views. The return value is ignored.

This message is not always sent when a view is closed. Do not use this method to do clean-up when a view is closing—use the `ViewQuitScript` method instead. The `ViewQuitScript` message is sent immediately after the `ViewHideScript` message when a view is being closed.

Here is an example of this method:

```
ViewHideScript: func()
   begin
   // open anotherView when this one is hidden
   anotherView:Open();
   end
```

ViewDrawScript

view:`ViewDrawScript()`

This message is sent when the view is drawn. First the view system draws the view, this message is sent, and the view frame and view highlighting (if any) are drawn. This message is sent before any child views are drawn. If you wish to augment the drawing done by the view system or to perform other operations whenever the view is drawn, do it in this method.

If you want to draw in a view other than when the `ViewDrawScript` message is sent, use the `DoDrawing` view method, documented in “Drawing and Graphics Reference” (page 10-1)

▲ **WARNING**
All coordinates in the `viewBounds` slot and the global coordinates of the bounds, such as returned by `GlobalBox`, of a view must be within the range -32768 to 32767. If this is not the case, the behavior of the views and view scripts are undefined. ▲

Here is an example of using this method:

```
ViewSetupFormScript: func()
   // set up line objects and save them in the lines slot
   begin
   local box;
   box := self:LocalBox();
   self.lines[MakeLine(0, 0, box.right, box.bottom),
   MakeLine(0, box.bottom, box.right, 0)];
   end

ViewDrawScript: func()
   // draw an X in the view
   begin
   :DrawShape(self.lines, nil);
   end
```

Note
The system searches for this method only in the current view and its protos. The parent chain is not searched for the method. ◆

ViewHiliteScript

view:`ViewHiliteScript(`*on*`)`

This message is sent just before the system is about to highlight or unhighlight the view.

on — A Boolean value that is non-`nil` if the view is to be highlighted or `nil` if the view is to be unhighlighted.

The return values is unspecified, it is assumed that you have handled the highlighting or unhighlighting operation, and the system won't do it. If this method returns `nil`, the system performs the operation.

Note that you don't have to use the `DoDrawing` method to draw in your `ViewHiliteScript` method.

Here is an example of this method:

```
ViewHiliteScript: func(on)
   begin
   local box;
   box := self:LocalBox();
   r := MakeRoundRect(box.left+3, 0, box.right-3,
                      box.bottom, 4);
   :DrawShape(r, {transferMode: modeXor,
                  fillPattern: vfBlack});
   true;
   end
```

Note
The system searches for this method only in the current view and its protos. The parent chain is not searched for the method. ◆

ReorientToScreen

view:`ReorientToScreen()`

This message is sent to each child of the root view when the screen orientation changes. It is sent to validate a view as supporting landscape or rotation or it is sent to a view during rotation so that the view can adjust itself appropriately. The return values is unspecified.

▲ **WARNING**

An application must have a `ReorientToScreen` method in order to be opened on a landscape screen. If a user tries to open an application that doesn't have this method, a slip is displayed to give the user the option of not opening the application at all or rotating the screen back to portrait before it is opened. ▲

When the screen orientation changes, the system checks each child of the root view to see if the `ReorientToScreen` method exists. If this slot exists, `ReorientToScreen` is sent to each child view and the rotation occurs. If it doesn't exist, a slip appears warning the user that some functions will not show after rotation because they can't operate while rotated. The slip contains a "Cancel" and "OK" button. If the user taps "Cancel" the rotation is cancelled and nothing happens. If the user taps "OK," any view that doesn't implement the `ReorientToScreen` method is closed and the rotation occurs.

To support rotation, your application should implement this method in its base view or any other view that will be a child of the root view. `ReorientToScreen` should resize, move, or close your application. The easiest way to implement this behavior is take advantage of the default function provided by the ROM by placing the function `ROM_DefRotateFunc` in your `ReorientToScreen` slot as in this example:

```
ReorientToScreen: ROM_DefRotateFunc
```

If the view is offscreen, any `viewbounds` slot in the view frame is also removed. This behavior restores the view to its default position if the user has dragged it.

A more sophisticated way of handling rotation in the `ReorientToScreen` method is to use the `GetAppParams` function to check the new screen dimensions, and then resize and redisplay the base application view and all child views, if necessary.

ViewScrollDownScript

view:`ViewScrollDownScript()`

This message is sent when the view system receives a scroll down event, which occurs when the user taps the downward-pointing scroll arrow. There is no default view-system operation that occurs as a result of this event—only this message is sent. Note that “scrolling down” means that visually the items on the screen move upward, showing you new items that were previously hidden “below” the bottom of the view.

Only a view with the `vApplication` flag set in its `viewFlags` slot can receive this message.

Here is an example of this method:

```
ViewScrollDownScript: func()
   begin
   if index < length(notes)-1 then
      begin
      roll:SyncScroll(notes, index, 1);  // 1 = down
      index := index + 1;
      end
   end
```

ViewScrollUpScript

view:`ViewScrollUpScript()`

This message is sent when the view system receives a scroll up event, which occurs when the user taps the upward-pointing scroll arrow. There is no default view-system operation that occurs as a result of this event—only this message is sent. Note that “scrolling up” means that visually the items on the screen move downward, showing you new items that were previously hidden "above" the top of the view. The return values is unspecified.

Only a view with the `vApplication` flag set in its `viewFlags` slot can receive this message.

Here is an example of this method:

```
ViewScrollUpScript: func()
   begin
   if index > 0 then
      begin
      roll:SyncScroll(notes, index, -1);  // -1 = up
      index := index - 1;
      end
   end
```

ViewOverviewScript

view:`ViewOverviewScript()`

This message is sent when the view system receives an overview event, which occurs when the user taps the overview dot between the scroll arrows. There is no default view-system operation that occurs as a result of this event—only this message is sent. The return values is unspecified.

Usually the overview button is used to toggle between two views of the data in an application: a close-up (normal) view, and an overview view.

Only a view with the `vApplication` flag set in its `viewFlags` slot will be sent this message.

Here is an example of this method:

```
ViewOverviewScript: func()
   begin
   if (cardPrefs.mode = modeCloseUp) then
      cardPrefs.mode := modeOverview
```

```
else
   cardPrefs.mode := modeCloseUp;
closeUp:Toggle();
overView:Toggle();
status:RedoChildren();
end
```

ViewAddChildScript

view:`ViewAddChildScript(`*child*`)`

This message is sent when a child is about to be added to a view of the `clEditView` class.

child The child template to use to create the child view.

This method gives you a chance to create and add the child view, or to do some other processing before the view is created and added automatically.

If this method returns non-`nil`, it is assumed that you have added the child view entry to your view's `stepChildren` array and have created the child view. If this method returns `nil`, these things are done for you.

In any case, a view must be instantiated from the template passed to this method—either by you or by the system. If you return non-`nil`, and fail to instantiate the view, the system displays an error message, because it expects the view to exist.

Here is an example of using this method:

```
ViewAddChildScript: func(child)
   begin
   AddToDefaultStore(mySoup, child);
   AddUndoAction(KillAddition, [child]);
   AddView(myView, child);
   end
```

Note
The system searches for this method only in the current view and its protos. The parent chain is not searched for the method. ◆

Use this method if you have a `clEditView` that is creating paragraph and polygon child views with the `vNoScripts` flag set, and you want to override the `viewFlags` slot to remove the `vNoScripts` flag.

ViewChangedScript

view`:ViewChangedScript(`*slot*, *view*`)`

This message is sent when the value of a slot in the view is changed as a result of the `SetValue` function, or as a result of other view operations such as changing the bounds, changing the contents or the text style, and so on. The return values is unspecified.

slot	A symbol that is the name of the slot whose value changed.
view	The view that slot resides in.

Here is an example of this method:

```
ViewChangedScript: func(slot, view)
   begin
   if slot = 'text then
      textChanged := true; //set flag if text was changed
   end
```

ViewDropChildScript

view`:ViewDropChildScript(`*child*`)`

This message is sent when a view of the `clEditView` class is about to remove a child view.

child	The child view to remove.

This method gives you a chance to remove the child view entry from your view's `viewChildren` array, or to do some other processing before the view is removed.

The return values is unspecified, it is assumed that you have removed the child view entry from your `viewChildren` array. If this method returns `nil`, this is not assumed and it is done for you. In either case, the child view itself is deleted for you by the system. (Note that you can use the `RemoveView` function to delete the view yourself.)

Here is an example of this method:

```
ViewDropChildScript: func(child)
   begin
   EntryRemoveFromSoupXmit(child, kAppSymbol);
   base:RedoChildren();
   nil;
   end
```

Note
The system searches for this method only in the current view and its protos. The parent chain is not searched for the method. ◆

ViewIdleScript

view:`ViewIdleScript()`

When an idler is installed for a view, this message is sent repeatedly and at regular intervals when the view is open, to allow you to do periodic tasks such as polling a network for information or updating a clock on the display.

You install an idler for a view by sending it the `SetupIdle` message, which specifies an initial delay after which the `ViewIdleScript` message is sent. The `ViewIdleScript` method returns an integer which specifies the delay, in milliseconds, until it is called again. For example, to have the system call this method every second, you should return 1000.

To stop idling, you can return the value `nil`, or you can send the view the `SetupIdle` message with a value of zero.

There is no default view-system operation that occurs during idling—only the `ViewIdleScript` message is sent.

Note
When you install an idler for a view, the time that the `ViewIdleScript` message will next be sent is not guaranteed to be the exact interval you specify. This is because the idler may be delayed if a method is executing when the interval expires. The `ViewIdleScript` message cannot be sent until an executing method returns.

Do not install idlers that use repeated intervals of less than 100 milliseconds. ◆

Here is an example of this method:

```
ViewShowScript: func() // initialize blinking sequence
   begin
   icon := onBitmap;
   self.numBlinks := 0;
   self:SetupIdle (750); // start in 3/4 second
   end

ViewIdleScript: func()
   begin
   if icon = onBitmap then
      icon := offBitmap;
   else begin
      icon := onBitmap;
      numBlinks := numBlinks + 1;
   end;
   self:Dirty();
   if numBlinks < 4 then // blink 4 times
      return 750; // return time until next blink
```

```
numBlinks := 0; // else return 0 to stop blinking
end
```

This example blinks an icon in a view of the `clPictureView` class when the view is shown.

Be careful not to send this message too frequently for long periods of time (for example, many times each second for a few minutes). This causes the Newton hardware to consume significantly more power than usual and reduces battery life.

Note

The system searches for this method only in the current view and its protos. The parent chain is not searched for the method. ◆

ViewDrawDragDataScript

sourceView:`ViewDrawDragDataScript(`*bounds*`)`

bounds The bounds that were passed to `DragAndDrop`.

If supplied, this method draws the image that will be dragged. The default (if this method is missing) is to use the area of the screen inside the rectangle defined by *bounds* parameter to `DragAndDrop`.

This method returns a Boolean value. Returning non-`nil` means that this method handled the drawing. Returning `nil` means that the default behavior should take place.

ViewDrawDragBackgroundScript

sourceView:`ViewDrawDragBackgroundScript(`*bounds*`,` *copy*`)`

bounds The *bounds* parameter as passed to `DragAndDrop`.

copy The *copy* parameter as passed to `DragAndDrop`.

If supplied, this method draws the image that appears behind the dragged data. The default (if this method is missing or if it returns `nil`) is to use the bitmap of the area inside the rectangle defined by *bounds* XORed with the bitmap from ViewDrawDragDataScript. Note that the XOR happens only if *copy* is nil.

This method returns a Boolean value. Returning non-nil means that this method handled the drawing. Returning nil means that the default behavior should take place.

ViewGetDropTypesScript

destView:ViewGetDropTypesScript(*currentPoint)*

Returns an array of symbols; that is, the data types accepted by the view at the location *currentPoint*. For example, 'text or 'picture. The array is sorted by priority (preferred type first). This method can return nil, meaning no drop is allowed at the current point.

currentPoint	The current pen position in global coordinates (a frame containing x and y slots).

ViewFindTargetScript

*destView:*ViewFindTargetScript(*dragInfo*)

Lets the destination view redirect the drop to a different view. ViewFindTargetScript returns a view frame of the view that gets the drop messages. It is called right after the ViewGetDropTypesScript.

dragInfo	An array of frames (one frame per dragged item). See DragAndDrop (page 2-46) for a list of approved slots.

ViewDropApproveScript

*sourceView:*ViewDropApproveScript(*destView*)

Provides a way for the *sourceview* to disallow dropping onto a particular view. ViewDropApproveScript returns nil if the drop shouldn't happen, and non-nil if the drop should happen. It is called only if the drop types match up with the dragged data and the *destView*, and is called right before the ViewDropScript, ViewDropMoveScript and/or ViewDropRemoveScript methods are called.

destView	Destination view in which the dropping will occur.

ViewDragFeedbackScript

destView`:ViewDragFeedbackScript(`*dragInfo*`, `*currentPoint*`, `*show*`)`

Allows a view to give visual feedback while items are dragged over it.

dragInfo	The same parameter passed to `DragAndDrop` (page 2-46).
currentPoint	The current pen position in global coordinates (a frame containing x and y slots).
show	A Boolean value indicating whether to show or hide the feedback. Specify non-`nil` to show the feedback or `nil` to hide it. Hiding the feedback means erasing any highlighting drawn when *show* is non-`nil`, so the view appears normally.

This method returns a Boolean value. Returning non-`nil` means that the method did draw. Returning `nil` means that no feedback was drawn, so this method does not need to be called again with *show* `nil` at the point *dragPoint*. The return value is ignored if *show* is `nil`.

This method is always called with *show* set to `nil` after it's called with show set to non-`nil`. This action ensures that your function is called twice for every "point" being dragged. An example use is drawing your drag feedback with the XOR drawing mode. By calling `ViewDragFeedbackScript` a second time, the view can ensure that it was using the *dragPoint* when drawing and can XOR the exact image onto the screen again. The screen will then show the original pixels.

Alternately, if no "drawing" occurred during `ViewDragFeedbackScript`, return `nil` and the script won't be called again.

Note that XORing is not required as a draw mechanism. The view might be saving part of the screen to an offscreen bitmap and drawing feedback. Then when asked to hide the feedback (show is set to `nil`), it could restore the original image from the offscreen bitmap.

ViewGetDropDataScript

src: `ViewGetDropDataScript(`*dragType*`,` *dragRef*`)`

Called when a destination view that accepts all the dragged items is found. `ViewGetDropDataScript` is called for each item being dragged.

dragType	The type accepted by the destination view for this particular item as passed to `DragAndDrop` in the *dragInfo* array.
dragRef	The drag reference for this item as passed to `DragAndDrop` in the *dragInfo* array.

`ViewGetDropDataScript` returns a frame containing the actual data to be dropped into the destination view. This data could be any frame (not necessarily a view). The frame should contain a `text` slot if the required type is `'text`, a `points` slot if the required type is `'polygon`, an ink slot if the required type is `'ink`, or an `icon` slot if the required type is `'picture`. For `polygon`, `ink`, or `picture` types, the frame should also contain a `viewBounds` slot in the *src* view coordinates.

If the dragged item is a view—that is, the view slot was set in the `dragType` array element passed to `DragAndDrop`—the default behavior occurs by returning a frame with the necessary slots unless the `ViewGetDropDataScript` returns a non-`nil` value.

If you want to drag system data types to or from a plain view, use these formats for the built in types:

dragType	RequiredSlots	Optional slots
`'text`	`text` `viewBounds`	any other `clParagraphView` slots
`'polygon`	`points` `viewBounds`	any other `clPolygonView` slots
`'ink`	`ink` `viewBounds`	any other `clPolygonView` slots
`'picture`	`icon` `viewBounds`	any other `clPictureView` slots

Note
The `viewBounds` slot is no longer necessary for text type. However, if the `viewBounds` slot exists, it will be used. ◆

ViewDropScript

destView:`ViewDropScript(`*dropType*`,` *dropData*`,` *dropPt*`)`

This message is sent to the destination view for each dragged item.

dropType — One of the types that the destination view returns from the `ViewGetDropTypesScript` method.

dropData — The frame that the source view returns from the `ViewGetDropDataScript` method. If this frame has a `viewBounds` slot, this slot is converted to be in destination view coordinates before calling `ViewDropScript`.

dropPt — The last stroke point in global coordinates (a frame containing `x` and `y` slots).

This method returns a Boolean value. Returning non-`nil` means that this method handled the drop. Returning `nil` means that the drop is not accepted.

Note that this method posts an undo action, if necessary.

ViewDropMoveScript

sourceView:`ViewDropMoveScript(`*dragRef*`,` *offset*`,` *lastDragPt*`,` *copy*`)`

This message is sent for each dragged item when dragging and dropping in the same view. (In this case, `ViewGetDropDataScript` and `ViewDropScript` messages are not sent.)

dragRef — The drag reference for this item (as passed to `DragAndDrop` in the *dragInfo* array).

offset — A frame with `x` and `y` slots indicating the horizontal and vertical offsets of the item.

lastDragPt The last stroke point in global coordinates (a frame containing `x` and `y` slots).

copy The *copy* parameter as passed to `DragAndDrop`.

This method returns a Boolean value. Returning non-`nil` means that this method handled the move. Returning `nil` means that the move was not done.

Note that this method posts an undo action if necessary.

ViewDropRemoveScript

sourceView`:ViewDropRemoveScript(`*dragRef*`)`

This message is sent for each dragged item when the *copy* parameter to `DragAndDrop` is `nil`.

This method removes the item from the source view.

dragRef The drag reference for this item (as passed to `DragAndDrop` in the *dragInfo* array).

This method returns a Boolean value. Returning non-`nil` means that this method handled removing the item. Returning `nil` means that the remove operation was not done.

Note that if you are using your own drop types and your own scripts, an undo action must be added to this method for this part of the operation.

ViewDropDoneScript

destView:`ViewDropDoneScript()`

Sent at the very end of each drag and drop to let the destination view know that all specified items have been dropped or moved.

View Warning Messages

The warnings in Table 2-7 are printed to the inspector when a NewtonScript application calls a view method in situations where the requested operation is unwise, unnecessary, ambiguous, invalid, or just a bad idea. The function or method typically does nothing other than print this warning message, but such messages point out situations where code needs to be changed since these calls may very well not be supported in the future.

In the future, you might get an exception thrown instead of just this error message, or something more serious could occur since the problem might not be detected.

If the global variable `noEvilLiveOn` is set to `true`, a breakloop is entered, which helps to find out exactly which view is causing the problem. Setting `noEvilLiveOn` also causes other "incompatibility" errors to enter a breakloop.

Table 2-7 View warning messages

Error number	Message
4711	`Remove[Step]View was called while the parent view was being opened or closed`
4712	`Remove[Step]View was called with a template instead of a view frame`
4713	`Remove[Step]View was called on a view which was being opened or closed`
4714	`Remove[Step]View was called with a read-only stepChildren array (i.e., the view wasn't previously added with AddView)`
4715	`Close() was sent to a view which was opening or closing`

Table 2-7 View warning messages

Error number	Message
4716	`Toggle() was sent to a view which was opening or closing`
4717	`Toggle() was sent to a view whose parent was being opened or closed`
4718	`Show() was sent to a view which was opening or closing`
4719	`Hide() was sent to a view which was opening or closing`
4720	`RedoChildren() was sent to a view which was opening or closing`
4721	`SyncChildren() was sent to a view which was opening or closing`
4722	`SetKeyView() was sent to a view that wasn't a clParagraphView`

NewtApp Reference

This chapter describes the NewtApp framework data types and prototypes (protos). The protos are divided into the following categories:

- general application protos
- slot view protos
- labelled input-line protos

Required Code

This section describes the required `InstallScript` and `RemoveScript` functions.

Required InstallScript and RemoveScript Functions

A NewtApp application has required `InstallScript` and `RemoveScript` functions that you must include in your application build so it can register properly for various system services. You may copy these functions directly from the following code:

```
InstallScript := func(partFrame)
begin
   partFrame.removeFrame := (partFrame.theForm):
                  NewtInstallScript(partFrame.theForm);
end;

RemoveScript := func(partFrame)
begin
   (partFrame.removeFrame):NewtRemoveScript(removeFrame);
end;
```

General Application Protos

Included in this section are

- data storage proto `newtSoup`
- base view proto `newtApplication`
- base view control protos
- layout protos
- entry view protos

newtSoup

This is the abstract proto (in other words, it has no visible component) that contains soup-handling routines. Soup definitions in a NewtApp application must be based on the `newtSoup` proto, and are set up in the `newtApplication.allSoups` slot (page 3-10).

Slot descriptions

`soupName`
: Required. This should be a string that is unique to your application. If the application has only one soup, you can use a string version of your application symbol, for example, `"MyApp:SIG".`

  For an application that uses more than one soup, you can add a prefix to a string version of the application symbol, so the soup name becomes something like `"00:MyApp:SIG"`.

`soupIndices`
: An array of frames in which you define the indices for your soup. An index can be based on a single slot in the entry, or multiple slots in the entry. See the “Data Structures” (page 9-1) for more information about how to define a valid index. Here is an example:

```
soupIndices:
   [
               {structure: 'slot,
               path: 'title,
               type: 'string},

               {structure: 'slot,
               path: 'timeStamp,
               type: 'int},

               { structure: 'multislot,
               path: ['label1, 'label2],
               type: ['string, 'int] }
               ]
```

`soupQuery`
: Required. A soup query. Currently you cannot define a `tags` slot or a `validtest` method in the soup query. The soup query can include everything else; that is,

`BeginKey`, `EndKey`, `indexValidTest`, words, and text. Here are a few examples:

```
soupQuery: {type: 'index,
   indexPath:'title}

soupQuery: {type: 'index,
   indexPath:'timeStamp,
   BeginKey: time1, EndKey: time2}

soupQuery: {words: ["Newton", "NewtApp"]}
```

`soupDescr` Optional. A string describing the soup.

`defaultDataType`
Optional. (This slot pertains to applications that use stationery.) A unique symbol naming a data type for your soup entries. You may reuse your application signature as a value for this slot. An example is `'|BasicCard:sig|`. If an entry adopted from stationery does not already have a type defined (in its `class` slot) it is assigned this value.

AddEntry

myNewtSoup`:AddEntry(`*entry*, *store*`)`

Adds the entry to the specified store. If no store is given the entry is added to the default store. The return value is unspecified.

entry The entry to add. The only valid entries are those returned by the various cursor and entry methods.

store The result of a call to `GetDefaultStore` or `GetStores`—naming the device on which to store data. A value of `nil` causes the entry to be added to the default store.

AdoptEntry

myNewtSoup:`AdoptEntry(`*entry*, *type*`)`

Returns a soup entry frame with the values in the entry frame. This new entry consists of the frame specified in the `CreateBlankEntry` method, which you define in the `newtApplication.allSoups` slot, and—if your application has a dataDef—an entry defined in either a `FillNewEntry` or `MakeNewEntry` method. Note that if `FillNewEntry` exists, `MakeNewEntry` is not called.

entry	Required. If `nil`, a blank entry is created. The new entry is based on this entry.
type	Optional. Defaults to `nil`. If the value is `true`, the method looks for the value of the `class` slot of this entry. See Chapter 5, "Stationery," in *Newton Programmer's Guide*, for more information on the class slot.

The `class` slot and other slots of the dataDef entry are preserved as the entry is added to the application soup. If an *entry* is provided with a `class` slot, the type is automatically set to the same value as the `class` slot. If the value of the *type* parameter is `nil` and there is no `class` slot, the value of the `defaultDataType` slot, which is set in the `newtSoup` definition, is used to set the `type` and `class` slots for the entry.

CreateBlankEntry

myNewtSoup:`CreateBlankEntry()`

Returns a blank entry. Override this method to create the necessary structure of your soup. You may or may not want to put a `class` slot in your soup entry. However, note that any routable item must have one. (For more information about how the `class` slot is used, see Chapter 21, "Routing Interface," in *Newton Programmer's Guide*.)

DeleteEntry

myNewtSoup:`DeleteEntry(`*entry*`)`

Removes an entry from its soup. The entry frame is converted to a plain frame (which is unmarked as belonging to a soup).

entry	The entry to remove from the soup.

DuplicateEntry

myNewtSoup:`DuplicateEntry(`*entry*, *store*`)`

Clones and returns the specified entry. The duplicate entry is stored on the specified storage device.

entry	The entry to be duplicated.
store	The result of a call to `GetDefaultStore` or `GetStores`—naming the device on which to store data. A value of `nil` causes the entry to be added to the default store.

DoneWithSoup

myNewtSoup:`DoneWithSoup(`*appSymbol*`)`

Unregisters both the soup changes and the union soup to which the `newtSoup` you sent this message belongs.

appSymbol	A constant value specifying a unique alphanumeric symbol by which the application identifies itself to the system. An example of a suitable value is `'|Sample newtApp:DTS|`.

FillNewSoup

myNewtSoup:`FillNewSoup()`

Called by `MakeSoup` to add soup values to a new soup. The return value is unspecified. You should define this method with soup values appropriate to your application. A typical use of this method is to create “starter” entries for a new soup. If this is the desired behavior, you must create the entries and add them to the soup.

GetAlias

myNewtSoup`:GetAlias(`*entry*`)`

Returns an entry alias. This alias represents the specified soup entry—for fast access later—without holding on to the actual entry. The entry alias can be used later as input to the `GotoAlias` function to retrieve the soup entry. See "Entries" beginning on page 11-17 in *Newton Programmer's Guide* for more information.

entry The soup entry to which this method creates a an alias.

GetCursor

myNewtSoup`:GetCursor()`

Returns the cursor set up for the soup named within the `allSoups` slot of the `newtApplication` proto.

GetCursorPosition

myNewtSoup`:GetCursorPosition()`

Returns an alias to the cursor entry.

GotoAlias

myNewtSoup`:GotoAlias(`*alias*`)`

Returns the soup entry referenced by the specified alias. Returns `nil` if the entry cannot be retrieved. When this error occurs, typically it is because the original store, the original soup, or the original entry cannot be found.

alias The entry alias for which this method retrieves the corresponding soup entry.

MakeSoup

myNewtSoup:MakeSoup(*appSymbol*)

Used by the newtApplication proto to return and register a new soup. It assumes the soup is a standard union soup. If the soup is a new soup, it's filled with values by a call to FillNewSoup. Override this method to implement different behavior.

appSymbol A constant value specifying a unique alphanumeric symbol by which the application identifies itself to the system. An example of a suitable value is '|Sample newtApp:PIEDTS|.

Query

myNewtSoup:Query(*querySpec*)

Message you send to a newtSoup to perform a query on the soup. It returns a cursor that references a set of soup entries.

The *querySpec* frame may include the slots structure, path, type, and tagSpec. For more information on queries, see "Queries" (page 11-10) in *Newton Programmer's Guide*.

SetupCursor

myNewtSoup:SetupCursor()

Creates or resets the cursor as specified by the queryspec in the soupQuery slot.

newtApplication

The application base view template for all NewtApp applications. In an application, this proto contains the application-wide elements like the folder tab bar and status bar. It also contains references to all the layout protos and sets up the application soup.

Handlers for application-wide events like scrolling and filing are defined in this proto. It also dispatches the information to the appropriate parts of the application.

You must define the slots marked as required. Many of these contain strings that describe objects for menus or are used in alerts and notification slips.

Slot descriptions

`appSymbol` Required. A constant value that specifies a unique alphanumeric symbol by which the application identifies itself to the system. An example of a suitable value is `'|IOU:PIEDTS|`.

If you use NTK as your development environment, the application symbol is constructed for you from values you set in the Output Settings dialog box for that application.

`title` Required. A string that names your application. It is used by the system. An example is `"Roll Starter"`.

`appObject` Required. An array of two strings, in both the singular and plural, describing the data objects in the application soup. These strings are used by the system in the filing and action menus and for setting up soups. An example is `["Ox", "Oxen"]`.

`appAll` Required. A string used in the folder tab picker (pop-up menu) to provide the `All` *items* option. For example, the value of the `appAll` slot in the built-in Notes application is `"All Notes"`.

`aboutInfo` Optional. Defines information about your application that appears when the user chooses About from the `newtInfoButton` (page 3-23). To use, create a slot in your application's base a called `aboutInfo` and place a frame in this slot with the following slots:

```
{
tagLine: "", // A tagline for your application
version: "", // The version number for the application
copyright: "", //Copyright information
```

```
trademarks: "", // Trademark information
}
```

`aboutView` Optional. Defines information about your application that appears when the user chooses About from the `newtInfoButton` (page 3-23). To use, create a slot in your application's base view called `aboutView`. Use the `GetLayout` function to place a template of your view in this slot. A view is then created from the specified template when the user taps About in the `newtInfoButton`.

`allSoups` Required. Define the soup(s) for your application in this frame. Your soup definition should consist of a frame based on the `newtSoup` proto (page 3-2) containing the slots `soupName`, `soupIndices`, and `soupQuery`. An optional `soupFilter` slot supports filing.

Following is a sample `allSoups` frame:

```
allSoups: {
            mySoup: {
               _proto: newtSoup,
               soupName: "MySoup:SIG",
               soupIndices: [],
               soupQuery: {type: 'index},
               CreateBlankEntry: func()
                  {  slot1: 123,
                     slot2: 456, }
               }
            }
```

Note that each layout is tied to one of these soups by using the soup name(s) in its `masterSoupSlot`.

`allLayouts` Required. A frame that contains references to the application's layouts. Two slots are required: `default` and `overview`. These slots must contain paths to existing layout files.

A suitable definition for the `allLayouts` frame follows:

```
allLayouts:
{default:GetLayout("DefaultLayoutFile"),
 overview:GetLayout("OverviewLayoutFile"),
 }
```

`scrollingEndBehavior`
: Optional. Defaults to `'beepAndWrap`. You may also set it to the values `'wrap`, `'stop`, `'beepNWrap`, or `'beepAndStop`.

The values select how scrolling is handled at the end of a view. `'wrap` causes scrolling to display from the last entry around to the first (or vice versa). `'stop` means that scrolling stops when the display reaches either end. `'beepAndStop` means the application will stop at the last entry and play a beep. `'beepNWrap` means to continue scrolling past the last entry, and play a scrolling sound and "wrap" to the first entry.

Each scrolling choice comes in a quiet and noisy form. If you choose the noisy version, it makes an extra scrolling sound.

`scrollingUpBehavior`
: Optional. Defaults to `'bottom`. You can set it to either `'top` or `'bottom`.

These settings select how roll-style entries are displayed when scrolling up. For instance, say you scroll backwards to a note that is two screens high; you'll see either the bottom or top screenful of the note. A roll-style application would use `'bottom`, but an application that uses information slips would use `'top`.

`statusBarSlot`
: Optional. A symbol that is the declared name of the status bar. It is used by the layout to govern the appearance/disappearance of buttons on the status bar. For this to work, the layouts must also have

`menuLeftButtons` and `menuRightButtons` slots. See `newtStatusBarNoClose` (page 3-29) and `newtLayout` (page 3-32), for more information.

The following slots are used to create and save preferences.

Slot descriptions

`prefsView` Optional. Contains a template of your preferences slip and is opened when the user selects Prefs in the newtApp.

`theApp` Optional. Adds a reference to the application's base view, the default newtAboutBox.

The following slots are important if you are incorporating stationery into your application:

Slot descriptions

`allDataDefs` Required if your application supports stationery. A frame that contains the symbol(s) identifying the dataDef(s) and a reference to the file(s) containing the data definition(s) for this application. Following is the `allDataDefs` slot of the Basic Card example:

```
{|basicCard:SIG|: GetLayout("iouDataDef")}
```

The system automatically registers all dataDefs in this frame when the application installs. For more information about dataDefs, see Chapter 5, "Stationery," in *Newton Programmer's Guide*.

`allViewDefs` Required if your application supports stationery. This frame contains the unique `dataDef` symbol(s), which are registered in the base view `allDataDefs` slot, and the references to the layout files for the viewDef(s), which can display their data. The following example contains two viewDef template references for the default and notes layout files:

```
{|IOU:SIG|:
      {default:
      GetLayout("iouDefaultViewDef"),
      notes:
      GetLayout("iouNotesViewDef"),}}
```

The system uses this slot to register the view formats for each given `dataDef`.

`superSymbol` Required for stationary. A unique symbol that identifies the superset of data defs used for this application. It is recommended that you set it to the value of the application symbol if the application has only one dataDefs. For instance, assuming one data type for the application, both your application symbol and `superSymbol` could be set to `'|IOU:SIG|`.

Note that any would-be stationery extensions to this application must also have a `superSymbol` that matches this value.

Following are the routing, filing, and find slots:

Slot descriptions

`doCardRouting` Optional. Defaults to `true.` This enables the filing interface to allow moves to and from cards. Set to `'onlyCardRouting` for filing to cards without folders.

`dateFindSlot` Optional. Enables your application to be used in a `dateFind` query. Set it to a path expression that evaluates to a slot in your soup entry that contains a date. This slot must be indexed in the `newtApplication.allSoups` slot. An appropriate value is `'timeStamp`.

`routeScripts` Optional. Contains default route scripts for Delete and Duplicate. If you do not want these options to show in the Action menu, you must override the default `routeScripts` array.

The following slots are included for your information only and should not be set by you. They are maintained automatically by the NewtApp framework code.

Slot descriptions

`labelsFilter`	Created dynamically as needed by the system, it is used to store filing settings by the `newtApplication` proto.
`newtAppBase`	This identifies the base view of your application. The system uses the value of `newtAppBase` to identify, for instance, which view should be closed when a close box is tapped.
`retargetChain`	This contains a dynamically built array of views contained by (or chain out from) a particular view. When the base container view is changed and redrawn, these views are also updated.
`targetView`	This is the view in which data from the target entry is displayed.
`target`	This usually points to the entry being displayed and is used by system services such as filing.
`layout`	This is set to the current layout.

GetAppPreferences

myNewtApplication`:GetAppPreferences()`

Returns a frame of preferences for the application. Use this method to add a preference slip to your application.

NewtApplication Stationary Methods

The following methods support adding stationary to your application.

ShowLayout

myNewtApplication:`ShowLayout(`*layout*`)`

Used to display a particular layout, at the appropriate time, in your application. This method sets the current layout to the layout you specify. A parameter value of `nil` sets the value of the current layout to the value of the previous layout. You can use it to switch the display from one layout to the other layout (for example, from the main view to the overview.)

layout A symbol referring to a specific layout, as listed in the `allLayouts` slot.

AddEntryFromStationery

myNewtApplication:`AddEntryFromStationery(`*stationerySymbol*`)`

Called by the stationery button (`newtNewStationeryButton` proto) to create a blank entry and initialize its `class` slot with the value passed in as *stationerySymbol*.

stationerySymbol A symbol referring to the value of the stationery's `symbol` slot. It is used to set a `class` slot for the new blank entry. An example of an appropriate value from the built-in Notes soup is `'paperroll`.

AdoptEntryFromStationery

myNewtApplication:`AdoptEntryFromStationery (`*adoptee*, *stationerySymbol*, *store*`)`

Like `AddEntryFromStationery`, but also copies all slots from the existing entry into the new entry. There is no protection here, so be careful it does not overwrite existing slots.

adoptee The data being adopted. This is usually a soup entry.

stationerySymbol A symbol that is the same as the stationery's `dataDef` symbol. It creates a new entry from an existing entry. The existing entry is created on the appropriate store, and then is used to set a `class` slot according to the stationery symbol. The new entry is built using the `MakeNewEntry` and `FillNewEntry` methods in the stationery dataDef. After the entry is built, all slots from the existing entry are copied to the new entry and the new entry is added to the soup.

store The store on which to keep the information. If `nil` is specified, data is stored on the internal storage device.

AdoptSoupEntryFromStationery

myNewtApplication`:AdoptSoupEntryFromStationery(`*adoptee*, *stationerySymbol*, *store*, *soup*`)`

Copies all slots from the entry to be adopted into the new entry and sets the `class` slot of that entry to the value of the *stationerySymbol*. You may specify to which soup and store the entry should be added.

adoptee The entry being adopted.

stationerySymbol A symbol referring to the value of the stationery's `symbol` slot. It is used to set a `class` slot for the new blank entry. An example of an appropriate value from the built-in Notes soup is `'paperroll`.

store The store on which to keep the information. If `nil` is specified, data is stored on the internal storage device.

soup The symbol for one of the soups in the `allSoups` slot. Use `nil` to indicate the current soup.

NewtApplication Filing Methods

The following methods, defined in the `newtApplication` proto, are used to support filing in your application.

FolderChanged

myNewtApplication:`FolderChanged(`*soupName*, *oldFolder*, *newFolder*`)`

Changes the folder tab label to the new folder name if it is different from the old folder name, and saves the new folder information for the soup.

soupName	Required. The name of the soup.
oldFolder	Required. The folder where the document was previously found.
newFolder	Optional. A missing *newFolder* parameter means the folder was deleted.

FilterChanged

myNewtApplication:`FilterChanged()`

Saves the old folder name for each soup in the `allSoups` slot, updates it to the new folder name, and sets the soup cursor to refer to the new folder. Finally, it sends the `FilterChanged` message to the `newtLayout` proto so it targets the appropriate view for the new folder.

ChainIn

myNewtApplication:`ChainIn(`*chainSymbol*`)`

Adds a view to an array of views to be notified when the data in a layout is changed by sending the `Retarget` message. This is automatically done for you in the `newtFilingButton` proto and the `newtAZTabs` proto.

Any time the contents of a view are changed, this method updates the affected view(s) and change the data target entry.

chainSymbol	A symbol naming a slot that holds an array of views that need to be notified when a `Retarget` message is sent. The symbol should be `'retargetChain` for the `retargetChain` slot provided in the `newtApplication` proto.

ChainOut

myNewtApplication:`ChainOut(`*chainSymbol*`)`

Removes a view from an array of views which are to be notified when the data in a layout is changed by sending the `Retarget` message. This is done automatically for you in the `newtFilingButton` proto and the `newtAZTabs` proto.

Any time the contents of a view are changed, this method updates the affected view(s) and change the data target entry.

chainSymbol — A symbol naming a slot that holds an array of views that need to be notified when a `Retarget` message is sent. The symbol should be `'retargetChain` for the `retargetChain` slot provided in the `newtApplication` proto.

GetTarget

myNewtApplication:`GetTarget()`

Returns the current soup entry, which is also known as the target soup entry. The target in the application level is undefined.

GetTargetView

myNewtApplication:`GetTargetView()`

Returns the view in which the target soup entry is displayed. The target view in the base application level is undefined.

newtApplication Find Methods

The following methods, defined in the `newtApplication` proto, are used to add Find support to your application. You do not call any of these methods. For more about the Find system services, see Chapter 13, "Find Reference," in the *Newton Programmer's Guide*.

DateFind

myNewtApplication:DateFind(*date*, *findType*, *results*, *scope*, *findContext*)

The default DateFind method as provided in the NewtApplication proto. You must supply a dateFindSlot to your newtApplication proto for your application to utilize this DateFind method.

This method searches for all items that occur on, before, or after a date, depending on which choice the user makes from the Find dialog box.

This DateFind method displays a status view that reports where it is currently searching for the date value. It looks for the specified date in all the soups specified in the allSoups slot of your application and builds an array that contains the results. You should use the ShowFoundItems method to report the results.

date	Specifies the date selected by the user. The date is represented as an integer that is the number of minutes passed since midnight, January 1, 1904.
findType	Either the symbol 'dateBefore or 'dateAfter. Specifies whether the user chose to find items before or after the date specified by the value of the *date* parameter.
results	This DateFind method appends the slot myResult to the *results* array passed to the DateFind method by the system. The exact content of the myResult slot depends on the kind of finder proto used to create the frame returned by your search method. If you used the soupFinder proto, the frame contains a cursor that iterates over a list of entries returned by your search method's query on the application data soup. If you used the ROM_CompatibleFinder proto, the frame contains an array of found items. If a global find is in progress, the *results* array may contain slots created by other applications' search methods.
scope	Either 'localFind or 'globalFind. Indicates whether the search is local or global, allowing you to handle these two cases differently if you prefer.

findContext A frame to which the message `SetStatus` is sent. The `SetStatus` function accepts as its sole argument a string to display to the user while the search is in progress.

Find

myNewtApplication:`Find(`*text*, *results*, *scope*, *findContext*`)`

Searches all the soups in the `allSoups` frame for the *text* specified by the user. The return value of this method is ignored; the results of the search are returned in the *results* parameter.

text Contains the user-specified string for which Find is to search.

results This `Find` method appends the slot `myResult` to the *results* array passed to the `Find` method by the system. The exact content of the `myResult` slot depends on the kind of finder proto used to create the frame returned by your search method. If you used the `soupFinder` proto, the frame contains a cursor that iterates over a list of entries returned by your search method's query on the application data soup. If you used the `ROM_CompatibleFinder` proto, the frame contains an array of found items. If a global find is in progress, the *results* array may contain slots created by other applications' search methods.

scope Either `'localFind` or `'globalFind`. Indicates whether the search is local or global, allowing you to handle these two cases differently if you prefer.

findContext A frame to which the message `SetStatus` is sent. The `SetStatus` function accepts as its sole argument a string to display to the user while the search is in progress.

ShowFoundItem

myNewtApplication:`ShowFoundItem(`*entry*, *finder*`)`

Switches folders as necessary to show the found items as they are chosen by the user from the dialog box.

entry	The entry in which the item is found.
finder	A NewtApp-compatible finder constructed by the `newtApplication` proto.

newtApplication Delete and Duplicate Methods

The following methods, defined in the `newtApplication` proto, can be used to delete and duplicate data items.

NewtDeleteScript

myNewtApplication:`NewtDeleteScript(`*what*, *view*`)`

Deletes the specified item(s) and removes it from the specified view. This method displays alerts, in case someone tries to use delete when nothing is selected or tries to delete items in the Overview. This method also saves the item and the view for a possible undo action.

what	A cursor or other reference to the item(s) to delete.
view	A symbol referring to the view in which the item appears.

NewtDuplicateScript

myNewtApplication:`NewtDuplicateScript(`*what*, *view*`)`

Duplicates the specified item(s) and adds the duplicate to the specified view. This method also displays an alert which appears if someone tries to duplicate when nothing is selected. This method saves the item and the view for a possible undo action.

what	A cursor or other reference to the item(s) to be duplicated.

view A symbol referring to the view in which the item appears.

NewtApplication Status Methods

The following methods, defined in the `newtApplication` proto, can be used to obtain information about and save the state of your application.

GetAppState

myNewtApplication`:GetAppState()`

Gets the application preferences and uses them to set the values of the labels filter, the current and previous layouts, and the recognition settings. It then returns a copy of the application preferences.

Your application may override `GetAppState`, `SaveAppState`, and `GetDefaultState` to add your own application preferences.

GetDefaultState

myNewtApplication`:GetDefaultState()`

This method sets the default values for the application preferences, including values for the labels filter, the position of the current layout, the current and previous layouts, and the recognition settings.

Your application may override `GetAppState`, `SaveAppState`, and `GetDefaultState` to add your own application preferences.

SaveAppState

myNewtApplication`:SaveAppState()`

Saves application status. The following is saved:

- folder positions for each entry in each soup in the `allSoups` slot
- filters used to determine filing location
- view positions, including the current and previous layouts

Your application may override `GetAppState`, `SaveAppState`, and `GetDefaultState` to add your own application preferences.

newtInfoButton

This proto provides the standard "i" information button, which always appears to the far left of the status bar. It is based on `protoInfoButton`, discussed in Chapter 6, "Controls Reference."

Unlike the `protoInfoButton`, the `newtInfoButton` proto provides the default methods `DoInfoAbout`, `DoInfoHelp`, and `DoInfoPrefs`, which are invoked when the user taps About, Help, or Prefs in the picker, as shown in Figure 3-1.

Figure 3-1 The Information button and picker

The following methods provide default handling for items in the picker menu of the `newtInfoButton`.

DoInfoAbout

myInfoButton`:DoInfoAbout()`

Closed and set to `nil` if an About view has been created. If no About view is open, one is created.

DoInfoHelp

myInfoButton`:DoInfoHelp()`

Closed and set to `nil` if an on-line Help book has been created. If no Help book is open, this method looks for an index to one in a `viewHelpTopic` slot in the base view. If one exists, the Help manual is opened to the index location; otherwise, it is just opened.

DoInfoPrefs

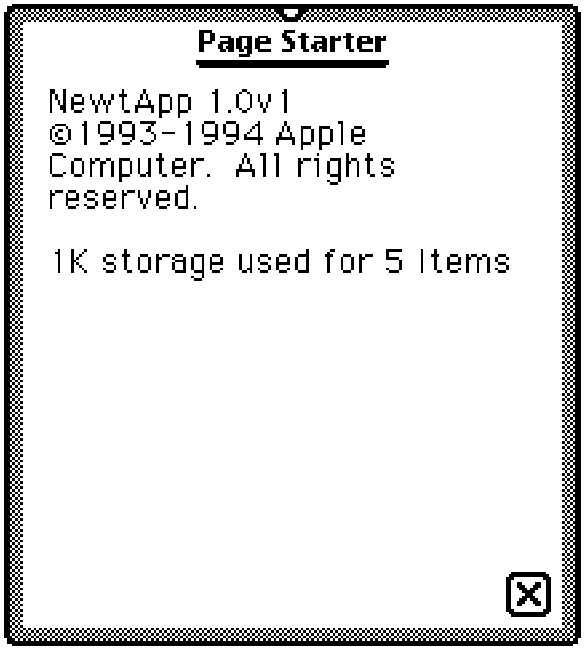
myInfoButton`:DoInfoPrefs()`

Closed and set to `nil` if a Preferences view has been created. If no Preferences view is open, one is created.

newtAboutView

This proto is the view in which information about the application is stored. The About view is displayed when the user chooses About from the Info ("i") button picker, which sends the `DoInfoAbout` message. It appears as shown in Figure 3-2.

Figure 3-2 The NewtApp About view



newtPrefsView

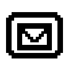
This proto is the view in which information about the application is stored. The Preferences view is displayed when the user chooses Prefs from the Info ("i") button picker and the method `DoInfoPrefs` is sent. It appears as shown in Figure 3-3.

Figure 3-3 A NewtApp Preferences view

newtActionButton

This proto provides the standard action button. If you have a card-style application and want routing, place this in the `menuRightButtons` slot of `newtStatusBar` (page 3-30) and the framework will place it correctly on the status bar. The action button belongs next to the close box (to the left). It appears as shown in Figure 3-4.

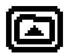
Figure 3-4 The Action button



newtFilingButton

This proto provides the standard filing button, with added functionality of working with the NewtApp framework. If you have a card-style application and want filing, place this in the `menuRightButtons` slot of `newtStatusBar` (page 3-30) and the framework will place it correctly on the status bar. The filing button belongs to the left of the action box. It appears as shown in Figure 3-5.

Figure 3-5 The Filing button



newtAZTabs

This proto is used to include alphabetical tabs, arranged horizontally, in a view; it is based on the `protoAZTabs` but adds useful functionality to that base. (See `protoAZTabs` in Chapter 6, "Controls Reference.") The `newtAZTabs` view appears as shown in Figure 3-6.

Figure 3-6 NewtApp A-Z tabs

When a view is changed and a new view is set up, as happens when someone taps an alphabet tab, each view is automatically added to a `retargetChain` array. When a view needs to update and redraw itself, the rest of the views in the chain of views contained by it are notified, and a `Retarget` message is sent to the entire chain.

Note that `newtAZTabs` works by using the index you have set up in an `indexPath` slot of the `soupQuery` for your soup. (These are defined in the `newtApplication.allSoups` base view slot.)

This proto defines its own versions of `RetargetNotify` and `PickLetterScript`, which you can override to add functionality appropriate to your application data. If you do, however, remember to call the inherited method.

PickLetterScript

myTabs`:PickLetterScript(`*letter*`)`

Called when the user taps a tab. The *letter* on the tab is matched to the value set up in the `indexPath` slot of the `soupQuery` frame (in the `newtApplication.allSoups` slot), and the entry and view are retargeted.

letter The letter that was tapped.

newtFolderTab

This is the plain folder tab. If you want filing to operate correctly in your application, it must use either this proto or the `newtClockFolderTab` proto. The `newtFolderTab` view is shown in Figure 3-7.

Figure 3-7 The plain folder tab

newtClockFolderTab

This folder tab incorporates a date and time indicator. It is automatically updated if the current folder is deleted. When the user taps the folder tab, a picker containing the list of folders available to your application displays. If you want filing to operate correctly in your application, it must use either the `newtFolderTab` proto or the `newtClockFolderTab` proto, shown in Figure 3-8.

Figure 3-8 The digital clock and folder tab

newtStatusBarNoClose

This proto is the basic component of the `newtStatusBar`: the bar alone, with no buttons or close box.

This proto implements the `menuLeftButtons` and `menuRightButtons` slots, which are placeholders for buttons you add. The slots `menuLeftButtons` and `menuRightButtons` are arrays of buttons to be displayed on the status bar. They are arranged at display time as stepchildren of the menu bar.

When there is no `statusBarSlot` (page 3-11) set in the `newtApplication` base view, the status bar figures the correct size of the buttons in the `menuLeftButtons` and `menuRightButtons` arrays and places them correctly. It is recommended that you use these slots to ensure the correct justification of your status bar buttons with future enhancements.

If the `statusBarSlot` in the base view has been set, the appearance and disappearance of the buttons on the status bar is governed by the values set for the `menuLeftButtons` and `menuRightButtons` slots, at the layout level of the application. See "newtLayout," beginning on page 3-32.

The buttons in the `menuLeftButtons` array are laid out from left to right, starting with the Info button. The buttons in the `menuRightButtons` array are laid out from right to left, starting with the close box.

Slot descriptions

`menuLeftButtons`

An array of standard text buttons. The elements in the array are laid out from left to right, with the first element at the far left. An appropriate value is shown in the following code:

```
menuLeftButtons:
   [newtInfoButton,
   newtNewStationeryButton,
   newtShowStationeryButton]
```

`menuRightButtons`

An array of standard text buttons. The elements in the array are laid out from right to left, with the first element at the far right. An appropriate value is shown in the following code:

```
menuRightButtons:
   [newtActionButton,
   newtFilingButton,]
```

newtStatusBar

This proto is based on the `newtStatusBarNoClose`. The only difference between the two is that this status bar includes a large close box at its far right side, as shown in Figure 3-9. As with the `newtStatusBarNoClose` proto, you may use the `menuLeftButtons` and `menuRightButtons` arrays.

Figure 3-9 A status bar view

Slot descriptions

`menuLeftButtons`

An array of standard text buttons. The elements in the array are laid out from left to right, with the first element at the far left. An appropriate value is shown in the following code:

```
menuLeftButtons:
   [newtInfoButton,
   newtNewStationeryButton,
   newtShowStationeryButton]
```

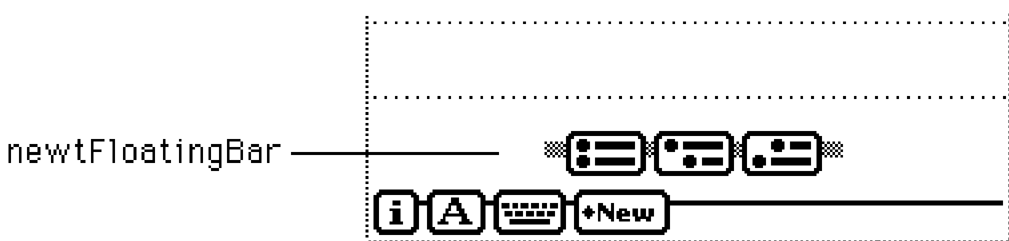
`menuRightButtons`
An array of standard text buttons. The elements in the array are laid out from right to left, with the first element at the far right. An appropriate value is shown in the following code:

```
menuRightButtons:
   [newtActionButton,
   newtFilingButton,]
```

newtFloatingBar

This proto is like a standard `newtStatusBar`, but it floats at the bottom of a view. It was originally designed for the Notes application where individual view types such as the Outline view have their own menu buttons that are not necessary for the main application view. Like the `newtStatusBar` proto, it implements a `menuButtons` slot, in which you may enumerate the buttons to appear on the floating bar. A floating bar view is shown in Figure 3-10.

Figure 3-10 A floating bar view



Slot description

`menuButtons` An array of button protos. Buttons are laid out, an equal distance apart, left to right in array order on the status bar.

newtLayout

This proto must have at least one `newtEntryView` proto as a child view. (It may also contain other protos.) For layouts to work correctly, you must set the `masterSoupSlot` to the soup from the `newtApplication.allSoups` slot to be used for this layout. In addition, you can direct your application to force a new entry to be created (or not) when a user opens an empty folder, by setting a layout's `forceNewEntry` slot.

The `menuLeftButtons` and `menuRightButtons` slots allow you to control which buttons appear on the status bar from the layout layer of the application. (The `statusBarSlot` of the `newtApplication` base view must also be set.)

The following slots originate in the `newLayout` proto and are inherited by the other layout protos:

Slot descriptions

`name` — Optional. An example is `"All Info".`

`masterSoupSlot` — Required. A symbol that refers to the soup in the `newtApplication.allSoups` frame that is the main soup of your application. It sets up the cursor and soup query for your application. An appropriate value would be `'mySoup`.

`forceNewEntry` — Optional. Defaults to `true`. Creates a blank entry for this layout when the application is switched to a folder with no entries.

If `forceNewEntry` is set to `nil`, no blank entries are created. Instead, the application displays the string, "There are no *items* in this folder," where *items* is replaced by the value of the `appAll` slot set in the `newtApplication` base view.

`menuRightButtons`

Optional. If the `statusBarSlot` in the base view is set, this is used to replace the `menuRightButtons` on the status bar in the main layout.

`menuLeftButtons`
: Optional. If the `statusBarSlot` in the base view is set, this is used to replace the `menuLeftButtons` on the status bar in the main layout.

The following slots are included for your information. They are maintained automatically, so you need not worry about setting them. The `dataCursor` slot is the main cursor to your application soup.

Slot descriptions

`dataSoup`	Set to the soup that contains the data this layout displays.
`dataCursor`	The main cursor to the data soup; it points to the topmost visible entry.

The following methods are defined in the `newtLayout` proto.

FlushData

myLayout`:FlushData()`

Flushes all entries in the child views held by the layout view.

NewTarget

myLayout`:NewTarget()`

Resets the view origin and redoes the screen.

Retarget

myLayout`:Retarget(`*setViews*`)`

Sets the cursor (`dataCursor`) to the new or changed entry and redraws the screen after the cursor is changed, if the *setViews* parameter is `true`. Note that you should not use this method with a `newtOverLayout` or `newtRollOverLayout` proto.

setViews	If set to `true`, the child views are redrawn.

DoRetarget

myLayout`:DoRetarget`

If received by the entry layer, it performs a `ReTarget` on itself. If received by the layout layer, it performs a `ReTarget`, with a non-nil value, on itself.

ScrollCursor

myLayout`:ScrollCursor(`*delta*`)`

Moves the cursor *delta* entries and resets it.

delta	An integer which can be greater than 0 or less than or equal to 0, depending on the direction for the scroll and the amount to scroll. If *delta* is not equal to 0 (and the cursor is valid), the cursor is moved that number of places. A value less than or equal to 0 causes the cursor to reset to the end of the entries (for a scrolling end behavior of `'wrap` or `'beepAndWrap`) or to move to the next entry (for a scrolling end behavior of `'stop` or `'beepAndStop`). A value greater than 0 causes the cursor to reset (for a scrolling end behavior of `'wrap` or `'beepAndWrap`) or to move to the previous entry (for a scrolling end behavior of `'stop` or `'beepAndStop`).

SetUpCursor

myLayout`:SetUpCursor()`

Sets the cursor to an entry in the master soup and returns the entry to which the cursor is set. If there are no entries in the master soup and `forceNewEntry` is `true`, this method creates a blank entry (by calling `AddBlankEntry`) and sets the cursor to it.

Scroller

myLayout:`Scroller(`*numAndDirection*`)`

Traverses the number of entries specified by the parameter. In addition, depending on whether the parameter is less than or greater than 0, the scroller scrolls either up or down.

numAndDirection	Either +*n* or –*n*, where *n* is the number of entries to traverse. A value less than 0 is a scroll up and a value greater than 0 is a scroll down.

IMPORTANT

This cannot be used in a `newtOverLayout` or `newtRollOverLayout`. ▲

ShowFoundItem

myLayout:`ShowFoundItem(`*entry*, *finder*`)`

Uses the cursor already set up in the `dataCursor` slot to go to the slot in the specified entry and conditionally sends the `ShowFoundItem` message to any child views. You may choose to override the method to customize it to the specific data.

entry	A valid soup entry.
finder	A NewtApp-compatible finder.

ViewScrollDownScript

myLayout:`ViewScrollDownScript()`

Produces a visual effect and calls the `scroller` method with a value of 1.

ViewScrollUpScript

myLayout:`ViewScrollUpScript()`

Produces a visual effect and calls the `scroller` method with a value of -1.

newtRollLayout

An example of this prototype can be seen in the built-in Notes application, which it was designed to support. This proto is meant to work with stationery-based children and does not work with other protos without a lot of effort on your part.

A `newtRollLayout` calculates at run time how many children it has, depending on the number and size of the entries in the soup. It uses the layout file—which must contain a `newtRollEntryView` proto you provided as the value of the `protoChild` slot—as the default child view to use when it dynamically builds itself.

IMPORTANT

Do not place the entry view of a roll-style application inside a layout view; instead, it must be in a layout file (in NTK) which is declared in an expression in the `protoChild` slot, as shown in the following example:

```
MyRollLayout.protoChild :=
GetLayout("DefaultEntryView")
```
▲

Slot description

`protoChild` Required. Reference to the layout file containing the view to use to lay out the child views. The child view must be a `newtRollEntryView`. This is the most important `newtRollLayout` slot. Do not create the entry view within a layout view in a page-style application. Instead, create it in a separate layout file.

An appropriate value for the `protoChild` slot of a `newtRollLayout` is `GetLayout("DefaultEntryView")`.

There are no new methods specifically for the roll layout proto. However, it does have its own version of the `Scroller` method, modified so it works with the long pages of the `newtRollLayout`. See the `newtLayout Scroller` method (page 3-35) for more information.

newtPageLayout

This layout allows one entry to be visible at a time; otherwise, it acts the same as the roll layout. The entry shown can be longer than one screenful.

A `newtPageLayout`, like the `newtRollLayout` proto, calculates at run time how large it is, depending on the size of its child views. It uses the layout file—which must contain a `newtPageEntryView` proto you provided as the value of the `protoChild` slot—as the default child view to use when it dynamically builds itself.

IMPORTANT

Do not place the entry view of a page-style application inside a layout view; instead, it must be in a layout file (in NTK) which is declared in an expression in the `protoChild` slot, as shown in the following example:

```
MyPageLayout.protoChild :=
GetLayout("DefaultEntryView")
```

▲

Slot description

`protoChild` Required. Reference to the layout file containing the view to use to lay out the child views. The child view must be a `newtRollEntryView`. This is the most important `newtRollLayout` slot. Do not create the entry view within a layout view in a roll-style application. Instead, create it in a separate layout file.

An appropriate value for the `protoChild` slot of a `newtPageLayout` is `GetLayout("DefaultEntryView")`.

newtOverLayout

This is the default overview. It is based on `protoOverview`. (See `protoOverview` (page 5-85) for more information.) It is singled out by the `newtApplication` proto so that overview events invoke it.

As with the `protoOverview`, the `newtOverLayout` proto doesn't have view children; instead, it builds up shapes containing the overview information and handles taps. These shapes are returned by the `Abstract` method.

Because of the way the `newtOverLayout` proto is implemented, you should make sure that if you override an inherited method, you include a call to that method by using the conditional message send (:?) operator.

Slot descriptions

`masterSoupSlot`	Required. A symbol that matches a value in the `allSoups` slot in the `newtApplication` base view.
`dataCursor`	Required. Do not set this; value is inherited from the parent layout proto.
`name`	Required. Set it to something meaningful, like "Overview."
`centerTarget`	Optional. Defaults to `nil`. When set to `true`, the current entry is centered in the overview list.
`forceNewEntry`	Optional. Defaults to `true`. Creates a blank entry for this layout when the application is switched to a folder with no entries. If `forceNewEntry` is set to `nil`, no blank entries are created. Instead, the application displays the string, "There are no *items* in this folder," where *items* is replaced by the value of the `appAll` slot set in the `newtApplication` base view.
`menuRightButtons`	Optional. If the `statusBarSlot` in the base view is set, this is used to replace the `menuRightButtons` in the `newtStatusBar` in the main layout.
`menuLeftButtons`	Optional. If the `statusBarSlot` in the base view is set, this is used to replace the `menuLeftButtons` in the `newtStatusBar` in the main layout.

`nothingCheckable`
: Optional. When `true`, the check boxes and vertical dotted line are suppressed.

Several methods are defined in this proto.

Abstract

myOverLayout`:Abstract(`*targetEntry*, *bbox*`)`

Returns a shape or shape list representing an item in the overview. It is passed two parameters; the first is the target soup entry and the second a bounds frame within which the returned shape should be placed. You should override this method to extract text from your soup format.

It extracts an icon for the entry (if one is provided) from the `icon` slot of a dataDef.

targetEntry Required. The soup entry frame to be displayed.

bbox Required. The bounding box defining the shape for the overview information. This includes a value for the left, right, top, and bottom.

An `Abstract` method example follows:

```
Abstract:
       func(item, bbox )
       begin
       // returns a shape for one line in the overview
       MakeText(item.name, bbox.left, bbox.top,
                   bbox.right, bbox.bottom);
       end;
```

GetTargetInfo

myOverLayout`:GetTargetInfo(`*targetType*`)`

Used by several system services (such as Filing, Find, and Routing) to get information about the currently selected item. You can override this method if necessary.

targetType A symbol identifying what special kind of information the view should return, besides the default frame. Currently, the only symbol defined is `'filing`. Any other value is ignored.

Slot descriptions

This method returns a frame that has the following slots:

`target` The value of the `target` slot in the view to which this message is sent.

`targetView` The value of the `targetView` slot in the view to which this message is sent. If *targetType* is `'filing`, this slot contains the value of the `targetApp` slot in the current view instead.

`targetStore` If the `target` slot is a soup entry, the store on which the entry resides is returned in this slot.

HitItem

myOverLayout`:HitItem(`*index*, *x*, *y*`)`

A method called when an item is tapped. The default method returns `true` if it handled the tap; that is, if it determined the tap was within the `selectIndent` margin and selected the item.

If you choose to override this method, you should check the *x*, *y* values; if you don't want to handle them, call `inherited:HitItem`. Also, be sure to exclude the indent margin from your test.

index The index to the item in the list (the first one being 0).

x The *x* coordinate of the tap, relative to the left edge of the item that was tapped.

y The *y* coordinate of the tap, relative to the top edge of the item that was tapped.

newtRollOverLayout

Same as the `newtOverLayout` proto, except that it must be used in a roll-style application. It is based on `newtOverLayout`. It is singled out by the `newtApplication` proto so overview events invoke it.

The `newtOverLayout` proto doesn't have view children; instead, it builds up a shape containing the overview information and handles taps. These shapes are returned by the `Abstract` method.

Because of the way the `newtRollOverLayout` proto is implemented, you should make sure that if you override an inherited method, you include a call to that method by using the conditional message send (:?) operator.

Slot descriptions

`masterSoupSlot` Required. A symbol that matches a value in the `allSoups` slot in the `newtApplication` base view.

`dataCursor` Required. You do not set this, it is inherited from the parent layout proto.

`name` Required. Set it to something easy to remember, like "Overview."

`forceNewEntry` Optional. Defaults to `true`. Creates a blank entry for this layout when the application is switched to a folder with no entries.

If `forceNewEntry` is set to `nil`, no blank entries are created. Instead, the application displays the string, "There are no *items* in this folder," where *items* is replaced by the value of the `appAll` slot set in the `newtApplication` base view.

`centerTarget` Optional. Defaults to `nil`. When set to `true`, the current entry is centered in the overview list.

`menuRightButtons`
Optional. If the `statusBarSlot` in the base view is set, this replaces the `menuRightButtons` in the `newtStatusBar` in the main layout.

`menuLeftButtons`
Optional. If the `statusBarSlot` in the base view is set,

this replaces the `menuLeftButtons` in the `newtStatusBar` in the main layout.

`nothingCheckable`
: Optional. When `true`, the check boxes and vertical dotted line are suppressed.

newtEntryView

The `newtEntryView` proto is the invisible container view for the protos that allow you to view and edit data. See "Slot View Protos" (page 3-49) for details. This proto is essential because it sets the `target` slot to refer to the soup entry that contains the data for the slot views to display.

There are no unusual slots to set, just the usual bounds and justify slots, and then only if you want to override the default settings.

The following slots are set automatically. Note that dataDefs and viewDefs are identified and used as target entries and target views in several `newtEntryView` slots.

IMPORTANT

Do not change the values of any of the following slots, or your application will not work correctly. ▲

Slot descriptions

`entryChanged` — When an entry is changed in a viewDef, this is set to `true` for flushing.

`entryDirtied` — If the targeted viewDef was changed once and a flush occurred, this is set to `true`. When the view is closed down, it checks this. If set, it does a broadcast soup change to other applications.

`target` — Set to the entry that is ready to display.

`viewJustify` — Optional. Defaults to parent full justify for horizontal and vertical `vjParentFullH + vjParentFullV`

`currentDataDef` — Set by the enclosed stationery view to the current dataDef. (See Chapter 5, "Stationery," in *Newton Programmer's Guide* for more information.) This is a

	convenient access point for items like the `newtEntryRollHeader,` so it can pull out the appropriate icon from the `newtInfoBox`.
`currentViewDef`	Set by the enclosed stationery view to the current viewDef.
`currentStatView`	Set by the enclosed stationery view to the current context of the viewDef. If the target entry has a dataDef displayed, this points to it. Internal methods need to know the context for the view that contains the dataDef so messages may be sent to it.

The following methods are defined for the `newtEntryView` proto and are inherited by all entry views that are based on it.

StartFlush

myEntryView`:StartFlush()`

Starts the timer that flushes out the entry after a few seconds of inactivity. Normally this is called automatically by a dataDef, but if you have some other reason for causing an entry to be flushed, call this directly. Calling this sets the `entryChanged` slot and begins the flush timer.

EndFlush

myEntryView`:EndFlush()`

Called when the flush timer fires. If you want an immediate flush, set `entryChanged` to `true` and call this method.

EntryCool

myEntryView`:EntryCool(`*report*`)`

Checks to see if the target entry is on read-only media.

report	If *report* is a non-`nil` value, the notice “This is on a write-protected card and cannot be changed” is displayed, if the target entry in on read-only media.

JamFromEntry

myEntryView:JamFromEntry(*otherEntry*)

Looks for a JamFromEntry method in each child of the entry view and sends the same message to its childviews if appropriate. It then retargets the view to display the changes. For more information, see the slot view's redefinition of JamFromEntry (page 3-50).

otherEntry A soup entry. This is intended to be an entry other than the one to which the entryView is already targeted.

Retarget

myEntryView:Retarget()

Changes the display for the viewDef(s) and dataDef(s) before conditionally sending the Retarget message to each child view. For more information, see the slot view's redefinition of Retarget (page 3-59).

DoRetarget

myEntryView:DoRetarget()

If received by the entry layer, it performs a ReTarget on itself. If received by the layout layer, it performs a ReTarget, with a non-nil value, on itself.

newtFalseEntryView

This proto, which is based on newtEntryView, allows the use of the NewtApp framework's slot view protos and stationary without the rest of the NewtApp structure for updating entries. It is ideal for converting an existing non-NewtApp application to use the NewtApp slot view protos.

When you use slot views or stationary outside a NewtApp application, you must put them in a newtFalseEntryView proto and make sure the target and targetView slots are set. This is accomplished by sending a Retarget message to the newtFalseEntryView whenever entries are changed.

Writing a changed entry back to the soup is the responsibility of the application. You may want to set up a flush timer, or at least write back changes when scrolling and closing.

Slot descriptions

`targetSlot` Optional. Defaults to `'target`. There's no need to reset it if the slot in the parent context of this view, which holds the current entry (or target), is named `target`. If not, set it to the symbol that refers to the slot in the parent context that holds the data from the target entry.

`dataCursorSlot` Optional. Defaults to `'dataCursor`. There's no need to reset it if the slot in the parent context of this view, which refers to the main soup cursor, is named `dataCursor`. If not, set it to the symbol that refers to the slot in the parent context that refers to the main soup cursor.

`dataSoupSlot` Optional. Defaults to `'dataSoup`. There's no need to reset it if the slot in the parent context of this view, which refers to the main soup, is named `dataSoup`. If not, set it to the symbol that refers to the slot in the parent context that refers to the main soup.

`soupQuerySlot` Optional. Defaults to `'soupQuery`. There's no need to reset it if the slot in the parent context of this view, which refers to the soup query, is named `soupQuery`. If not, set it to the symbol that refers to the slot in the parent context that refers to the soup query.

The `newtFalseEntryView` inherits all the methods documented in the `newtEntryView` proto, although they have been altered slightly to provide a simulated NewtApp application environment.

newtRollEntryView

This proto is based on the `newtEntryView` proto and is equivalent to it, except that it supports the roll style application (as implemented by the `newtRollEntryView` proto). It dynamically sizes the entries, depending on the size of the viewDef. You must use stationery with this proto.

Slot descriptions

`target` Set by the system to point to the current entry.

`targetView` Refers to the `newtRollEntryView` proto itself, so that routing and other system services can use it.

`bottomlessHeight`
Optional. Sets the height of the entry view when it is the last item in a roll style application. Set to the constant `kEntryViewHeight`.

newtEntryPageHeader

This proto implements the standard header/divider bar for a page entry view. If this header is displayed in association with some stationery (a dataDef is the current target entry) and it has an icon assigned to its `icon` slot (page 3-48) that icon is used at the far left of the header. Otherwise a default icon provided by the system is used.

When you press the header icon on the left of the bar, the `newtInfoBox` proto page 3-47 is automatically opened. If your entry has a `title` slot, the title is displayed in the area where the date is shown; otherwise, the date is displayed. You can see all these features in the built-in Notes application.

Figure 3-11 A page header

newtEntryRollHeader

This proto implements the standard header/divider bar in a roll entry view. If this header is displayed in association with some stationery (a dataDef is the current target entry) and if the dataDef has an icon assigned to its `icon` slot (page 3-48), it is used at the far left of the header. Otherwise, a default icon provided by the system is used.

When you tap the header icon, a `newtInfoBox` proto (page 3-47) is automatically displayed. If your entry has a `title` slot, the title is displayed; otherwise, the date is displayed. You can see all of these features in the built-in Notes application. A roll header is shown in Figure 3-12.

Figure 3-12 A roll header

Slot descriptions

`hasFiling` Optional. Defaults to `true`. Set to `nil` for no Filing or Action buttons.

`resizable` Optional. Defaults to `true`. Set to `nil` for no drag resizing.

newtEntryViewActionButton

This is the standard Action button. It must be a child of the entry view. It handles the usual routing actions, but in the entry view rather than the application base view context.

newtEntryViewFilingButton

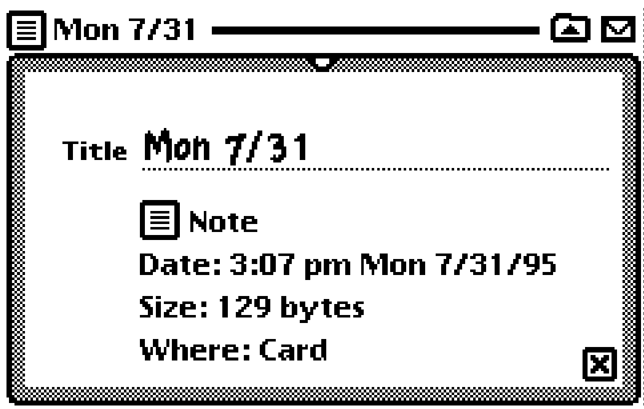
This is the standard Filing button, but it must be a child of the entry view. It handles the usual filing actions, but in the entry view rather than the application base view context.

newtInfoBox

This is a floating view based on `protoFloatNGo`. It displays informational text including the date, the size of the target entry, and the storage location of the entry. It also contains an input line with the label "Title." If the text on that line is changed, the new text is saved automatically and displayed next to the icon on the title bar after the proto is closed.

If your application uses stationery, the icon you declared in the `icon` slot is used next to its description, which is also taken from the dataDef. You need to add nothing to get a view that looks very similar to the one from the built-in Notes application shown in Figure 3-13.

Figure 3-13 A NewtApp Information slip



Slot descriptions

`icon`	Optional. An icon representing the object about which the information is provided.
`description`	Optional. A string describing the entry being displayed.

Slot View Protos

The slot view protos include all the protos you use to view and edit the data held in the slots of a soup entry. The slot view protos usually have a one-to-one correspondence with soup slots.

There are two categories of slot views:

- Simple read-only (RO) and edit views
- Labelled input-line protos

All slot views assume a soup entry has been set by the parent proto as the value of the `target` slot. The `target` slot is a reference to the soup entry containing the data to be displayed in a slot view. This soup entry will also stores the user-entered data.

This is set at run time by the NewtApp framework, where `target` is a slot defined in the `newtApplication` base view. The `targetView` is the `newtEntryView` proto that contains the slot view in which the target data is to be displayed.

When slot views are used outside a NewtApp application, the `target` and `targetView` slots must be set by you. In this case, the slot view protos must be contained by a `newtFalseEntryView` proto (page 3-44), which must be the view referred to by the `targetView` slot.

Slot views also require a `path` slot. Depending on the proto, this slot must be a path expression leading to a slot that holds a certain kind of data. For instance, the `path` slot of a `newtROTextDateView` proto must refer to a slot in an entry that contains dates.

Also included in this view category are two protos: `newtEntryLockedIcon`, (page 3-59) which you can use to indicate locked media or read-only views and `newtStationeryView` (page 3-59) which provides a bounding box for your dataDef stationery component.

Slot description

`path` Required for all slot views. A symbol that is a path expression to the slot in the target frame where the initial value for the input line resides, and in which the final value is to be stored.

The slot identified by the path expression should contain the specified data for the specific slot view.

Also defined for the slot view protos is a `TextScript` method that displays the text for the target entry and a `JamFromEntry` method that puts the path of a new entry into the `path` slot. These work for all simple slot views.

TextScript

mySlotView`:TextScript()`

Returns a text representation of the data at the specified path in the target soup entry for any slot view in your application.

JamFromEntry

mySlotView`:JamFromEntry(`*otherEntry*`)`

Replaces the path expression in the `path` slot with a new path expression. The new path is formed by appending the value of the *otherEntry* parameter to the path expression that leads to the soup entry in which the slot resides, which it obtains from the `jamSlot` slot (if it's not `nil`).

This essentially resets the target entry to a different entry and causes the display to change so the user is looking at the new value.

otherEntry A soup entry. This is intended to be an entry other than the one to which the `entryView` is already targeted.

For an example of when you might want to use this method, imagine you are developing an order-entry system. You want the customer address stored in the order, but it's in the Names soup. To extract the data, you set the `jamSlot` to a path expression that leads to the address in the Names soup and send the `JamFromEntry` message with the Names soup entry as the value of the *otherEntry* parameter.

newtROTextView

This proto displays read-only text. It is the base proto for the rest of the simple slot views.

Slot descriptions

`path` — Required. The slot identified by this path expression is the slot from which to get the initial text to display in this view, and in which to store the final value.

`styles` — Optional. Defaults to `nil`.

`tabs` — Optional. Defaults to `nil`.

`jamSlot` — Optional. Defaults to `nil`. If this view has a `jamSlot` that is not `nil`, the slots from an entry passed to the `JamFromEntry` method are placed ("jammed") into the soup slot referred to by `path`.

The `jamSlot` may be set to a path expression that defines the path to use to extract data from a slot in an entry, when the entry is not the one already targeted by the entry view (which encloses the slot view).

See also the methods `TextScript` and `JamFromEntry` in "Slot View Protos" (page 3-49).

newtTextView

This is the other base proto for the slot views; it is based on the read-only text view (`newtROTextView`). Use it to display editable text that does not need a label.

Slot descriptions

`path` — Required. The slot identified by this path expression is the slot from which to get the initial text to display in this view, and in which to store the final value.

`styles` — Optional. Defaults to `nil`.

`tabs` — Optional. Defaults to `nil`.

`jamSlot` Optional. Defaults to `nil`. If this view has a `jamSlot` that is not `nil`, the slots from an entry passed to the `JamFromEntry` method are placed ("jammed") into the soup slot referred to by `path`.

The `jamSlot` may be set to a path expression that defines the path to use to extract data from a slot in an entry, when the entry is not the one already targeted by the entry view (which encloses the slot view).

See also the methods `TextScript` and `JamFromEntry` in "Slot View Protos" (page 3-49).

newtRONumView

A read-only view for numbers, which is based on the NewtApp read-only text view (`newtROTextView`). It has functionality added for number formatting.

Slot descriptions

`path` Required. The slot identified by this path expression is the slot from which to get the initial text to display in this view, and in which to store the final value.

`format` Optional. The format string for displaying the data defaults to `%.10g` and a 10-place decimal. See `FormattedNumberStr` (page 23-17) for complete details.

`integerOnly` Optional. Defaults to `true`, signaling that conversion from text to number should result in an integer.

See also the methods `TextScript` and `JamFromEntry` in "Slot View Protos" (page 3-49).

newtNumView

An editable number view that is based on the read-only number view (`newtRONumView`) and inherits its slots. Specify number formatting by assigning values to the `format` and `integerOnly` slots.

Slot descriptions

`path`	Required. The slot identified by this path expression is the slot from which to get the initial text to display in this view, and in which to store the final value.
`format`	Optional. The format string for displaying the data defaults to `%.10g` and a 10-place decimal.
`integerOnly`	Optional. Defaults to `true`, signaling that conversion from text to number should result in an integer. A value of nil allows real (decimal) numbers.

See also the methods `TextScript` and `JamFromEntry` in "Slot View Protos" (page 3-49).

newtROTextDateView

This proto is set to contain text and dates. Depending on which of the two slots, `longFormat` or `shortFormat`, is non-`nil`, this proto displays either long or short dates, such as February 29, 1984, or 2/29/84.

Slot descriptions

`path`	Required. The slot identified by this path expression is the slot from which to get the initial text or date to display in this view, and in which to store the final value.
`longFormat`	Optional. Defaults to `yearMonthDayStrSpec`, which is a format for use by the `LongDateStr` function (page 17-23). The `longdate` specification is defined by the system. Either this slot or the `shortFormat` slot should not be `nil`, so the view can choose the format.
`shortFormat`	Optional. Defaults to `nil`. This is a format defined by the system for use by the `ShortDateStr` function (page 17-24).

See also the methods `TextScript` and `JamFromEntry` in "Slot View Protos" (page 3-49).

newtTextDateView

This editable view is based on its read-only version (`newtROTextDateView`) and inherits its slots.

Slot descriptions

`path` — Required. The slot identified by this path expression is the slot from which to get the initial text or date to display in this view, and in which to store the final value.

`longFormat` — Optional. Defaults to `yearMonthDayStrSpec`, which is a format for use by the `LongDateStr` function (page 17-23). The `longdate` specification is defined by the system. Either this slot or the `shortFormat` slot should not be `nil,` so the view can choose the format.

`shortFormat` — Optional. Defaults to `nil`. This is a format defined by the system for use by the `ShortDateStr` function (page 17-24).

See also the methods `TextScript` and `JamFromEntry` in “Slot View Protos” (page 3-49).

newtROTextTimeView

This proto is based on the `newtROTextView` proto, but has functionality added to display and format a time string. The slot to be displayed must contain a time or text.

Slot descriptions

`path` — Required. The slot identified by this path expression is the slot from which to get the initial text and/or time to display in this view, and in which to store the final value.

`format` — Optional. Defaults to `ShortTimeStrSpec` which is a format for use by the `TimeStr` function (page 17-27).

See also the methods `TextScript` and `JamFromEntry` in “Slot View Protos” (page 3-49).

newtTextTimeView

This editable view protos from its read-only version (`newtROTextTimeView`) and inherits its slots.

Slot descriptions

`path`	Required. The slot identified by this path expression is the slot from which to get the initial text and/or time to display in this view, and in which to store the final value.
`format`	Optional. A format for use by the `TimeStr` function (page 17-27). Defaults to `ShortTimeStrSpec`.

See also the methods `TextScript` and `JamFromEntry` in “Slot View Protos” (page 3-49).

newtROTextPhoneView

This view, which is based on the `newtROTextView` proto, displays a telephone number from the application soup.

Slot description

`path`	Required. The slot identified by this path expression is the slot from which to get the initial phone number to display in this view, and in which to store the final value.

See also the methods `TextScript` and `JamFromEntry` in “Slot View Protos” (page 3-49).

newtTextPhoneView

This view, based on the `newtROTextView` proto, formats a number entered into it by a user as a telephone number.

Slot description

`path` Required. The slot identified by this path expression is the slot from which to get the initial numbers to display in this view, and in which to store the final value.

See also the methods `TextScript` and `JamFromEntry` in "Slot View Protos" (page 3-49).

newtROEditView

This is a fixed-size edit view that displays the application soup. It may also be set up to have its own scrollers by setting the `optionFlags` slot.

Slot descriptions

`doCaret` Optional. Defaults to `true`, which autosets the caret.

`optionFlags` Optional. Defaults to `kNoOptions` (which has a numeric value of 0) and sets the scrollers not to show. The constant `kHasScrollersOption` (which has a numeric value of 1) sets them to show.

`viewLineSpacing`
Optional. Defaults to 28.

`path` Required. The slot identified by this path expression is the slot from which to get the initial numbers to display in this view, and in which to store the final value.

This proto also defines the method `ScrollToWord` for your convenience. See also the methods `TextScript` and `JamFromEntry` in "Slot View Protos" (page 3-49).

ScrollToWord

myeditView:`ScrollToWord(`*words*, *hilite*`)`

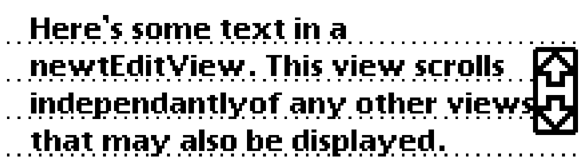
This method finds the specified word, scrolls the edit view to the found word, and highlights it—if the *hilite* parameter is `true`. If no match is found for the specified word in any view child of the edit view, `ScrollToWord` does nothing. This method does not work in roll layouts.

words	May be a string or an array of single words to find.
hilite	If `true`, the matching text of the paragraph view is highlighted.

newtEditView

This view protos is based on its read-only version (`newtROEditView`) and behaves simply, somewhat like a `clEditView`. (See "General Input Views" beginning on page 8-6 in *Newton Programmer's Guide*.) Unlike the read-only version, this proto accepts user-entered text. A `newtEditView`, with scroll bars showing, is shown in Figure 3-14. This proto can use any of the `NewtROEditView` slots (page 3-56).

Figure 3-14 A newtEditView proto



See also the methods `TextScript` and `JamFromEntry` in "Slot View Protos" (page 3-49).

newtCheckBox

This view is based on the `protoCheckBox` page 6-24. Basically, it works so the check mark is on when the value of the `target.(path)` slot is equal to the value of the `assert` slot. If you want more complex behavior, override the `ViewSetupFormScript` and the `ValueChanged` method.

Slot descriptions

`path`	Required. The slot identified by this path expression is the slot from which the initial text to display in this view is gotten, and in which the final value is to be stored.
`assert`	Optional. Defaults to `true`. Holds the "checked" value
`negate`	Optional. Defaults to `nil`. Holds the "unchecked" value.

The values of `assert` and `negate` are written back to and read from target.

See also the methods `TextScript` and `JamFromEntry` in "Slot View Protos" (page 3-49).

This proto also implements the following two methods.

ViewSetupFormScript

myCheckbox`:ViewSetupFormScript()`

Checks the value of `target.(path)`for equality against the value of the `assert` slot. Override this method for more complex behavior.

ValueChanged

myCheckbox`:ValueChanged()`

If the equality check in the `ViewSetupFormScript` is non-`nil`, the slot `target.(path)` is set to the `assert` value; otherwise, it is set to the `negate` value. Override this method for more complex behavior.

newtStationeryView

This view holds nothing; its function is to give a viewDef its bounding box. It contains the instantiated view of a ViewDef template. This proto is different from the `newtStationery` proto, page 4-3 which you use to create a dataDef.

newtEntryLockedIcon

You use this proto to show a lock icon if the slot is on locked media, on a ROM card, or contained in a read-only view. The `newtEntryLockedIcon` proto is set either to show or not show when your view is opened.

Slot description

`icon`	Optional. Defaults to `nil`; it may also have the value `lockedIcon`.

The following methods are defined internally to `newtEntryLockedIcon.` They should not be changed, or the proto does not work as documented.

Retarget

myLockedIcon`:Retarget()`

Calls `SetIcon` to show either the locked or unlocked icon (according to whether the store is locked or in ROM) and redraws the icon.

SetIcon

myEntryLockedIcon`:SetIcon()`

Checks the target soup entry to find out if it is or locked or in ROM. If it is, the locked icon is displayed.

Labelled Input-Line Slot View Protos

The NewtApp labelled input-line protos function similarly to the `protoLabelInputLine` family of protos. (If you are not familiar with those protos, you may read about them in Chapter 8, "Text and Ink Input and Display.")

In addition to their label and pop-up menu capabilities, these protos include the `flavor` and `access` slots. The `access` slot limits the type of access each label input-line slot view allows. The `flavor` slots contain references to the NewtApp filter protos. These protos assign appropriate pickers and correct formatting for the intended data type. They are enumerated in Table 3-1.

Table 3-1 The NewtApp filters used to set the `flavor` slot

Filter*	Description	Slots
`newtTextFilter`	This is the filter the other filter protos are based on. It allows the label input-line proto, which uses it as the value of its `flavor` slot, to accept text input.	This proto contains no slots for you to set.
`newtInteger Filter`	This filter is based on the `newtTextFilter` proto. It is set to accept only integers as input and contains a `format` slot, which you may set.	`format`: Optional. Defaults to `%.10g`. You should change this as needed.
`newtNumber Filter`	This filter is based on the `newtIntegerFilter` proto. It is set to accept all numbers as input and contains a `format` slot, which you may set.	`format`: Optional. Defaults to `%.10g`. You should change this as needed.

Table 3-1 The NewtApp filters used to set the `flavor` slot (continued)

Filter*	Description	Slots
`newtDateFilter`	This filter is based on the `newtTextFilter` proto. It is set to accept dates as input and contains two format slots, which you may set; one must be set to a non-`nil` value. This proto specifies that the `protoDatePopup` picker is to be used.	`shortFormat`: Optional. Defaults to `nil`. May be set to a format used by the `ShortDateStr` function. `longFormat`: Optional. Defaults to `yearMonthDayStrSpec`, a format used by the `LongDateStr` function.
`newtSimpleDateFilter`	This filter is based on the `newtDateFilter` proto and, is similarly set to accept and format dates. This filter allows dates that look like 5/15/55 or 5/15 and is useful for birthday input lines. It also contains two format slots, one of which must be set to a non-`nil` value.	`shortFormat`: Optional. Defaults to `nil`. May be set to a format used by the `ShortDateStr` function. `longFormat`: Optional. Defaults to `monthDayStrSpec`, which is the format used by the `LongDateStr` function to withhold the year.
`newtTimeFilter`	This filter is based on the `newtTextFilter` proto. It contains a `format` and `increment` slot, which you may set. If an input line of a `newtTimeFilter` flavor uses a pop-up menu, a `protoTimePopup` picker is specified by this proto.	`format`: Optional. Defaults to `shortTimeStrSpec`. You should change this as needed. `increment`: Optional. Defaults to 10.

Table 3-1 The NewtApp filters used to set the `flavor` slot (continued)

Filter*	Description	Slots
`newtDateN TimeFilter`	This filter is based on the `newtTextFilter` proto. It contains the slots `format`, `longFormat`, and `shortFormat`, which you may set. Note that of the two slots, `longFormat` and `shortFormat`, one must be set to a non-`nil` value. If an input line of a `newtDateNTimeFilter` flavor uses a pop-up menu, a `protoDateNTimePopup` picker is specified by this proto.	`shortFormat`: Optional. Defaults to `nil`. May be set to a format used by the `ShortDateStr` function. `longFormat`: Optional. Defaults to `yearMonthDayStrSpec`, the format used by the `LongDateStr` function to withhold the year. `format`: Optional. Defaults to `shortTimeStrSpec`. You should change this as needed.
`newtPhoneFilter`	This filter is on the `newtTextFilter` proto and is used to format numbers as phone numbers.	`kind`: Optional. Defaults to `nil`. The built-in types include `fax`, `home`, and `work`, and are used to change the label for the input line.
`newtCityFilter`	This filter is based on the `newtTextFilter` proto and is used to format text as cities.	This proto contains no slots for you to set.
`newtStateFilter`	This filter is based on the `newtTextFilter` proto and is used to format text as state names or abbreviations. If an input line of a `newtStateFilter` flavor uses a pop-up menu, a `protoLocationPopup` picker is specified by this proto.	This proto contains no slots for you to set.

Table 3-1 The NewtApp filters used to set the `flavor` slot (continued)

Filter*	Description	Slots
`newtCountryFilter`	This filter is based on the `newtTextFilter` proto and is used to format text as country names or abbreviations. If an input line of a `newtCountryFilter` flavor uses a pop-up menu, a `protoLocationPopup` picker is specified by this proto.	This proto contains no slots for you to set.
`newtSmartNameFilter`	This filter is based on the `newtTextFilter` proto and is used to present the Names soup to the user, who may choose a name that appears on the input line. If an input line of a `newtSmartNameFilter` flavor uses a pop-up menu, a `protoPeoplePopup` picker is specified by this proto.	This proto contains no slots for you to set.

*Filter names in the first column are all one word. They have been broken here due to space limitations.

newtProtoLine

The `newtProtoLine` is the base view for the input line protos. This proto inherits behavior from both the view class `clView` and the proto `newtROTextView`. In addition, it contains built-in code that creates the label picker and interprets menu item commands.

Most of the following slots are included for your information only. The only slot you should change for the built-in protos is the `label` slot. Do not change the access or flavor of the other slots; they will not work as planned.

Slot descriptions

`label` Optional. Defaults to the empty string. Provides a string containing the text you wish to display in the input-line label.

`labelCommands` Optional. An array of strings that should appear in a picker when the user taps the label. If this slot is supplied, the picker feature is activated and the label is shown with a diamond to its left to indicate that it is a picker. The currently selected item in the list, if there is one, is marked with a check mark to its left. A sample value is

```
["picker option one", "picker option two"]
```

`curLabelCommand`
Optional. If the `labelCommands` slot is supplied, this slot specifies which item in that array should be initially marked with a check mark. Specify an integer, which is used as an index into the `labelCommands` array. If you omit this slot, no item is initially marked with a check mark. Note that you must update this value when a different value is chosen.

`usePopup` Optional. Defaults to `true`. When set to `true` and you provide a `labelCommands` array, the input-line label displays a diamond, indicating a picker (pop-up menu).

`access` Optional. Defaults to `'readWrite`. Valid values include `'readWrite`, `'readOnly`, and `'pickOnly`. Do not change this value for the built-in protos, or they will not work as expected.

`flavor` Optional. Defaults to `newtFilter`. See Table 3-1 for a list of filters. Do not change this value for the built-in protos or they will not work as expected.

`memory` Optional. Defaults to `nil`. Used to reference a list of the last `n` items chosen. The value of this slot is a symbol that names the list. The symbol must incorporate your developer signature.

This proto also contains the following methods:

ChangePopup

myProtoLine:`ChangePopup(`*item*, *entry*`)`

Allows you to change a menu item before it is displayed (assuming there is a picker menu). For example, if you do a name query, but want to display “Bob Johnson, Apple” instead of just “Bob,” use this method. If `ChangePopup` isn’t defined, the menu just shows the original data.

item	An item to be displayed in the picker menu.
entry	The entry corresponding to the item selected from the picker menu.

UpdateText

myProtoLine:`UpdateText(`*newText*`)`

Updates text for an Undo action. It changes the old text to the text passed in as a parameter and posts that change to the Undo system service.

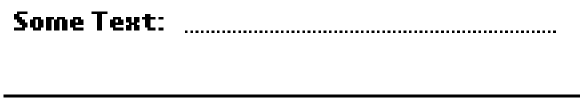
newText	A string to which the entry is changed, which is passed in as the parameter to this method.

newtLabelInputLine

This proto is used for a one-line input field that includes a text label and can optionally feature a pop-up menu. It is similar to `protoLabelInputLine`, and can use all of the slots available to that proto. It also shares some behavior (`jamSlot`, etc.) with the text view, and is based on the `newtProtoLine` proto.

The `newtLabelInputLine` proto is a one-line input field that includes a text label at its left. When a `labelCommands` array is provided, a diamond appears to the left of the label and the contents of the array appear in a picker menu. Without `labelCommands`, the `newtLabelInputLine` proto appears as shown in Figure 3-15.

Figure 3-15 A NewtApp label input line



Slot descriptions

`access` — Optional. Defaults to `'readWrite`. Valid values include `'readWrite`, `'readOnly`, and `'pickOnly`. Do not change this value for the built-in protos, or they will not work as expected.

`label` — Optional. Defaults to the empty string. Set to a string such as `“Some Text”`, which is the label text you wish to display.

`labelFont` — Optional. Sets the font used for the label. The default is `ROM_fontSystem9Bold`.

`labelCommands` — Optional. An array of strings that should appear in a picker when the user taps the label. If this slot is supplied, the picker feature is activated and the label is shown with a diamond to its left to indicate that it is a picker. The currently selected item in the list, if there is one, is marked with a check mark to its left. A sample value is:
`["picker option one","picker option two"]`

`curLabelCommand` — Optional. If the `labelCommands` slot is supplied, this slot specifies which item in that array should be initially marked with a check mark. Specify an integer, which is used as an index into the `labelCommands` array. If you omit this slot, no item is initially marked with a check mark.

`usePopup` — Optional. Defaults to `true`. When set to `true` and you provide a `labelCommands` array, the input line label displays a diamond, indicating a picker (pop-up menu).

`path` — Required. The path expression should identify the soup slot where the text is saved. An example is

```
[pathExpr: kAppSoupSymbol, 'someText]
```

`flavor` — Set to `newtTextFilter`; do not change this, or the proto will not work as expected.

newtROLabelInputLine

This is the same as `newtLabelInputLine`, except that there is no dotted line and the text displayed is read-only.

Slot descriptions

`label` — Optional. Defaults to the empty string. Provide a string containing the text you wish to display in the input-line label. An example is

```
"Some Text:"
```

`path` — Required. The path expression should identify the soup slot where the text is saved. An example is

```
[pathExpr: kAppSoupSymbol, 'someText]
```

`flavor` — Set to `newtNumberFilter`; do not change this, or the proto will not work as expected.

newtROLabelNumInputLine

This proto (the read-only version and its editable counterpart) is the numeric equivalent of the `newtLabelInputLine` protos. It is based on the `newtProtoLine` proto, but has a `newtNumberFilter` as the value of its `flavor` slot, which imparts number formatting features to it.

The read-only display consists of the label designated in the `label` slot and the data stored in the location specified by the `path` slot, but without a dotted line for the input line. Note that it is not possible to create a picker for a `newtROLabelInputLine`. An example is shown in Figure 3-16.

Figure 3-16 A NewtApp label display line for text

Slot descriptions

`label` Optional. Defaults to the empty string. Provides a string containing the text you wish to display in the input-line label. An example of a valid value is

```
"A Number:"
```

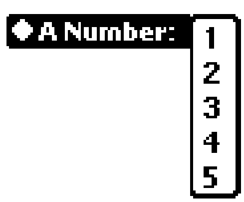
`path` Required. A path expression of the form:

```
[pathExpr: yourSoupSymbol, 'aNumber]
```

newtLabelNumInputLine

This is the same as the `newtROLabelNumInputLine`, except that data may be entered on the dotted input line and is saved to the data location specified in the `path` slot. The proto, with a `labelCommands` array with the specified value `["1","2","3","4","5"]` and a `true` value for the `usePopup` slot, is shown in Figure 3-17.

Figure 3-17 A NewtApp label number input line



Slot descriptions

`access` Optional. Defaults to `'readWrite`. Valid values include `'readWrite`, `'readOnly`, and `'pickOnly`. Do not change this value for the built-in protos, or they will not work as expected.

`label` Optional. Defaults to the empty string. Provides a string containing the text you wish to display in the input line label. An example of a valid value is

```
"A Number:"
```

`labelCommands` Optional. An array of strings that should appear in a picker when the user taps the label. If this slot is supplied, the picker feature is activated and the label is shown with a diamond to its left to indicate that it is a picker. The currently selected item in the list, if there is one, is marked with a check mark to its left. A sample value is: `["1","2","3","4","5"]`

`usePopup` Optional. Defaults to `true`. When set to `true` and you provide a `labelCommands` array, the input-line label displays a diamond, indicating a picker (pop-up menu).

`path` Required. A path expression of the form

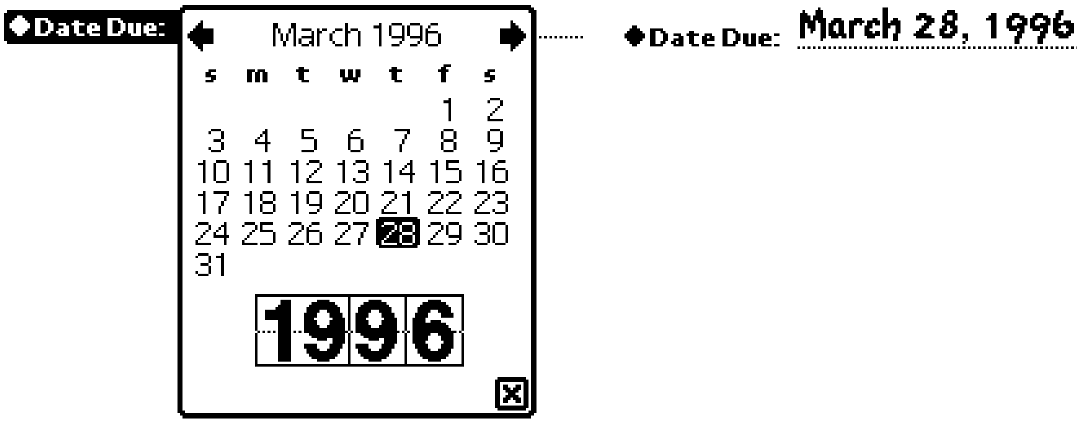
```
[pathExpr: yourSoupSymbol, 'aNumber]
```

`flavor` Set to `newtNumberFilter`; do not change this, or the proto will not work as expected.

newtLabelDateInputLine

This proto allows inputs of dates through a system-provided picker or by directly entering them on the input line. A label date input-line view is shown in Figure 3-18.

Figure 3-18 A NewtApp label date input line



When a date is entered on the input line, the calendar changes to match. If the date is written in any other format than the one shown in Figure 3-18, it is accepted and recognized but is changed automatically to the date format shown in the figure.

Note that neither the `labelCommands` nor the `usePopup` slot is necessary with this proto. The pop-up menu is specified in the `newtDateFilter`.

Slot descriptions

`access` Optional. Defaults to `'readOnly`. Valid values include `'readWrite` and `'pickOnly`. Do not change this value for the built-in protos, or they will not work as expected.

`label` Optional. Defaults to the empty string. Provides a string containing the text you wish to display in the input-line label.

`path` Required. A path expression that leads to a slot with a date in it, of the form

```
[pathExpr: soupSymbol, 'aDate]
```

flavor — Set to `newtDateFilter`; do not change this, or the proto will not work as expected.

newtROLabelDateInputLine

This is the same as the `newtLabelDateInputLine` except that it is used to display, not edit, a date from a soup slot. As with all the read-only input-line protos, the dotted line disappears when it is displayed. An example is shown in Figure 3-19.

Figure 3-19 A `newtROLabelDateInputLine` proto

Slot descriptions

`label` — Optional. Defaults to the empty string. Provides a string containing the text you wish to display in the input-line label.

`path` — Required. A path expression that leads to a slot with a date in it, of the form

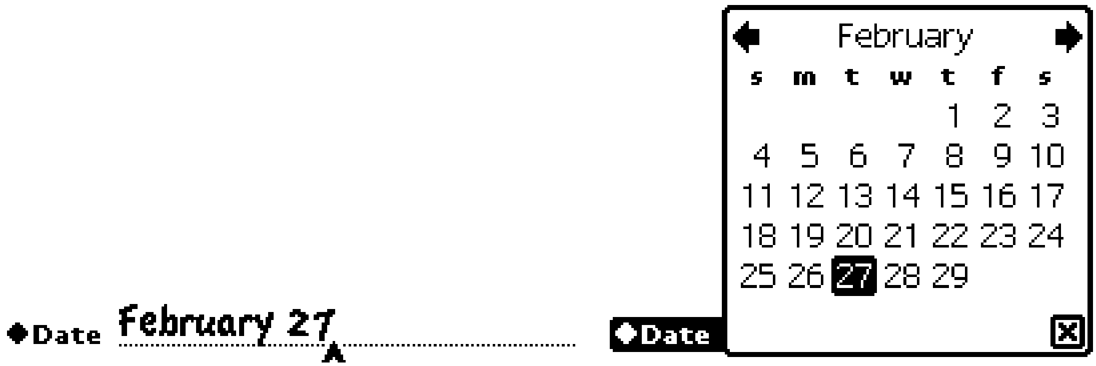
```
[pathExpr: soupSymbol, 'aDate]
```

`flavor` — Set to `newtDateFilter`; do not change this, or the proto will not work as expected.

newtLabelSimpleDateInputLine

This proto accepts simple dates (dates without the year, such as 7/24 and July 24) in addition to fully specified dates (such as 7/24/88 and July 24, 1988). It is useful for birthday and anniversary fields. The `newtLableSimpleDateInputLine` proto is based on the `newtProtoLine` proto. It is shown in Figure 3-20.

Figure 3-20 The simple date input line



Slot descriptions

`access` Optional. Defaults to `'readWrite`. Valid values include `'readOnly`, and `'pickOnly`. Do not change this value for the built-in protos, or they will not work as expected.

`label` Optional. Defaults to the empty string. Provide a string containing the text you wish to display in the input-line label.

`path` Required. A path expression that leads to a slot with a date in it, of the form:

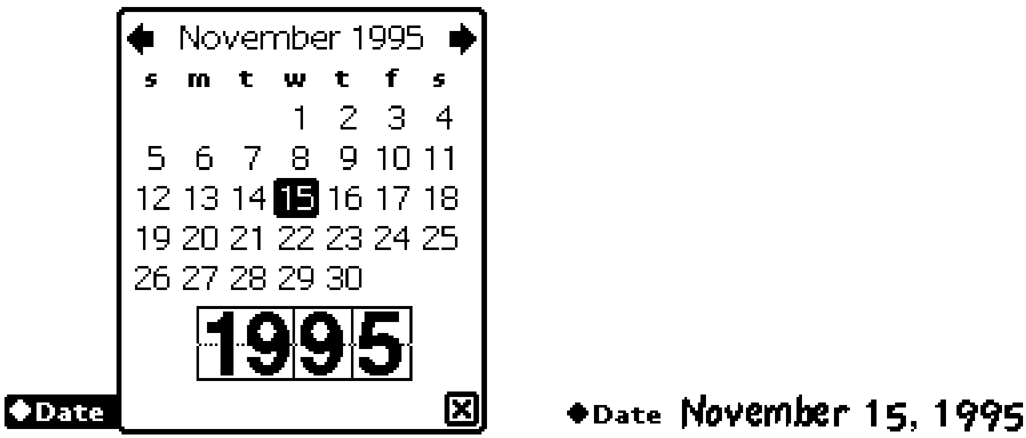
```
[pathExpr: soupSymbol, 'birthday]
```

`flavor` Set to `newtSimpleDateFilter`; do not change this, or the proto will not work as expected.

newtNRLabelDateInputLine

This proto is based on `newtProtoLine` and allows date input through a system-provided `protoDatePopup` picker. The initial display is simply the label with a diamond to its left and no input line following it. Once a date has been displayed, any attempt to edit it causes the date picker to display. It is shown in Figure 3-21.

Figure 3-21 Date input with picker-only access



Slot descriptions

`access` Optional. Defaults to `'pickonly`. Valid values include `'readWrite`, `'readOnly`, and `'pickOnly`. Do not change this value for the built-in protos, or they will not work as expected.

`flavor` Set to `newtDateFilter`; do not change this, or the proto will not work as expected.

`label` Optional. Defaults to the empty string. Provides a string containing the text you wish to display in the input-line label.

`path` Required. A path expression that leads to a slot with a date in it, of the form:

```
[pathExpr: yourSoupSymbol, 'date]
```

newtROLabelTimeInputLine

This proto is based on `newtProtoLine` and is set to display a time. No input or editing is recognized.

Slot descriptions

`label` Optional. A string which labels the input line.

`path` Required. A path expression, that leads to a slot with a time in it, of the form

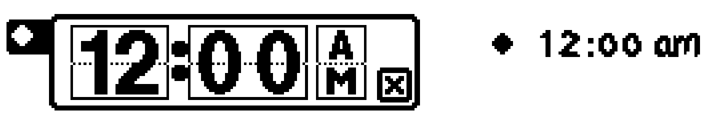
```
[pathExpr: soupSymbol, 'time]
```

`flavor` Set to `newtTimeFilter`; do not change this, or the proto will not work as expected.

newtNRLabelTimeInputLine

This allows date input through a system-provided `protoTimePopup` picker only. The picker is specified by the `newtTimeFilter`, which is the value of its `flavor` slot. You should not change this or the proto will not work as expected. It is based on `newtProtoLine`. It appears as shown in Figure 3-22.

Figure 3-22 Time input with picker-only access



Slot descriptions

`label` Optional. A string that labels the input line.

`flavor` Set to `newtTimeFilter`; do not change this, or the proto will not work as expected.

`access` Defaults to `'pickOnly`, can be `'readOnly`.

newtLabelTimeInputLine

This proto provides a labelled input line for a time. When it initially displays, the line is blank and a diamond appears to the left of the label. When the label is tapped, a time picker displays. It is shown in Figure 3-23.

Figure 3-23 A `newtLabelTimeInputLine` proto

Slot descriptions

`label` Optional. A string that labels the input line.

`flavor` Set to `newtTimeFilter`; do not change this, or the proto will not work as expected.

`path` Required. Must be a path expression identifying a soup slot that holds a time.

newtNRLabelDateNTimeInputLine

This proto is set up to contain times and dates, and is based on `newtProtoLine`. Depending on which of the two slots, `longFormat` or `shortFormat`, is non-`nil`, this proto displays either long or short dates, such as 10:05 AM, or 10:10 AM. For more information about these formats, which are used in calls to `LongDateStr` and `ShortDateStr`, see "Date and Time Format Specifications" (page 17-11).

Slot descriptions

`flavor` Set to `newtDateNTimeFilter`; do not change this, or the proto will not work as expected.

`access` Defaults to `'pickOnly` can be `'readOnly.`

`path` — Required. Must be a path expression identifying a soup slot that holds a date and time.

`longFormat` — Optional. Defaults to `yearMonthDayStrSpec`. The `longdate` specification as defined by the system. Either this slot or the `shortFormat` slot should be non-`nil` so the view can choose the format.

`shortFormat` — Optional. Defaults to `nil`. This is a `shortdate` specification as defined by the system. Either this slot or the `longFormat` slot should be non-`nil` so the view can choose the format.

newtLabelPhoneInputLine

This proto formats numbers as phone numbers, just like the `newtTextPhoneView` (page 3-56), except that this proto has a label. It is based on `newtProtoLine`.

Slot descriptions

`flavor` — Set to `newtPhoneFilter`; do not change this, or the proto will not work as expected.

`access` — Defaults to `'readWrite`.

`label` — Optional. Defaults to the empty string. Provide a string containing the text you wish to display in the input-line label.

`usePopup` — Optional. Defaults to `true`. When set to `true`, the input-line label displays a diamond, indicating a picker (pop-up menu).

`memory` — Optional. Defaults to `nil`. This keeps track of the most recent choices and displays them as items in the picker. The value of this slot is a symbol that names the list. The symbol must incorporate your developer signature.

newtAreaCodeLine

This proto is for numbers only and specifically for area codes. Double-tapping the input line displays the phone keyboard. It is based on `newtProtoLine`.

Slot description

`flavor`	Set to `newtPhoneFilter`; do not change this, or the proto will not work as expected.
`access`	Defaults to `'readWrite`.
`label`	Optional. Defaults to the empty string. Provide a string containing the text you wish to display in the area code line label.
`path`	Required. Must be a path expression identifying a soup slot that holds a area code.

newtAreaCodePhoneLine

Allows area code input, as well as phone number input. It contains the basic functionality for parsing phone numbers, and for updating, targeting, drawing, and setting up the views in which they occur. It is based on `newtProtoLine`.

Slot descriptions

`path`	Required. The slot identified by this path expression is the slot from which the initial text to display in this view is gotten, and in which the final value is to be stored.
`flavor`	Set to `newtPhoneFilter`; do not change this, or the proto will not work as expected.
`access`	Defaults to `'query`
`label`	Optional. Defaults to the empty string. Provide a string containing the text to display in the input-line label.

newtSmartNameView

This proto gets names from the Names application soup. It is based on `newtProtoLine`, so it also implements a label. When you use it, a tap on the picker menu item Other displays the `protoPeoplePopup` picker with the names from the Names soup. If you wish to control this behavior, you may implement your own version of the `JamFromEntry` method. See the sample in the section, "Creating a Custom Labelled Input-Line Slot View" (page 4-24) in *Newton Programmer's Guide*.

Slot descriptions

`flavor` — Set to `newtSmartNameFilter`; do not change this, or the proto will not work as expected.

`access` — Defaults to `'readWrite.`

`label` — Optional. Defaults to the empty string. Provide a string containing the text you wish to display in the input line label.

`usePopup` — Optional; the default is `nil`. If `true`, the proto creates a pop-up menu under the label. If the user chooses an item in the pop-up menu that item is displayed on the input line and the value of the target is changed to refer to the chosen soup entry. If the user chooses the menu item, Other, the `protoPeoplePicker` is displayed, allowing a choice from that soup.

`path` — Required. A path expression leading to the slot in the application soup where data changes should be stored.

Stationery Reference

This chapter documents the data structures, protos, and functions relevant to using dataDefs and viewDefs.

Data Structure

This section documents the viewDef frame.

viewDef Frame

You create a viewDef by basing it on a general view proto or class, such as a `clView`, and adding the slots specified here. Note that once the viewDef has been created it must be added to an application by using a `newtStationeryView` proto, as described in Chapter 4, "NewtApp Applications," in *Newton Programmer's Guide*.

Slot descriptions

`type` Required. The view types `'editor`, `'viewer`, and `'routeFormat` are used by the system and the built-in applications to collect specific kinds of viewDefs. For

	instance, the Newton routing code collects viewDefs of type `'routeFormat` (and `'printFormat`, for compatibility) and offers them as choices in the Format picker within the routing slip. You may also define custom types for your application.
`symbol`	Required. A symbol that identifies this view for the dataDef. One viewDef for each dataDef must have the `symbol` slot set to `'default`. This symbol is saved as a convenient reference by which to retrieve the view.
`name`	Required. A string that is used to build menus like the Show menu. An example of a suitable value is `"Note"`.
`version`	Required. This integer should match the version number of the dataDef.
`viewDefHeight`	Required, except in card-style applications. An integer that specifies a default height for applications that display data in a roll format. This value is not used by a card-style NewtApp application.

The following methods are used with viewDefs.

MinimalBounds

myViewDef:`MinimalBounds(`*entry*`)`

Returns the minimal enclosing bounding box for the data in a soup entry.

entry	A soup entry.

In a viewDef, you must use the `MinimalBounds` method if the height of the entry is dynamic, as it is in a paper roll-style application. This method is not necessary for a card-style application, which has a fixed height. If the entry size is static, use the `viewDefHeight` slot instead.

SetupForm

targetViewDef:`SetupForm(`*entry*, *entryView*`)`

Allows you to modify the data displayed by a viewDef before it is displayed. This function is called by the `ViewSetupFormScript` method of the entry view containing the viewDef to be displayed. Override this method to modify the data before it's instantiated.

entry	The target soup entry.
entryView	The target view, in which the soup entry is about to be displayed.

Protos

This section describes the `newtStationery` proto, which is used to construct a dataDef, and the stationery button protos.

newtStationery

You use this proto as the template when constructing a dataDef. Its basic function is to create the infrastructure for specified kinds of data; it is not a view proto.

Slot descriptions

`description`	Optional. A string describing this dataDef's data entry. An example is `"Lined note paper"`. This is used in the Information slip (`newtInfoBox` proto), which is seen when the icon on the header bar is tapped.
`height`	Required, except in card-style applications. This is the default height used by viewers that display the data type in a paper-roll format, like the built-in Notes application. This value should match the value in the `viewDefHeight` slot of the viewDef. It is not used by a card-style NewtApp application.
`icon`	Optional; a bitmap frame. If you provide an icon for this `dataDef`, it is used in the New menu (the `newtNewStationeryButton` proto); the header bar (`newtEntryRollHeader`); and in the Information slip

	(`newtInfoBox` proto), which is seen when the icon on the header bar is tapped.
`name`	Required. This string appears in the New button's picker to identify the dataDef. The New button (implemented by the `newtNewStationeryButton` proto) collects all the strings from the `name` slots of the registered dataDefs that have the same `superSymbol` slot value and displays them as items in the New picker. For example, the Notes application uses the string `"Note"` to identify one of its dataDefs.
`symbol`	Required. A unique symbol that identifies the data type (also known as the class) of the entries that are created using this dataDef. The example in this chapter uses the constant `kDataSymbol`, set to the value of `kAppSymbol`, as the value of both this slot and an optional `class` slot within the entry template. The value of the `symbol` slot is used by a `newtStationery` view to select the viewDef and dataDef to use for a given entry.
`superSymbol`	Required. A unique symbol used to identify the application with which this dataDef should be associated. The value of this slot must match the value of a `superSymbol` slot in the host application.
`version`	Required. This integer identifies the version number of the viewDef.

The following methods are defined in `newtStationery`.

FillNewEntry

myDataDef`:FillNewEntry(`*newEntry*`)`

Returns a modified soup entry when given a new entry as returned by the `CreateBlankEntry` method.

newEntry	A frame that is a soup entry, as returned by the `CreateBlankEntry` method (page 3-5), which is defined in the `newtApplication.allSoups` slot of a NewtApp application.

You should use this method to add a `class` slot value and the other application-specific data structures you require to the entry. It is recommended that you put application-specific data structures in a slot embedded within the entry. For an example of this, see "Using FillNewEntry" beginning on page 5-6 in *Newton Programmer's Guide*.

MakeNewEntry

myDataDef`:MakeNewEntry()`

Returns a frame that will be added to some soup to make an entry. This method is used only if `FillNewEntry` does not exist. However, it is useful if you are creating stationery as an auto part instead of as part of a NewtApp application. Furthermore, if the application using this dataDef has no `CreateBlankEntry` method, then `MakeNewEntry` is called.

StringExtract

myDataDef`:StringExtract(`*entry*, *nLines*`)`

Called by overviews and Find to get a string description of an entry for display in an overview. You must supply a version of this method that creates a string description from your soup entry.

entry	A soup entry.
nLines	An integer specifying if your method should return one or two lines of text.

TextScript

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

Extracts a text version of an entry for use by routing (for example, as an e-mail message).

item	The In/Out Box item frame. The data being routed is stored in the `body` slot of this frame. Because the `body` slot might contain an alias constructed by the Routing interface, in order to access it you should always call the `ResolveBody` routing format method (page 18-15) on

item. `ResolveBody` returns the data in the `body` slot whether or not it is referenced by an alias.

target The soup entry that is being routed.

This method must return a string containing the data you want to be routed from the soup entry.

newtStationeryPopupButton

This button proto is used as the basis for both the `newtNewStationeryButton` and the `newtShowStationeryButton`; the former displays a list of dataDefs, and the latter a list of viewDefs.

The `newtStationeryPopupButton` is based on the `protoPopupButton`, thus incorporating the necessary functionality for creating a picker for the stationery buttons. It also includes the `StatScript` method, which you must define to assign an action to a picker choice, and the `SetUpStatArray` method, which you may override to intercept or tweak the stationery items before they are displayed in the picker.

The methods `BuildPopup` and `ViewSetupFormScript` are defined internally to `newtStationeryPopupButton`. If you need to use one of these methods, be sure to call the inherited method first (for example, `inherited:?ViewSetupFormScript()`); otherwise the proto may not work as expected.

▲ **WARNING**
Do not override the internally defined methods `ButtonClickScript`, `PickActionScript`, and `PickCancelledScript`. ▲

Slot descriptions

`text` — Required. A string that is the text displayed in the button. An example is `"New"`.

`form` — Required. A symbol that determines which form of stationery is shown in the picker. Specify either `'viewDef` or `'dataDef`.

`symbols` — Optional. Specifies the list of stationery to display in the picker. This slot defaults to `nil`, which indicates that all stationery of the kind indicated in the `form` slot are to be displayed in the picker. If you don't want all the stationery, you can specify an array of unique symbols.

When collecting viewDefs, specify an array of dataDef symbols in this slot. All viewDefs registered for those dataDefs are collected. When collecting dataDefs, specify an array of `superSymbol` symbols. In this case, all dataDefs whose `superSymbol` slot matches one of the specified symbols are collected.

`types` — Required when the `form` slot is set to `'viewDef`. This slot indicates which types of viewDefs are to be included in the picker. This slot must contain an array of viewDef type symbols, for example: `['viewer, 'editor, 'symbolYouDefined]`.

This slot is ignored if the `form` slot is set to `'dataDef`. The default value of this slot is `nil`.

`sorter` — Optional. The default is the symbol `'|str<|` for sorting in alphabetical order. Set to `nil` to prevent sorting.

This slot can be set to any of the string sort tests defined for the *test* parameter in "Sorted Array Functions" (page 23-43).

`shortCircuit` — Optional. A Boolean that controls the pop-up behavior of the button. This slot defaults to `true`. When it is set to `true` and there is only one item in the stationery picker array, the diamond normally displayed to the left of the text in the button is not shown. Tapping the button does not display a picker but instead causes the action to occur with the single item. Set this slot to `nil` if you prefer a picker list with one item.

The following methods are defined in this proto.

SetUpStatArray

popupButton`:SetUpStatArray()`

Returns a list of stationery to display in the picker. Override this method to change or intercept what is displayed in the picker.

The default method returns the stationery array to be used in the picker by calling the `GetDefs` function (page 4-14) with the values you provide for the `form`, `symbols`, and `types` slots as its parameters.

StatScript

popupButton`:StatScript(`*stationeryItem*`)`

Called when an item is chosen from the stationery picker. This method should perform an action appropriate for the chosen stationery item.

stationeryItem The stationery that corresponds to the item chosen from the popup menu. It can be either a viewDef or a dataDef, depending on which is specified in the `form` slot.

newtNewStationeryButton

This proto implements the New button. This button collects the dataDef stationery for your application and includes them in a picker that is displayed when the user taps the New button. If an icon exists for a dataDef, it is also displayed in the picker list, next to the stationery name.

If there is only one dataDef for your application, the default behavior of this button is to hide the diamond that indicates it's a picker. If more than one dataDef exists for the application, the diamond appears at the left of the button. You can control this behavior by changing the `shortCircuit` slot (page 4-7). An example of this can be seen in the built-in Calls application, where the New button is used to create a New entry and display a blank page. The Calls application menu bar is shown in Figure 4-1.

Figure 4-1 Calls application menu bar

When a picker item is chosen, the proto (through the `StatScript` method) adds a new entry (defined by the dataDef) to the application soup and displays the blank entry. If you wish to perform other actions when the user chooses an item, override the `StatScript` method (inherited from the `newtStationeryPopupButton` proto) and be sure to call the inherited method in your code.

The `newtNewStationeryButton` picker that appears in the built-in Names application is shown in Figure 4-2.

Figure 4-2 `newtNewStationeryButton` in Names

The `newtNewStationeryButton` proto is based on `newtStationeryPopupButton`, and thus inherits its methods and slots.

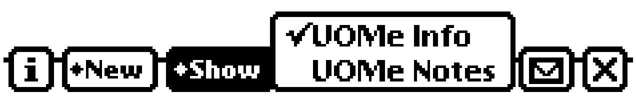
newtShowStationeryButton

This proto implements the Show button. This button collects the viewDef stationery for your application and includes them in a picker that is displayed when the user taps the Show button. If an icon exists for a viewDef, it is also displayed in the picker list, next to the stationery name.

If there is only one viewDef for your application, the default behavior of this button is to hide the diamond that indicates it's a picker. If more than one viewDef exists for the application, the diamond appears at the left of the button. You can control this behavior by changing the `shortCircuit` slot (page 4-7).

You should use a Show button when you want to be able to extend your application with multiple views of the data. For instance, you may wish to allow a choice between an informational view and an editable view, in which the user can enter notes, as shown in Figure 4-3.

Figure 4-3 `newtShowStationeryButton`



When a picker item is chosen, that viewDef is displayed and a checkmark is placed next to the picker item to indicate which is the current viewDef. If you wish to perform other actions when the user chooses an item, override the `StatScript` method (page 4-8) inherited from the `newtStationeryPopupButton` proto, and be sure to call the inherited method in your code.

The `newtShowStationeryButton` proto is based on `newtStationeryPopupButton`, and thus inherits its methods and slots. The following slot is different.

Slot description

`types` This slot indicates which types of viewDefs are to be included in the picker. This slot must contain an array of viewDef type symbols, for example: `['viewer, 'editor, 'symbolYouDefined]`.

The default value of this slot is `['viewer, 'editor]`.

newtRollShowStationeryButton

This Show button is based on the `newtShowStationeryButton`; this version is meant to be used within a page- or roll-style application. It has all the same slots and methods as `newtShowStationeryButton`.

Again, if you wish to modify the `StatScript` method (page 4-8), make sure to call the inherited method.

newtEntryShowStationeryButton

This Show button is based on the `newtShowStationeryButton`; this version is meant to be used within the entry view of an application. Like the `newtShowStationeryButton`, it allows the user to change the viewDef being displayed. However, unlike that proto, this occurs for only the entry being displayed. This enables a different view for each entry. For instance, one entry might be a note, while another might be an information view.

Functions

This section describes global functions used to register stationery components and retrieve information about them.

RegDataDef

`RegDataDef(`*dataDefSym*, *newDefTemplate*`)`

Registers a dataDef with the system. The return value of this function is undefined and you should not rely on it.

If you build an application using the NewtApp framework protos, the base view proto, `newtApplication`, automatically registers any dataDefs you create by using the values you put in its `allDataDefs` slot. For more information see "Registering DataDefs and ViewDefs" beginning on page 4-20 in *Newton Programmer's Guide*.

dataDefSym The symbol that uniquely identifies the dataDef you wish to add to the system registry. The symbol is the value of the dataDef's `symbol` slot. An example of an appropriate value is `'|IOU:PIEDTS|`

newDefTemplate The dataDef template. If you've defined the dataDef in a layout file in NTK, the template may be obtained with a call like this:

```
GetLayout("iouDataDef");
```

UnRegDataDef

`UnRegDataDef(`*dataDefSym*`)`

Unregisters a dataDef registered by `RegDataDef`. The return value of this function is undefined and you should not rely on it.

dataDefSym The symbol that uniquely identifies the dataDef you wish to remove from the system registry. The symbol is the value of the dataDef's `symbol` slot.

RegisterViewDef

`RegisterViewDef(`*viewDef*, *dataDefSym*`)`

Registers a viewDef view template or routing format frame under the unique identifying symbol of its corresponding dataDef in the system registry. The return value of this function is undefined and you should not rely on it.

viewDef The viewDef view template or routing format frame.

dataDefSym The symbol identifying the dataDef associated with this viewDef. This symbol corresponds to the class of data with which this viewDef or routing format can be used.

If you build an application using the NewtApp framework protos, the base view proto, `newtApplication`, automatically registers any viewDefs you create by using the values you put in its `allViewDefs` slot. For more information see "Registering DataDefs and ViewDefs" beginning on page 4-20 in *Newton Programmer's Guide*.

If you are building an auto part extension, use a line of code like the following in its `InstallScript` function:

```
RegisterViewDef(GetLayout("defaultViewDef"),kDataSymbol);
```

UnRegisterViewDef

`UnRegisterViewDef(`*viewDefSym*, *dataDefSym*`)`

Removes a viewDef or routing format frame from the system registry. The return value of this function is undefined and you should not rely on it.

viewDefSym The symbol identifying the viewDef or routing format. This is the value of the `symbol` slot in the viewDef or routing format frame.

dataDefSym The symbol under which the viewDef or routing format was registered.

GetDefs

`GetDefs(`*form*, *symbols*, *types*`)`

Returns an array of dataDef or viewDef stationery that match the specified criteria.

form — This symbol determines which of the stationery forms is returned. Specify either `'viewDef` or `'dataDef`.

symbols — Specifies the list of stationery to return. Specify either `nil` or an array of symbols. The value `nil` causes this function to return all stationery of the kind indicated by the *form* parameter. If you don't want all the stationery, you can specify an array of unique symbols to select particular stationery.

When collecting viewDefs, specify an array of dataDef symbols in this slot. All viewDefs registered for those dataDefs are returned. When collecting dataDefs, specify an array of `superSymbol` symbols. In this case, all dataDefs whose `superSymbol` slot matches one of the specified symbols are returned.

types — Indicates which types of viewDefs are to be returned. This parameter is used only when the *form* parameter is set to `'viewDef`. It is ignored if the *form* parameter is set to `'dataDef`.

When the *form* parameter is set to `'viewDef`, *types* can be `nil` or an array of symbols identifying viewDef types to return. The symbols you specify in the array may be any of the built-in symbols (`'viewer`, `'editor`, or `'routeFormat`), or they may include symbols you define. Here is an example of a *types* array: `['viewer, 'editor, 'symbolYouDefined]`.

Specifying a `nil` value causes this function to return viewDefs of all types.

GetDataDefs

`GetDataDefs(`*dataDefSym*`)`

Returns a dataDef, given the value of its `symbol` slot.

dataDefSym The value of the `symbol` slot of a dataDef.

The following example uses the symbol defined for the built-in Notes application:

```
GetDataDefs('paperroll)

{_proto: {symbol: NIL,
         superSymbol: NIL,
         name: "",
         description: "",
         icon: {@59},
         version: 0,
         height: 200,
         metadata: NIL,
         MakeNewEntry: <function, 0 arg(s) #4277B9,
         MinimalBounds: <function, 1 arg(s) #4277D9,
         SetupForm: <function, 2 arg(s) , #4277F9
         StringExtract: <function, 2 arg(s), #427819
         textScript: <function, 2 arg(s) #427839>},
   symbol: paperroll,
   name: "Note",
   superSymbol: notes,
   description: "Note",
   icon: {bits: <bits, length 76>,
            bounds: {#37B70D}},
   version: 1,
   metadata: NIL,
   MakeNewEntry: <function, 0 arg(s) #467251>,
   StringExtract: <function, 2 arg(s) #467271>,
   textScript: <function, 2 arg(s) #467291>}
```

GetAppDataDefs

`GetAppDataDefs(`*superSymbol*`)`

Returns an application's dataDefs when passed the value of the `superSymbol` slot of that application.

superSymbol	The value of the `superSymbol` slot defined in an application.

GetEntryDataDef

`GetEntryDataDef(`*soupEntry*`)`

Returns a dataDef for a given soup entry.

soupEntry	The soup entry whose dataDef you want to get.

GetEntryDataView

`GetEntryDataView(`*soupEntry*, *viewDefSym*`)`

Returns a viewDef for a given soup entry.

soupEntry	The soup entry whose viewDef you want to get.
viewDefSym	A symbol identifying a viewDef. This is the value of the `symbol` slot of a viewDef.

GetViewDefs

`GetViewDefs(`*dataDefSym*`)`

Returns a frame containing the viewDefs that are registered for a particular dataDef. If none are found, this function returns `nil`.

dataDefSym	A symbol identifying a dataDef. This is the value of the `symbol` slot of a dataDef.

GetDataView

`GetDataView(`*dataDefSym*, *viewDefSym*`)`

Returns a specific viewDef registered for a particular dataDef.

dataDefSym A symbol identifying a dataDef. This is the value of the `symbol` slot of a dataDef.

viewDefSym A symbol identifying a viewDef. This is the value of the `symbol` slot of a viewDef.

Pickers, Pop-up Views, and Overviews Reference

Data Structures

The `protoListPicker` uses a specialized data structure called a name reference and also has an array of column specification frames.

Name References

A name reference is a wrapper for a soup entry. A name reference has the following structure:

```
local aNameRef := {
   class: dataClass, usually a subclass of 'nameRef
   _unselected: true or nil,
   <application defined slots>,
};
```

Slot descriptions

`class` — A symbol specifying a registered data definition that can interpret the name reference. This is usually just `'nameRef` or a subclass of a name reference.

`_unselected` — A Boolean that determines whether an item is displayed as selected (in other words, checked) or not. By default, adding a name reference to the array of selections causes the name reference to be displayed as selected. However, if this slot is present and non-`nil`, the name reference is displayed with its checkbox unchecked.

This slot is useful if items are displayed for which no entry exists in the soup, and which should not be selected. For example, the system uses this slot when there are several possible locations for a meeting but only one can be chosen (`singleSelect` is `true`), and an item must be added to the list for each person attending the meeting. See the description of the `selected` slot in the description of `protoPeoplePicker` for more details.

Name references also have several global functions:

- The `IsNameRef` function determines whether a given item is a name reference.
- The `AliasFromObj` function returns an entry alias for an object.
- The `EntryFromObj` function returns the entry.
- The `ObjEntryClass` function returns the class of an entry (returned by the `EntryFromObj` function).

All these functions can be passed an alias, an entry, or a name reference. (If you pass any other type of object, the result is `nil`.)

To make a name reference, you can use the `MakeNameRef` method, as shown in the following example:

```
pickerDef.MakeNameRef := func(item, dataClass) begin
  local nameRef:= :MakeCanonicalNameRef(item,dataClass);
```

```
  if IsFrame(item) AND IsArray(nameRef.otherPrices) then
      Sort(nameRef.otherPrices, '|<|, nil);
nameRef;
end;
```

Column Specifications

A picker list for a `protoListPicker` is defined by an array of column specifications; each specification is a frame with the following slots.

Slot descriptions

`tapWidth` Required. An integer that specifies the width of the column. If this value is zero or negative, it's interpreted as a distance from the right margin of the view; if positive, it's considered a true width.

`fieldPath` Required. A symbol uniquely identifying the field that should be displayed in this column. This list picker uses this symbol to retrieve the data, and (in most cases, including the default case) is the actual path in the entry to the data field desired. However, it is possible to use the symbol purely as a marker—for example if the particular data required is a calculated aggregate of a number of data fields—as long as all the routines in the data definition that use this symbol are overridden to recognize this usage.

`optional` Optional. This slot tells the list picker that the contents of this field must be non-`nil` before the item may be selected. If `optional` is not set and the data specified by the `fieldPath` is `nil`, when an attempt to select the item is detected, the user is given the opportunity to fill in this field.

General Pickers

This section describes general-purpose pickers and pop-up views.

protoPopupButton

This proto is a text button that displays a picker when tapped. The button is highlighted while the picker is open. The picker appears to the right of the button if there's room; otherwise it appears to the left or slightly overlapping the button.

Figure 5-1 illustrates a pop-up button, without and with a picker.

Figure 5-1 Pop-up button and picker

The `ViewClickScript` method is used internally in the `protoPopupButton` and should not be overridden.

The `protoPopupButton` uses `protoTextButton` as its proto; `protoTextButton` is based on a view of the `clTextView` class.

Slot descriptions

`viewFlags` The default is `vVisible + vReadOnly + vClickable`.

`viewBounds`	Set to the size and location where you want the button to appear. If you do not set this slot, the button appears seven pixels to the right of its sibling. It is designed to be placed next to another button—in the status bar, for example.
`viewJustify`	Optional. The default setting is `vjSiblingRightH + vjCenterH + vjCenterV + oneLineOnly`.
`text`	A string that is the text inside the button.
`popup`	An array of items to be displayed in the picker list. See "Specifying the List of Items for a Popup" (page 6-37) in *Newton Programmer's Guide* for more information.

ButtonClickScript

picker`:ButtonClickScript()`

This method is called when the button is tapped. You can use this method if you want to construct the `popup` array dynamically. After setting the value of the `popup` slot, call the inherited `buttonClickScript` to preserve the pop-up behavior of the view. For example, `inherited:buttonClickScript()`.

PickActionScript

picker`:PickActionScript(`*index*`)`

This method is called when an item is selected from the picker list.

index	The index of the item that was chosen from the `popup` array.

If you don't supply this method, the button is simply unhighlighted. If you do supply this method, call the inherited method to unhighlight the button. For example, `inherited:PickActionScript(`*index*`)`.

If no item is selected because the user taps outside the list, the `PickCancelledScript` method is called instead.

PickCancelledScript

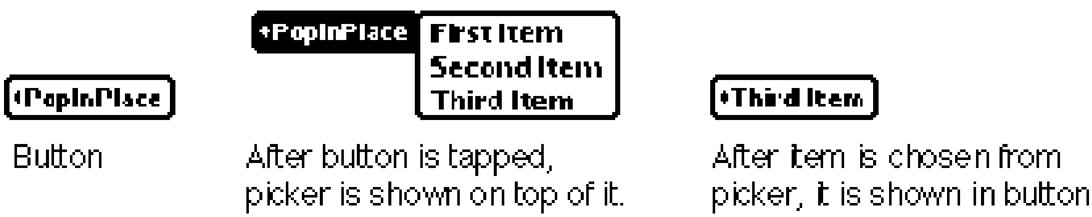
picker:`PickCancelledScript()`

This method is called if the picker is cancelled by a tap outside it. If you don't supply this method, the button is unhighlighted. If you do supply this method, call the inherited method to unhighlight the button. For example, `inherited:PickCancelledScript()`.

protoPopInPlace

This proto is a text button that displays a picker when it is tapped. When an item is chosen from the picker, the text of the chosen item appears in the button. Figure 5-2 shows an example of a `protoPopInPlace` text button.

Figure 5-2 A `protoPopInPlace` text button



Note that the `ViewSetupFormScript` is called multiple times; use the `ViewSetupDoneScript` to provide the initial text.

Also note that the `ViewClickScript` and `ButtonClickScript` methods are used internally; if you need to use one of these methods, be sure to call the inherited method.

The `protoPopInPlace` proto uses the `protoTextButton` as its proto; `protoTextButton` is based on a view of the `clTextView` class.

Slot descriptions

`viewBounds`	Set to the size and location where you want the button to appear. Note that the right bounds value is set automatically, based on the length of the text.
`viewFlags`	The default is `vVisible + vReadOnly + vClickable`.
`viewJustify`	Optional. The default setting is `vjCenterH + vjCenterV + noLineLimits`.
`text`	A string that is the text inside the button. This string must not begin with a space. Note that this string will be modified.
`popup`	An array of items to be displayed in the picker list. See “Specifying the List of Items for a Popup” (page 6-37) in *Newton Programmer’s Guide* for more information.

PickActionScript

picker`:PickActionScript(`*index*`)`

This method is called when an item is selected from the picker list.

index	The index of the item that was chosen from the `popup` array.

If you don’t supply this method, the button is unhighlighted. If you do supply this method, call the inherited method to unhighlight the button. For example, `inherited:PickActionScript(`*indes*`)`.

If no item is selected because the user taps outside the list, the `PickCancelledScript` method is called instead.

PickCancelledScript

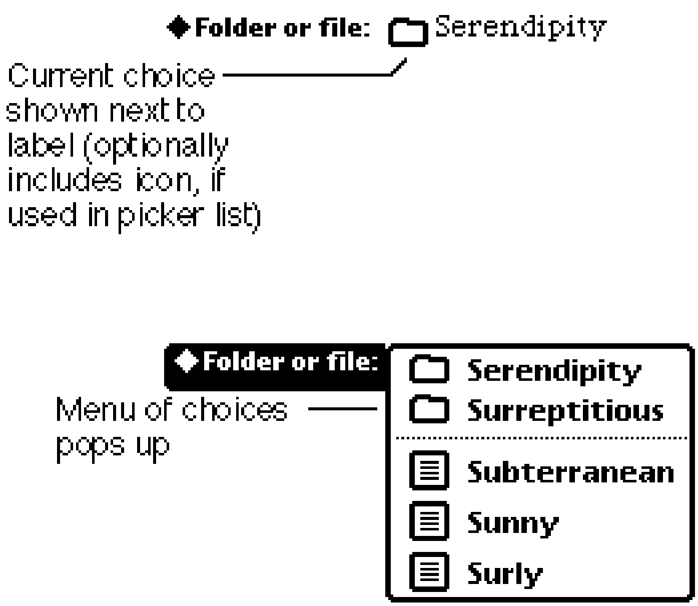
picker`:PickCancelledScript()`

This method is called if the picker is cancelled by a tap outside it. If you don’t supply this method, the button is unhighlighted. If you do supply this method, call the inherited method to unhighlight the button. For example, `inherited:PickCancelledScript()`.

protoLabelPicker

This proto is a label that displays a picker when it is tapped. The picker list can consist of simple strings, icons with strings, bit maps, or a two-dimensional grid (see “Specifying the List of Items for a Popup” (page 6-37) in *Newton Programmer’s Guide*). If the items are simple strings, the currently selected item is shown with a check mark next to it. The user can select a different item from the picker and that choice appears next to the label. Figure 5-3 shows an example.

Figure 5-3 A `ProtoLabelPicker`



The following methods are defined internally: `ViewSetupFormScript`, `ViewHiliteScript`, `ViewClickScript`, `PickActionScript`, and `PickCancelledScript`. 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.

Note that inking is automatically turned off when the view based on this proto is tapped.

The protoLabelPicker uses protoStaticText as its proto; protoStaticText is based on a view of the clParagraphView class. The protoLabelPicker itself implements the label portion of the proto. It has one child view, also a protoStaticText view, that implements the text value portion of the proto. This child view is named entryLine.

Here is an example of a template using protoLabelPicker:

```
myPicker := {...
_proto: protoLabelPicker,
viewBounds: RelBounds(10, 60, screenWidth-100, 16),
text: "Folder or file:",
lastchoice: nil,
labelCommands: [
   {item:"Serendipity", icon:folder, indent:25},
   {item:"Surreptitious", icon:folder},
   'pickSeparator,
   {item:"Subterranean", icon:doc},
   {item:"Sunny", icon:doc},
   {item:"Surly", icon:doc} ],

textSetup: func()
   lastChoice.item, // retrieve the last choice

iconSetup: func()
   lastChoice.icon,

LabelActionScript: func(index)
   lastchoice:=labelCommands[index].item,//store choice
...}
```

Slot descriptions

`viewBounds` Set to the size and location where you want the label with its item to appear. Note that if you use horizontal sibling-relative justification, you normally would specify relative values for the left and right bounds. For this proto, however, you must specify left and right bounds values whose difference equals the actual view width. The bounds values are used to calculate the width of the view that holds the text item.

`text` A string that is the text label. The label is drawn with a diamond to its left, to indicate to the user that this is a picker.

`labelCommands` An array of items that are the choices to be displayed in the picker. You can specify an array of strings, or you can specify an array of frames if you want the list items to appear as icons with strings. In the latter case, each frame represents one list item. See "Specifying the List of Items for a Popup" (page 6-37) in *Newton Programmer's Guide* for more information on specifying this list.

When `labelCommands` is an array of frames, you may want to provide the methods `TextSetup` and `IconSetup`. If `labelCommands` is an array of strings, you need only provide `TextSetup`.

To include a thin gray separator line, specify the symbol `'pickSeparator`. For a thicker black line, specify the symbol `'pickSolidSeparator`.

`iconBounds` Optional. Provide this bounds frame if you want the icon to appear next to the chosen item when the picker is not popped up (as in Figure 5-3). Specify the bounds of the largest icon in the list.

`iconIndent` Optional. The distance between the icon and the text when an icon/string item is shown next to the label. If you don't specify this slot, the default is 3 pixels.

`checkCurrentItem`

Optional. If non-`nil`, the currently selected item in the list, if there is one, is marked with a check mark to its

	left. If `nil`, check marks are not shown. Note that check marks are not shown for list items that are icons with strings.
`indent`	Optional. The distance, in pixels, to indent the picker from the beginning of the line (the beginning of the text label). If you don't include this slot, the picker is placed 6 pixels to the right of the text label by default.
`textIndent`	The distance from the left side of the view to the text in the picker list. This is set in the `ViewSetupChildrenScript` method of the proto.
`viewFont`	Optional. The font for the text label. The default is `ROM_fontSystem9Bold`.
`entryLine.viewFont`	Optional. This is the `viewFont` slot in the `entryLine` child of the `protoLabelPicker`. It sets the font for the text field to the right of the label. The default font is `editFont10`. This value is valid only at runtime, so if you want to change it, you need to do so in the `ViewSetupFormScript`.

LabelActionScript

picker`:LabelActionScript(`*index*`)`

When the user chooses an item from the picker, the new item is displayed next to the label and this method is called to allow additional processing.

index	The index of the item that was chosen from the `labelCommands` array.

TextSetup

picker`:TextSetup()`

This method is called to get the initial choice that should be shown next to the label when the view is being created. This method is passed no parameters and must return a text string (not a frame). It you don't include this method, the first item from the `labelCommands` array is used as the initial item.

IconSetup

picker`:IconSetup()`

This method is called to get the initial icon to display next to the label when the view is being created. This method is passed no parameters and must return an icon (not a frame). It you don't include this method, the icon associated with the first item from the `labelCommands` array is displayed.

TextChanged

picker`:TextChanged()`

This method is called whenever the value of the item is changed. If you don't supply this method, no default action occurs.

UpdateText

picker`:UpdateText(`*newItem*`)`

You can call this method to programmatically change the value of the text item. Note that you don't normally need to call this method; the text item is updated automatically when the user makes a selection from the picker.

newItem A string that is the new value for the text item.

UpdateIcon

picker`:UpdateIcon(`*newIcon*`)`

You can call this method to programmatically change the icon. Note that you don't normally need to call this method; the icon is updated automatically when the user makes a selection from the picker.

newIcon A bitmap for the new icon.

PickerSetup

picker`:PickerSetup()`

This method is called when the user taps the label; it gives you a chance to do your own processing, including setting up the `labelCommands` array.

This method should return non-`nil` if you want the default action to occur; that is, for the picker to pop up. If you return `nil`, the picker is not popped up. You must use this method or something else on your own. If you omit this method, non-`nil` is returned and the default action occurs.

PopIt

picker`:PopIt(`*position*`)`

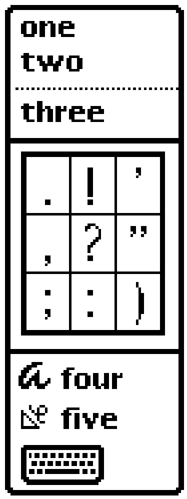
You can send this message to programmatically pop up the picker.

position The horizontal position of the picker; you should pass (`indent`–2) for this parameter.

protoPicker

The picker is a list of items (simple strings, bitmaps, two-dimensional grids, icons with strings, and separator lines) from which the user can choose one item by tapping it. Figure 5-4 illustrates different kinds of objects displayed in a picker.

Figure 5-4 Selection of items to choose



The `ViewSetupDoneScript` method is defined internally. If you need to use this method, be sure to call the inherited method also (for example, `inherited:?ViewSetupDoneScript()`), otherwise the proto may not work as expected.

The `protoPicker` is based on a view of the `clPickView` class.

Here is an example of a template using `protoPicker`:

```
picker := {...
   _proto: protoPicker,
   bounds: {left:34, top:66, right:96, bottom:96},
   viewFlags: vFloating+vReadOnly+vClickable,
   viewFormat: vfPen(2)+vfRound(4)+vfFrameBlack+
            vfFillWhite,
   pickItems: ["one",
      "two",
      'pickseparator,
```

```
      "three",
      'picksolidseparator,
      {bits:punctpict.bits, bounds:punctpict.bounds,
         width:3,height:3,cellFrame:1,outerFrame:2},
      'picksolidSeparator,
      {item:"four", icon:icon1, indent:15},
      {item:"five", icon:icon2},
      {item:keys}],
   PickActionScript: func(item)
      begin ...end,
   PickCancelledScript: func()
      Print("PickCancelledScript");
...}
```

Slot descriptions

`bounds` Must contain a `viewBounds`-like frame specifying a rectangle. The picker view is created so that one of its corners corresponds to one of the corners of the rectangle you specify. However, the system figures out exactly where to position the view, depending on how large it is and how much space is available around it. For example, it would normally be positioned so that its top-left corner corresponds to the top-left corner of the rectangle you specify. However, if you specify a location in the lower-right corner of the screen, where there won't be enough room for the picker, it will be positioned with its lower-right corner corresponding to the lower-right corner of the rectangle you specify.

Generally, a picker view appears as a result of tapping a button, word, or some other visible element. In most cases, simply specify the `viewBounds` slot of that element as the value of the `bounds` slot.

`viewBounds` This slot is ignored. Any value you place here is overwritten by the system, which calculates the value of this slot when the view is opened. The `bounds` slot controls the position of the view. The size of the view is determined by the width of the widest item and the total height of all items.

`viewFlags` The default is `vFloating + vReadOnly + vClickable`.

`viewFormat` Optional. The default setting is `vfFrameBlack + vfPen(2) + vfRound(4)`.

`viewJustify` Optional. The default setting is `vjCenterH + vjCenterV`.

`viewFont` Optional. The default font for text items in the list is `ROM_fontSystem10Bold`.

`viewEffect` Optional. The default view effect is `fxPopDownEffect`.

`pickItems` An array of items to be displayed in the picker list. There are many options, as described in "Specifying the List of Items for a Popup" (page 6-37) in *Newton Programmer's Guide*.

`pickTextItemHeight`
Optional. The height in pixels that should be reserved for each text item in the picker list. Note that each text item may actually occupy a height that is less than this amount. In this case, the item is vertically centered within the space. The default setting is 13 pixels.

`pickLeftMargin` Optional. The margin of blank space, in pixels, between the list entries and the view boundary on the left side. The default is 4.

`pickRightMargin`
Optional. The margin of blank space, in pixels, between the list entries and the view boundary on the right side. The default is 5.

`pickTopMargin` Optional. The margin of blank space, in pixels, above each bitmap item in the list. The default is 2.

`pickBottomMargin`
Optional. The margin of blank space, in pixels, below each bitmap item in the list. The default is 2.

`pickAutoClose` Optional. If the value of this slot is non-`nil` (the default), the picker is automatically hidden after the user selects an item by tapping it. If this slot is `nil`, the picker is not hidden after a selection is made. If you want to hide the view in this case, you must explicitly send it the `Hide` message. Regardless of the setting of this slot, the picker is automatically closed if the user cancels the list by clicking outside it.

`pickItemsMarkable`
: Optional. If the value of this slot is non-`nil`, space for marks is reserved at the left side of the list. If this slot is `nil` (the default), no space for marks is reserved. Note that space is reserved for marks if any of the list items has a mark specified, regardless of the setting of this slot.

`pickMarkWidth`
: Optional. The number of pixels of space to reserve for marks at the left side of the list. If you don't specify this value and marks are used, the space defaults to 10 pixels. All items are indented this amount.

`callbackContext`
: Optional. The view containing the `PickActionScript` and `PickCancelledScript` methods. If this slot is omitted, the picker view looks in itself for these methods.

PickActionScript

picker`:PickActionScript(`*itemPicked*`)`

This method is called when an item is selected from the picker list. If you don't supply this method, there is no default action. If no item is selected because the user taps outside the list, the `PickCancelledScript` method is called instead. Note that the `PickActionScript` method can be in the picker view itself or in a different view. If this method is in a different view, that view should be stored in the `callbackContext` slot.

itemPicked
: For a simple list, an integer that is the index of the selected item in the `pickItems` array is passed as a parameter to this method. For two-dimensional grids, a frame with three slots:

  `index` — The index of the grid item in the `pickItems` array.

`x`	The column index (zero-based) of the selected cell in the grid.
`y`	The row index (zero-based) of the selected cell in the grid.

PickCancelledScript

picker`:PickCancelledScript()`

This method is called if the picker is cancelled by a tap outside it and `pickAutoClose` is set to non-`nil`. If you don't supply this method, there is no default action. Note that the `PickCancelledScript` method can be in the picker view itself or in a different view. If this method is in a different view, that view should be stored in the `callbackContext` slot.

SetItemMark

picker`:SetItemMark(`*index*, *mark*`)`

You can call this method to set the mark character for an item in the list.

index	The integer index of the item whose mark you want to set.
mark	The character you want to set as the mark. Do not specify a string; you must specify a character (for example, `$>`). To set no mark for an item, specify `nil` for the character.

GetItemMark

picker`:GetItemMark(`*index*`)`

You can call this method to get the mark character for an item in the list. This method returns the character, or `nil` if the item has no mark character set.

index	The index of the item whose mark you want to get.

protoGeneralPopup

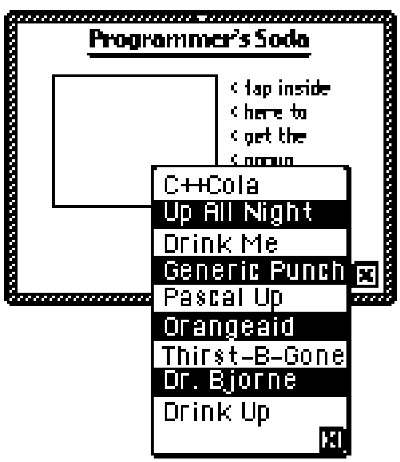
This proto provides a way to display a pop-up view that has a close box. You add your own custom children that also appear in the view. The pop-up view goes away (cancels) if a user taps outside of or taps the close box.

The `protoGeneralPopup` must have a `viewBounds` frame that is set to 0 width and 0 height. In addition, the `protoGeneralPopup` can have an `Affirmative` method that's called if the pop-up view is closed but not cancelled. The script takes no arguments.

Figure 5-5 shows an example of `protoGeneralPopup`. Notice that the close box is included by `protoGeneralPopup`.

`ViewQuitScript` is called by `protoGeneralPopup` and you should not override it.

Figure 5-5 Example of a pop-up view with a close box



Slot descriptions

`viewFlags` The default value is `vClickable + vFloating + vClipping`.

`cancelled`	Don't modify this slot. A Boolean indicating whether the user has cancelled the pop-up view; the default value is non-`nil`.
`context`	Don't modify this slot. The callback context. You do not need to change this slot; instead let `inherited?:New` handle the call back.

Affirmative

popup`:Affirmative()`

This method is called when the user closes the pop-up view without cancelling it, that is, using the close box, which accepts the changes.

New

popup`:New(`*bbox*`,` *callbackContext*`)`

You call this method to open the pop-up view.

bbox	A bounding box for the pop-up view. This box is only suggested; generally, you would use `:GlobalBox().`
callbackContext	The name of the view to which callback messages should be sent. Specify `self` if you define these methods in the initiating pop-up view.

PickCancelledScript

callbackContext`:PickCancelledScript()`

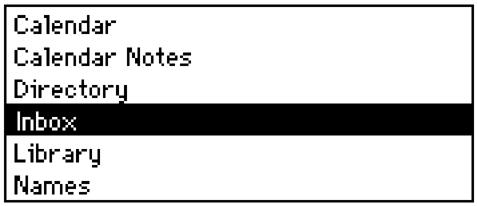
This method is called if the pop-up view is cancelled by tapping outside it. Take care when accessing your data.

protoTextList

This proto creates a scrollable list of items from which the user can choose one or more items by tapping. The selected items are highlighted in the list. The user scrolls the list by tapping the optional scroll arrows or tapping and dragging the pen either above or below the list.

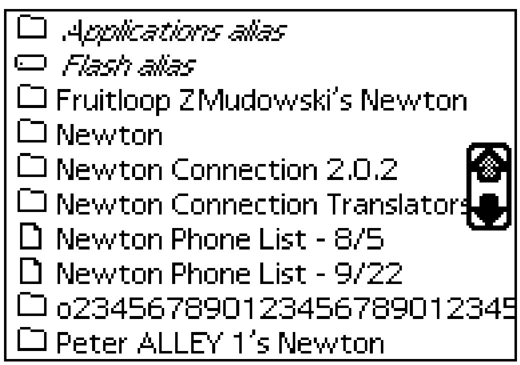
You can specify an array of strings as shown in Figure 5-6.

Figure 5-6 Scrollable list of items



Alternatively, you can specify an array of shapes that include both shapes and text as shown in Figure 5-7.

Figure 5-7 Scrollable list of shapes and text



The following methods are defined internally: `ViewClickScript`, `ViewSetupChildrenScript`, `ViewScrollDownScript`, `ViewScrollUpScript`, `DoScrollScript`, `HiliteLine`, `DrawHilite`, `SetChild`, `GetTotalLines`, `GetVisibleLines`, `GetViewHeight`, `GetViewWidth`, `GetLineHeight`, `ShowScrollers`, `SetViewHeight`, SetupList, InvertLine, and ButtonClickScript. If you need to use one of these methods, be sure to call the inherited method (for example, inherited:?ViewClickScript()) to retain full functionality.

The protoTextList is based on a view of the clView class. The protoTextList has a single child view (or more if scrollers are on), based on a view of the clTextView class (or clPictureView if shapes are shown), that displays the items in the list.

You can add or remove items from the list during run time by adding or removing items from the listItems array and then sending the view the SetupList and RedoChildren messages, in that order.

Slot descriptions

viewBounds	Set to the size and location where you want the list to appear. The value you set for the bottom bound is ignored. The bottom bound setting is calculated based on viewLines and viewFont unless viewLines is 0.
viewFont	Optional. The default font is ROM_fontSystem9.
viewFormat	Optional. The default setting is vfFillWhite + vfFrameBlack + vfPen(1).
viewLines	The number of lines to show in the list. This controls the height of the list view. If you don't specify viewLines, or if you specify 0, the number of lines that will fit in the bounds rectangle are calculated for you.
selection	Optional. This slot controls what is highlighted when the list is first displayed. On input, if you set selection to nil or -1, nothing is highlighted. You can set selection to the index of an item in the listItems array to highlight that item. The default setting is zero, highlighting the first item. On output, and while the protoTextList is displayed, selection contains the current selection. If the user doesn't select anything, selection is left as whatever the default was.
selectedItems	Optional. An array of selected items if multiple selection is enabled. Also contains the selected items when the user finishes making the selection.

`listItems` — An array of strings or an array of shapes that are the list items. Each item in the array corresponds to one line in the list. If you specify an array of shapes, each shape must be the same size. For shapes, the size of the selection highlight is based on the height of the shape. For text, the size of the selection highlight is based on the line height of the text. See "Specifying the List of Items for a Popup" (page 6-37) in *Newton Programmer's Guide* for more information.

`lineHeight` — The height of each line in pixels. Set by `setupList`.

`isShapeList` — Optional. Default is `nil`. Set to non-`nil` if using picts instead of text.

`useMultipleSelections` — Optional. Default is `nil`. Set to non-`nil` to allow multiple selections.

`useScrollers` — Optional. Default is `nil`. Set to non-`nil` to include scrollers.

`scrollAmounts` — If `useScrollers` is non-`nil`, you can specify an array of three integers representing lines, pages, and double-clicks. Default is `nil`.

The `protoTextList` scrolls using the `SetOrigin` method. Therefore, the slot `viewOriginY` contains the number of pixels the view is scrolled (and `viewOriginY DIV lineHeight` specifies the line number of the top displayed line). In addition, the `DoScrollScript` method scrolls the list by a specified offset.

DoScrollScript

list`:DoScrollScript(`*offset*`)`

This method scrolls the list by the specified offset.

offset — The offset, in pixels, by which to scroll.

ViewSetupFormScript

list:`ViewSetupFormScript()`

In this method, you must do two things: set the value of the `listItems` slot and call the internal method `SetupList`.

ButtonClickScript

list:`ButtonClickScript(`*index*`)`

This method is called after the pen is placed down and then lifted within the list. It is not called if the pen is lifted outside the bounds of the list.

index The index of the selected item in the `listItems` array.

Note that the selected item is kept in the `selection` slot. If `multipleSelection` is enabled, the selected items are stored in the `selectedItems` slot. In that case, you may not need to supply a `ButtonClickScript`.

protoTable

This proto is used to create a simple one-column table of text. Each of the table items can be selected (highlighted) by tapping it. Figure 5-8 shows an example:

Figure 5-8 One-column table of text

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

The `protoTable` includes `ViewScrollUpScript` and `ViewScrollDownScript` methods to handle scrolling. However, a view based on `protoTable` won't receive these system messages directly. To support scrolling, your application base view (which typically receives these messages from the system) should pass them along to the `protoTable` view.

Slot descriptions

`viewBounds` — Set to the size and location where you want the table to appear.

`def` — The table definition frame. Initially, you should set this to `protoTableDef`, which is the proto frame. Then in the `ViewSetupFormScript` method, you can change individual items. An example of the `protoTableDef` frame is shown in "protoTableDef" (page 5-27).

`scrollAmount` — Optional. The table scrolls one row at a time when the user taps a scroll button. If you want it to scroll more rows at a time, specify the number of rows here.

`viewFormat` — Optional. The default setting is `vfFillWhite + vfPen(1) + vfFrameBlack`.

`currentSelection` — Contains a string that is the text of the currently selected cell. If multiple selections are allowed, this string is the text of the last cell selected.

`selectedCells` — An array of indexes of selected cells. These are indexes into the `def.tabValues` array.

`declareSelf` — Do not change. This slot is set by default to `'base`. This symbol identifies the view for scrolling and other internal purposes.

ViewSetupFormScript

table:`ViewSetupFormScript()`

Use this method to clone the table definition frame, `def`, if you want to change any of the values in the frame at run time.

For example:

```
ViewSetupFormScript: func()
   begin
   def := Clone (def);
   def.tabValues := ["foo", "bar", "baz", "qux", "4",
                     "5", "6", "7", "8", "9"];
   // tabWidths must be =< the view width-2
   def.tabWidths := self; LocalBox() .right -2;
   def.tabDown := 10;
   end,
```

SelectThisCell

table:`SelectThisCell(`*cell*`)`

This method is defined internally and is called when the user taps a cell in the table. If you want to be notified whenever the user taps a cell, you can override this method. However, you must call the inherited method before doing anything else in your own method. For example:

```
selectThisCell: func(viewTapped)
   begin
   // first you MUST call the inherited method
   inherited:selectThisCell(viewTapped);
   // here you can do your own things
   ...
   end,
```

cell The child view representing the cell that was tapped.

protoTableDef

This proto defines the format of the table. You use it by setting the `protoTable` slot `def` to `protoTableDef`. You change individual items in the `ViewSetupFormScript` method. See `protoTable` for details.

Here is an example of protoTableDef:

```
protoTableDef := {
tabAcross: 1,
tabDown: 0,
tabWidths: 50,
tabHeights: 0,
tabProtos: protoTableEntry,
tabValues: nil,
tabValueSlot: 'text,
tabSetup: func(childView, hIndex, vIndex)
   begin
   childView.hIndex := hIndex - 1;// Save for selection
   childView.vIndex := vIndex - 1;
   end,
tabUniqueSelection: true, //use false for mult. selection
indentX: 2,
}
```

Slot descriptions

`tabAcross`	The number of columns in the table. This must be set to one (1). Multicolumn tables are not supported by `protoTable`.
`tabDown`	The number of rows in the table.
`tabWidths`	An integer giving the width of the single table column, in pixels.
`tabHeights`	An integer giving the height of a row, in pixels (constant for all rows).

`tabProtos` Each row in the table is child view of the table. This slot holds either a reference to a template used to create the child views, or an array of references to templates. For the slots for the default, see "protoTableEntry" (page 5-29).

`tabValues` A value used as the value of each child view. Alternately, an array of values mapped to table cells.

`tabValueSlot` A symbol naming the slot in each child view where that child's view value (specified in `tabValues`) is stored. (Remember to quote the symbol, as in, `'text`.) For example, if the table consists of child views based on the `clParagraphView` class (the default), you would specify `'text` for this slot, since the value of a `clParagraphView` is stored in its `text` slot.

`tabUniqueSelection`
A Boolean value. Set to non-`nil` to select only a single cell. Set to `nil` to select multiple cells.

`indentX` Reserved for internal use. Do not change.

IMPORTANT

If you allow multiple cell selection, your program will fail unless you ensure that the `selectedCells` slot is in RAM, since the proto attempts to add to this array. To make sure the slot is in RAM, use the following code in the `ViewSetupFormScript` method:
`self.selectedCells:=Clone(selectedCells);` ▲

TabSetup

table:`TabSetup(`*view*`,` *column*`,` *row*`)`

This method is called before each of the child views is instantiated. It allows you to do special initialization operations to each child view before it is instantiated. If you choose to override this method, call the inherited method also: `inherited:?TabSetup(childView, hIndex, vIndex)`.

view A reference to the child view.

column The column number of the child within the table.

row The row number of the child within the table.

protoTableEntry

This proto controls how the text in each row of the table appears; for example text justification and type of text selection. You use it by setting the `tabProtos` slot to `protoTableEntry` in `protoTableDef`. You change individual items in the `ViewSetupFormScript` method. See `protoTable` and `protoTableDef` for details.

Here is a list of the important slots in `protoTableEntry`:

Slot descriptions

`viewClass`	`clTextView` is a read-only `clParagraphView`; it supports no tabs or multistyled text.
`viewFlags`	`vVisible + vClickable + vReadOnly`
`viewJustify`	`vjLeftH + vjCenterV + oneLineOnly`
`viewTransferMode`	`modeOr`
`text`	Holds the text shown in this view.

ViewClickScript

entry`:ViewClickScript()`

This method sets `currentSelection` in the parent view (the table) to the value of the `text` slot. It also sends the `SelectThisCell` message.

ViewHiliteScript

entry`:ViewHiliteScript()`

This method highlights itself.

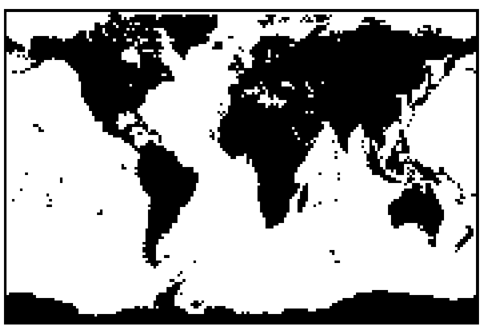
Map Pickers

These protos display various maps, let the user select a place, and return information about the location selected.

protoCountryPicker

This proto displays a picker from which a user can select a country, as shown in Figure 5-9.

Figure 5-9 Example of a country picker



You specify a `viewBounds`; the proto scales the picture to fit within it.

The picker behavior is automatic. On a tap, a picker listing nearby countries pops up. If the user selects a country, the `PickWorld` message is sent to your country picker view with one parameter, a frame containing information about the country picked.

Slot descriptions

`autoClose` Optional. Set to non-`nil` to force the `protoCountryPicker` view to close when the user

	chooses an item from a picker on the map. Set to `nil` to disable this autoclosing behavior. The default is `nil`.
`listLimit`	Optional. Set to the maximum number of items to be listed in one of the pickers that pops up when a user taps the map. The default value is 12.

PickWorld

picker`:PickWorld(`*info*`)`

This message is sent when the user picks a country.

info	A frame describing the country picked. The following example shows the information returned (from the Inspector output):

```
{name:"Guatemala", outgoing: s00, countryCode: 502,
latitude: 23363826, longitude: 401907529, continent:
'centralAmerica, currency: "Quetzal" },
```

protoProvincePicker

This proto is used to display a picker from which a user can select a Canadian province, as shown in Figure 5-10.

Figure 5-10 Example of a province picker

You specify a `viewBounds`, and the proto scales the picture to fit within it.

The picker behavior is automatic. On a tap, a picker listing nearby provinces pops up; if the user selects one, the `PickWorld` message is sent to your province picker view with one parameter, a frame containing information about the province picked.

Slot descriptions

`viewFlags` Optional. Should you override this slot, you must set `vClipping` because this proto draws outside of its bounds.

`autoClose` Optional. Set to non-`nil` to force the `protoProvincePicker` view to close when the user chooses an item from a picker on the map. Set to `nil` to disable this autoclosing behavior. The default is `nil`.

`listLimit` Optional. Set to the maximum number of items to be listed in one of the pickers that pops up when a user taps the map. The default value is 12.

PickWorld

picker`:PickWorld(`*info*`)`

This message is sent when the user picks a province.

info A frame describing the province picked. The following is an example of the information returned (from the Inspector output):

```
{name: "Nova Scotia", latitude: 67357415,
longitude:442918502},
```

protoStatePicker

This proto is used to display a picker from which a user can select a U.S. state, as shown in Figure 5-11.

Figure 5-11 Example of a state picker

You specify a `viewBounds`, and the proto scales the picture to fit within it.

The picker behavior is automatic. On a tap, a picker listing nearby states pops up; if the user selects one, the `PickWorld` message is sent to your state picker view with one parameter, a frame containing information about the state picked.

Slot descriptions

`viewFlags`	Optional. Should you override this slot, you must set `vClipping` because this proto draws outside its bounds.
`autoClose`	Optional. Set to non-`nil` to force the `protoStatePicker` view to close when the user chooses an item from a picker on the map. Set to `nil` to disable this autoclosing behavior. The default is `nil`.
`listLimit`	Optional. Set to the maximum number of items to be listed in one of the pickers that pops up when a user taps the map. The default value is 12.

PickWorld

picker:`PickWorld(`*info*`)`

This message is sent when the user picks a state.

info A frame describing the state picked. Here's an example of the information returned (from the Inspector output):

```
{name: "Florida", latitude: 42502280,
longitude: 414583648},
```

protoWorldPicker

This proto is used to display a picker from which a user can select a continent, as shown in Figure 5-12.

Figure 5-12 Example of a world picker

You specify a `viewBounds` frame, and the proto scales the world map picture to fit within it.

The picker behavior is automatic. On a tap, a picker listing nearby continents pops up. If the user selects one, the `PickWorld` message is sent to your world picker view with one parameter, a frame containing information about the continent picked.

Slot descriptions

`autoClose` — Optional. Set to non-`nil` to force the `protoWorldPicker` view to close when the user chooses an item from a picker on the map. Set to `nil` to disable this autoclosing behavior. The default is `nil`.

`listLimit` — Optional. Set to the maximum number of items to be listed in one of the pickers that pops up when a user taps the map. The default value is 12.

PickWorld

picker`:PickWorld(`*info*`)`

This message is sent when the user picks a continent.

info — A frame describing the continent picked. Here's an example of the information returned (from the Inspector output):

```
{ name: "Europe", topLatitude: 104391566,
leftLongitude:499588209, bottomLatitude: 49213166,
rightLongitude: 59652323},
```

Text Pickers

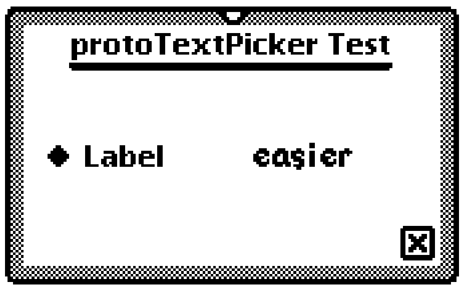
These protos allow the user to specify various kinds of information by picking text representations.

protoTextPicker

This proto displays a label picker with a text representation of an item. When the user taps the picker, the `PopIt` method, which allows a customized picker to be displayed, is executed. If the user picks an item, the `PickActionScript` is called. If you provide a customized picker, you must call `PickActionScript` with a correct `itemSelected` number.

Figure 5-13 shows an example of a slip that contains a `protoTextPicker` with its label preceded by `kPopChar`.

Figure 5-13 Example of a text picker



Slot descriptions

`label`	The constant `kPopChar &` is a string to be displayed as the picker label.
`indent`	You can specify an indent; otherwise, it's calculated for you.
`labelFont`	Optional. The font for the label; the default setting is `tsSize(10) + tsBold`.
`entryFont`	Optional. The font for the text picker line; the default setting is `editFont10`.

PopIt

picker`:PopIt(`*x*`)`

This method is called when the user taps the picker.

x	A value equal to (`indent` - 2).

PickActionScript

picker:`PickActionScript(`*item*`)`

This method is called after the user picks an item from the view displayed in `PopIt`.

item The item passed by `PopIt`.

PickCancelledScript

picker:`PickCancelledScript()`

This method is called if the pop-up view is cancelled by a tap outside it; if you don't supply this method, there is no default action.

TextSetup

picker:`TextSetup()`

This method returns a text string to be displayed in the entry part of the picker display.

protoDateTextPicker

This proto displays a label picker with a text representation of a date; for example "June 22, 1995". When the user taps the picker, the `protoDatePopup` is displayed, allowing the user to specify a different date. When the user taps the close box of the pop-up view, the text next to the label is updated with the new date. Figure 5-14 shows an example.

Figure 5-14 Example of a date text pop-up view

The `PopIt` and `TextSetup` methods are defined internally; you shouldn't need to override them.

The `protoDateTextPicker` uses the `protoTextPicker` as its proto; `protoTextPicker` is based on a view of the `clView` class.

Slot descriptions

`label`	A string to be displayed as the picker label.
`labelFont`	Optional. The font for the label; the default setting is `tsSize(10) + tsBold`.
`entryFont`	Optional. The font for the text picker line; the default setting is `editFont10`.
`date`	An initial date to display (as returned by the `Time` function). If you don't specify a date, the current date appears by default. This slot is also updated with the new date when the user closes the pop-up view.
`longFormat`	A symbol specifying the format in which to display the date; the default is `'yearMonthDayStrSpec`. See Chapter 17, "Localizing Newton Applications

	Reference," Table 17-1 (page 17-4), for a complete list of symbols for `longFormat`.
`shortFormat`	A symbol specifying the format in which to display the date. Use `shortFormat` only if you have a `nil` value for `longFormat`. See Chapter 17, "Localizing Newton Applications Reference," Table 17-2 (page 17-6), for a complete list of symbols for `shortFormat`.

Notes

Both `longFormat` and `shortFormat` must be present if you plan to use `shortFormat`. If you use `shortFormat`, `longFormat` must be set to `nil`.

If you implement `PickActionScript`, the parameter `newDate` is an array containing a single element of integer; it's the selected date in terms of minutes passed since midnight, 1/1/1904.

The slot `date` always contains the selected date (in terms of minutes passed since midnight, 1/1/1904 12:00). ◆

PickActionScript

picker`:PickActionScript(`*newDate*`)`

This method is called when the user taps the close box of the pop-up view. If you don't supply this method, there is no default action. If no item is selected because the user taps outside the list, the `PickCancelledScript` method is called instead.

newDate	The new date selected by the user.

PickCancelledScript

picker`:PickCancelledScript()`

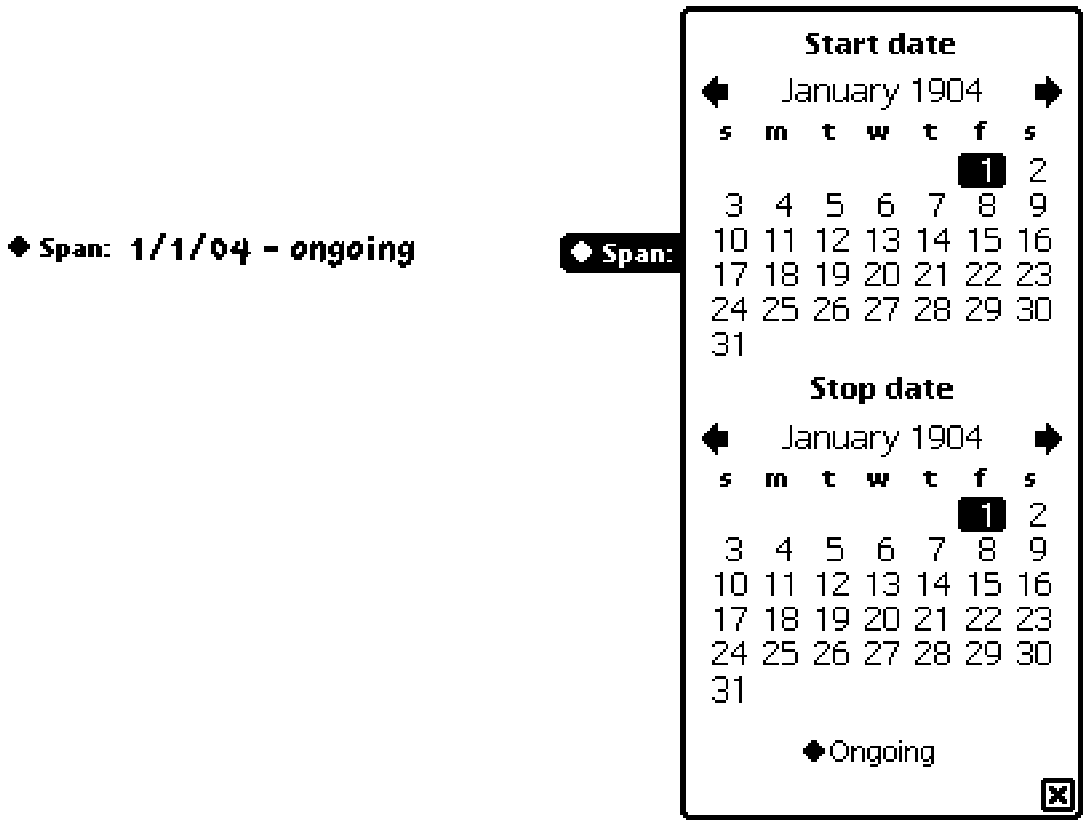
This method is called if the pop-up view is cancelled by a tap outside it; if you don't supply this method, there is no default action.

protoDateDurationTextPicker

This proto displays a label picker with a text representation of a date range; for instance "January 5, 1974 – February 7, 1975". When the user taps the picker, the `protoDateIntervalPopup` is displayed, allowing the user to specify a different range. When the user taps the close box of the pop-up view, the text next to the label is updated with the new date range.

Figure 5-15 shows an example of a `protoDateDurationTextPicker` with slot `shortFormat = 'numericDateStrSpec`. Notice the label is proceeded by `kPopChar`.

Figure 5-15 Example of date picker before and after it is tapped



The `PopIt` and `TextSetup` methods are defined internally; you shouldn't need to override them.

The `protoDateDurationTextPicker` uses the `protoTextPicker` as its proto; `protoTextPicker` is based on a view of the `clView` class.

Slot descriptions

`label`	A string to be displayed as the picker label.
`labelFont`	Optional. Font used to display the label; the default is `ROM_fontsystem10bold`.
`entryFont`	Optional. Font used to display the picked entry; the default is `10243 (= editFont10 ?).`
`indent`	Optional. If not supplied, `protoDateDurationTextPicker` calculates the indent based on the length of label.
`startTime`	An initial start date to display (as returned by the `Time` function).
`stopTime`	An initial end date to display (as returned by the `Time` function).
`longFormat`	A symbol specifying the format in which to display the time; the default is `'yearMonthDayStrSpec`.
`shortFormat`	A symbol specifying the format in which to display the time; the default is `nil`.

Note

Both `longFormat` and `shortFormat` must be present if you plan to use `shortFormat`. If you use `shortFormat`, `longFormat` must be set to `nil`.

You can provide a value for either a `longFormat` slot or a `shortFormat` slot, but not both, to specify the format in which to display the date range. ◆

PickActionScript

picker`:PickActionScript(`*startTime*, *stopTime*`)`

This method is called when the user taps the pop-up's close box. If you don't supply this method, there is no default action. If no item is selected because the user taps outside the list, the `PickCancelledScript` method is called instead.

startTime	The new starting time selected by the user.
stopTime	The new ending time selected by the user.

PickCancelledScript

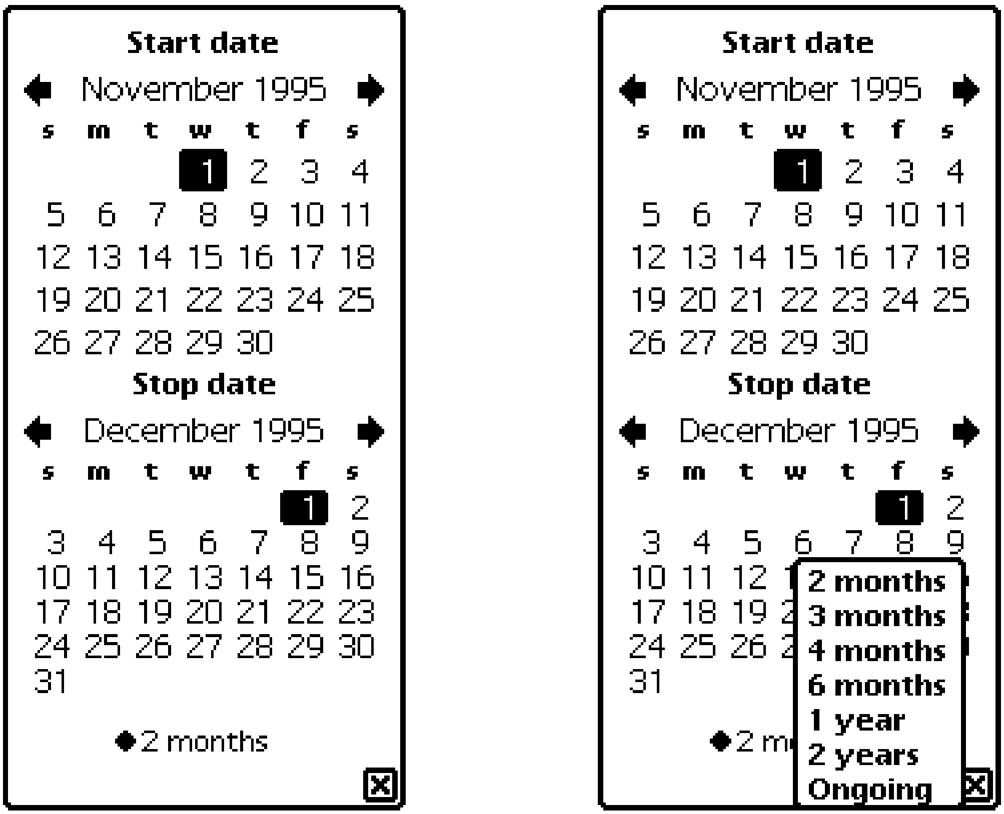
picker:`PickCancelledScript()`

This method is called if the pop-up view is cancelled by a tap outside it; if you don't supply this method, there is no default action.

protoRepeatDateDurationTextPicker

This proto displays a label picker with a text representation of a date range; for example, "January 5, 1974 - February 7, 1975". When the user taps the picker, the `protoDateIntervalPopup` is displayed, allowing the user to specify a different range. When the user taps the close box, the text next to the label is updated with the new date range. This looks essentially the same as Figure 5-15 (page 5-41). Figure 5-16 shows how the popup for this picker looks.

Figure 5-16 Example label picker with text representation



Unlike `protoDateDurationTextPicker`, `protoRepeatDateDurationTextPicker`'s `protoDateIntervalPopup`'s duration picker shows choices that are appropriate for the `repeatType` slot, and the duration displayed when the user taps a duration or stop date is given in units of the `repeatType`. For example, if the `repeatType` slot specifies monthly, the duration picker shows the choices for two months, three months, and so on, and the duration value string is in units of months. In contrast, a protoDateDurationTextPicker would always show choices for one week, two weeks, and so on and would display the duration value in units of weeks and days.

The ViewSetupFormScript, PopIt, TextSetup, and GetDuration methods are defined internally; you shouldn't need to override them. If you do override them, make sure to call the inherited method.

The protoRepeatDateDurationTextPicker uses protoDateDurationTextPicker as its proto; protoDateDurationTextPicker is based on protoTextPicker, which in turn is based on a view of the clView class.

Slot descriptions

label	A string to be displayed as the picker label.
labelFont	Optional. The font for the label; the default setting is tsSize(10) + tsBold.
entryFont	Optional. The font for the text picker line; the default setting is editFont10.
startTime	An initial start date to display (as returned by the Time function).
stopTime	An initial ending date to display (as returned by the Time function).

You can provide a value for either a longFormat slot or a shortFormat slot, but not both, to specify the format in which to display the date range.

longFormat	A symbol specifying the format in which to display the time; the default is nil.
shortFormat	A symbol specifying the format in which to display the time; the default is 'numericDateStrSpec.
repeatType	The repeatType slot contains one of the following constants that describe how often the meeting repeats: kDayofWeek, kWeekInMonth (1), kDateInMonth (2), kDateInYear(3), kPeriod(4), kNever(5), kWeekInYear(7).
mtgInfo	Used for repeating meetings and events. An immediate value containing packed repeating meeting information.

This slot is interpreted differently, depending on the value of the `repeatType` slot. For a complete list of values, see the description of the `mtgInfo` slot in Chapter 16, "Built-in Applications and System Data Reference," "Meeting Frames" (page 16-57).

PickActionScript

picker`:PickActionScript(`*startTime*`, `*stopTime*`)`

This method is called when the user taps the close box of the pop-up view. If you don't supply this method, there is no default action. If no item is selected because the user taps outside the list, the `PickCancelledScript` method is called instead.

startTime The new starting time selected by the user.

stopTime The new ending time selected by the user.

PickCancelledScript

picker`:PickCancelledScript()`

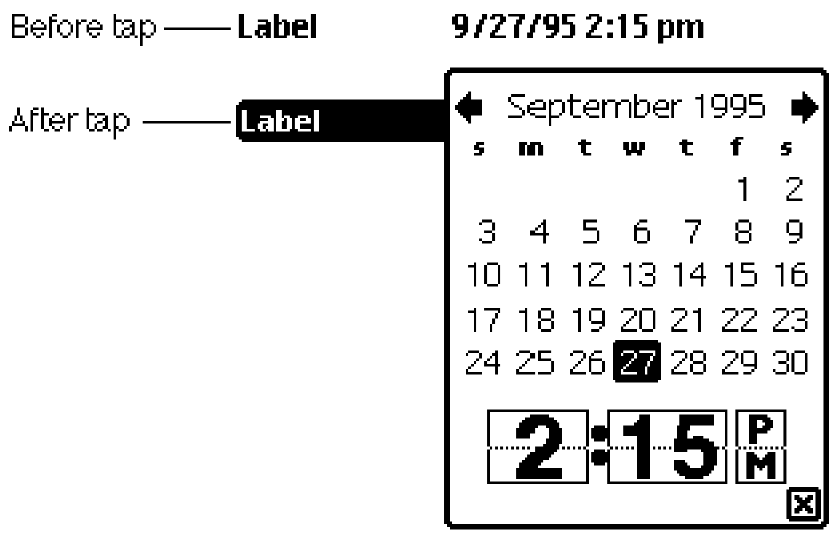
This method is called if the pop-up view is cancelled by a tap outside it; if you don't supply this method, there is no default action.

protoDateNTimeTextPicker

This proto displays a label picker with a text representation of a date and time; for example, "6/22/95 2:11 pm". When the user taps the picker, the `protoDateNTimePopup` is displayed, allowing the user to specify a different date and time. When the user taps the pop-up's close box, the text next to the label is updated with the new date and time.

Figure 5-17 shows an example of a date and time label picker before and after it is tapped.

Figure 5-17 Example of a date and time pop-up view



The `PopIt` and `TextSetup` methods are defined internally; you shouldn't need to override them.

The `protoDateNTimeTextPicker` uses the `protoTextPicker` as its proto; `protoTextPicker` is based on a view of the `clView` class.

Slot descriptions

`label`	Optional. A string to be displayed as the picker label.
`labelFont`	Optional. The font for the label; the default setting is `tsSize(10) + tsBold`.
`entryFont`	Optional. The font for the text picker line; the default setting is `editFont10`.
`date`	Optional. An initial date/time to display (as returned by the `Time` function). If you don't specify a date, the current date and time are used by default.
`format`	Optional. A symbol specifying the format in which to display the time; for example, "2:15 pm". The default

	value is `'shortTimeStrSpec`. See Chapter 17, "Localizing Newton Applications Reference," "Date and Time Format Specifications" (page 17-11) for information on specifying formats.
`longFormat`	Optional. A symbol specifying the format in which to display the date; for example, "September 27, 1995". The default is `nil`.
`shortFormat`	Optional. A symbol specifying the format in which to display the date; for example, "9/27/95". The default is `'numericDateStrSpec`.
`increment`	Optional. An integer representing the increment by which to change the time when the user taps the time picker portion of the pop-up view; a value of 15, for example, causes the time to change in 15 minute increments.

Note

You can provide a value for either a `longFormat` slot or a `shortFormat` slot, but not both, to specify the format in which to display the date and time. Because the default value of `longFormat` is `nil`, you can use `shortFormat` without providing a `longFormat` slot. ◆

PickActionScript

picker`:PickActionScript(`*newDate*`)`

This method is called when the user taps the pop-up's close box. If you don't supply this method, there is no default action. If no item is selected because the user taps outside the list, the `PickCancelledScript` method is called instead.

startTime	The new date and time selected by the user.

PickCancelledScript

picker:`PickCancelledScript()`

This method is called if the pop-up view is cancelled if the user taps outside it; if you don't supply this method, there is no default action.

protoTimeTextPicker

This proto displays a label picker with a text representation of a time; for example, "2:56 pm". When the user taps the picker, the `protoTimePopup` is displayed, allowing the user to specify a different time. When the user taps the pop-up's close box, the text next to the label is updated with the new time.

Figure 5-18 shows an example of a `protoTimeTextPicker` before and after it has been tapped.

Figure 5-18 Example of a label picker with a text representation of a time

The `PopIt` and `TextSetup` methods are defined internally; you shouldn't need to override them.

The `protoTimeTextPicker` uses the `protoTextPicker` as its proto; `protoTextPicker` is based on a view of the `clView` class.

Slot descriptions

`label`	A string to be displayed as the picker label.

`labelFont` Optional. The font for the label; the default setting is `tsSize(10) + tsBold`.

`entryFont` Optional. The font for the text picker line; the default setting is `editFont10`.

`indent` Optional. If not supplied, `protoDateDurationTextPicker` calculates the indent based on the length of label.

`time` The initial time (in number of minutes since midnight, 1/1/1904). This value is updated by the picker as the user picks a new value.

`format` Optional. A symbol specifying the format in which to display the time; the default is `'shortTimeStrSpec`. See Chapter 17, "Localizing Newton Applications Reference," "Date and Time Format Specifications" (page 17-11) for information on specifying formats.

`increment` Optional. An integer representing the increment by which to change the time when the user taps the pop-up view; the default value is 12, meaning that the time changes twelve minutes for each tap.

PickActionScript

picker`:PickActionScript(`*newTime*`)`

This method is called when the user taps the pop-up's close box. If you don't supply this method, there is no default action. If no item is selected because the user taps outside the pop-up view, the `PickCancelledScript` method is called instead.

newTime The new time selected by the user.

PickCancelledScript

picker`:PickCancelledScript()`

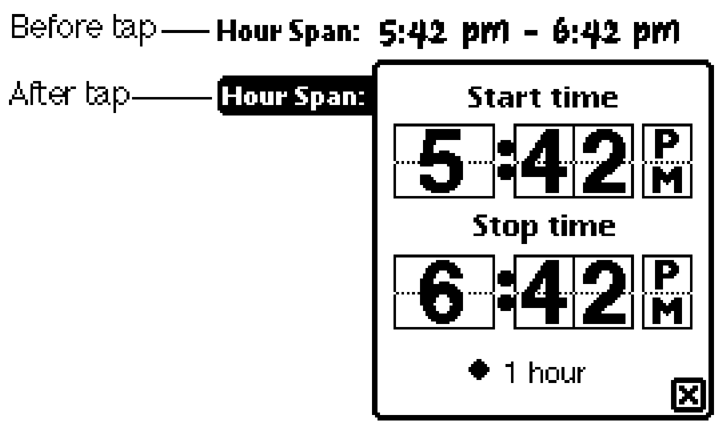
This method is called if the pop-up view is cancelled by a tap outside it; if you don't supply this method, there is no default action.

protoDurationTextPicker

This proto displays a label picker with a text representation of a time range; for example, “2:33 pm – 5:54 am”. When the user taps the picker, the `protoTimeIntervalPopup` is displayed, allowing the user to specify a different range. When the user taps the pop-up’s close box, the text next to the label is updated with the new time range.

Figure 5-19 shows an example `protoDurationTextPicker` before and after the user taps the picker.

Figure 5-19 Example label picker with a text representation of a time range



The `PopIt` and `TextSetup` methods are defined internally; you shouldn’t need to override them.

The `protoDurationTextPicker` uses the `protoTextPicker` as its proto; `protoTextPicker` is based on a view of the `clView` class.

Slot descriptions

`label` A string to be displayed as the picker label.

`labelFont`	Optional. The font for the label; the default setting is `tsSize(10) + tsBold`.
`entryFont`	Optional. The font for the text picker line; the default setting is `editFont10`.
`startTime`	An initial start time to display (as returned by the `Time` function).
`stopTime`	An initial ending time to display (as returned by the `Time` function).
`format`	A symbol specifying the format in which to display the time; the default is `'shortTimeStrSpec`. See Chapter 17, "Localizing Newton Applications Reference," "Date and Time Format Specifications" (page 17-11) for information on specifying formats.
`increment`	An integer representing the increment by which to change the time when the user taps the pop-up view; the default value is 1, meaning that the time changes one minute for each tap.

PickActionScript

picker:`PickActionScript(`*startTime*, *stopTime*`)`

This method is called when the user taps the pop-up's close box. If you don't supply this method, there is no default action. If no item is selected because the user taps outside the pop-up view, the `PickCancelledScript` method is called instead.

startTime	The start time selected by the user.
stopTime	The end time selected by the user.

PickCancelledScript

picker:`PickCancelledScript()`

This method is called if the pop-up view is cancelled if a user taps outside it; if you don't supply this method, there is no default action.

protoTimeDeltaTextPicker

This proto displays a label picker with a text representation of a time delta. When the user taps the picker, the `protoTimeDeltaPopup` is displayed, allowing the user to specify a new time delta. When the user taps the pop-up's close box, the text next to the label is updated with the new time delta.

Figure 5-20 shows an example of a `protoTimeDeltaTextPicker` before and after it is tapped.

Figure 5-20 Example of a label picker with a text representation of a time delta

The `PopIt` and `TextSetup` methods are defined internally; you shouldn't need to override them.

The `protoTimeDeltaTextPicker` uses the `protoTextPicker` as its proto; `protoTextPicker` is based on a view of the `clView` class.

Slot descriptions

`label`	The constant `kPopChar & is` a string to be displayed as the picker label.
`time`	An initial time (in number of minutes), which is then updated by the picker as a new value has been picked.
`labelFont`	Optional. The font for the label; the default setting is `tsSize(10) + tsBold`.
`entryFont`	Optional. The font for the text picker line; the default setting is `editFont10`.

`indent`	Optional. If not supplied, `protoDateDurationTextPicker` calculates the indent based on the length of label.
`minValue`	Optional. An integer specifying a minimum delta value.

PickActionScript

picker`:PickActionScript(`*newDuration*`)`

This method is called when the user taps the pop-up's close box. If you don't supply this method, there is no default action. If no item is selected because the user taps outside the list, the `PickCancelledScript` method is called instead.

newDuration	The number of minutes the user picked.

PickCancelledScript

picker`:PickCancelledScript()`

This method is called if the pop-up is cancelled by a tap outside it; if you don't supply this method, there is no default action.

protoMapTextPicker

This proto displays a label picker with a text representation of a country; for example, "Afghanistan". When the user taps the picker, a popup displays that allows the user to select a new country from an alphabetical list. When the user taps the pop-ups close box, the text next to the label is updated with the new country name.

Figure 5-21 shows an example of a `protoMapTextPicker` before and after it is tapped.

Figure 5-21 Example of a map text label picker

The `PopIt` and `TextSetup` methods are defined internally; you shouldn't need to override them.

The `protoMapTextPicker` uses the `protoTextPicker` as its proto; `protoTextPicker` is based on a view of the `clView` class.

Slot descriptions

`label`	A string to be displayed as the picker label.
`labelFont`	Optional. The font for the label; the default setting is `tsSize(10) + tsBold`.
`entryFont`	Optional. The font for the text picker line; the default setting is `editFont10`.
`indent`	Optional. If not supplied, the proto calculates it based on the length of label.

PickActionScript

picker`:PickActionScript(`*newName*`)`

This method is called when the user taps the pop-up's close box. If you don't supply this method, there is no default action. If no item is selected because the user taps outside the list, the `PickCancelledScript` method is called instead.

newName The new country name selected by the user.

PickCancelledScript

picker`:PickCancelledScript()`

This method is called if the pop-up view is cancelled by a tap outside it; if you don't supply this method, there is no default action.

protoCountryTextPicker

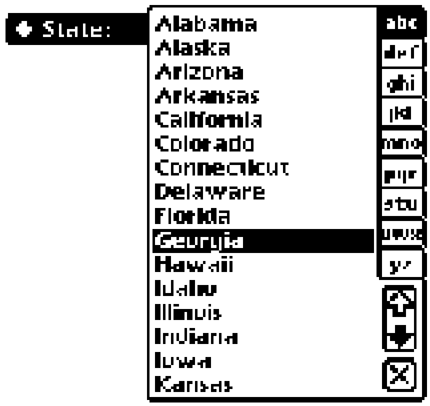
The `protoCountryTextPicker` is the same as `protoMapTextPicker` (which it uses as its proto).

protoUSstatesTextPicker

This proto displays a label picker with a text representation of a U.S. state; for example, "Ohio". When the user taps the picker, a popup displays that allows the user to select a new state from an alphabetical list. When the user taps the pop-up's close box, the text next to the label is updated with the new state name.

Figure 5-22 shows an example of `protoUSstatesTextPicker` before and after it has been tapped.

Figure 5-22 Example of a label picker with a text representation of a U.S. state



The `PopIt` and `TextSetup` methods are defined internally; you shouldn't need to override them.

The `protoUSstatesTextPicker` uses the `protoMapTextPicker` as its proto.

Slot descriptions

`label`	A string to be displayed as the picker label.
`labelFont`	Optional. The font for the label; the default setting is `tsSize(10) + tsBold`.
`entryFont`	Optional. The font for the text picker line; the default setting is `editFont10`.
`indent`	Optional. If not supplied, `protoDateDurationTextPicker` calculates the indent based on the length of label.
`params`	A frame with the following slot: `result` The default value is `'name`. You can change it to `'abbrev` in your `ViewSetupFormScript` if you want to get an abbreviated form of the name. For

example, give your `protoUSStatesTextPicker` the following `ViewSetupFormScript`:

```
func()
begin
        self.params := Clone(params) ;
        params.result := 'abbrev ;
        inherited:?viewSetupFormScript();
end
```

That will make the result an abbreviation and will change the picker label to an abbreviation as well.

PickActionScript

picker:`PickActionScript(`*newName*`)`

This method is called when the user taps the pop-up's close box. If you don't supply this method, there is no default action. If no item is selected because the user taps outside the list, the `PickCancelledScript` method is called instead.

newName The new state name selected by the user.

PickCancelledScript

picker:`PickCancelledScript()`

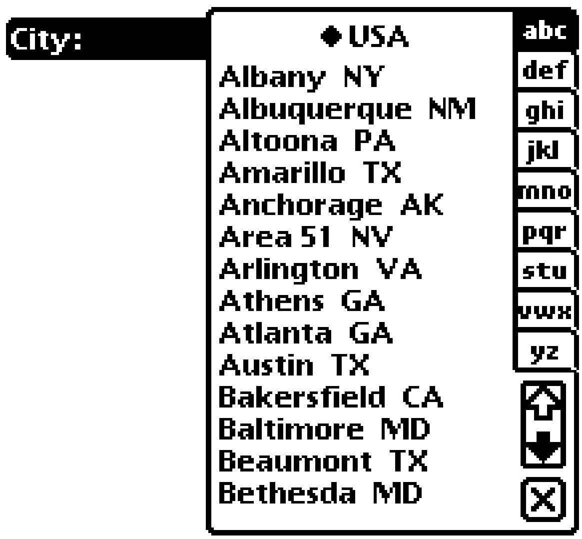
This method is called if the pop-up is cancelled by a tap outside it; if you don't supply this method, there is no default action.

protoCitiesTextPicker

This proto displays a label picker with a text representation of a city; for example, "Albany". When the user taps the picker, a popup displays that allows the user to select a new city from an alphabetical list. When the user taps the pop-up's close box, the text next to the label is updated with the new city name.

Figure 5-23 shows an example of a `protoCitiesTextPicker` before and after a tap is made.

Figure 5-23 Example of a city picker



The `PopIt` and `TextSetup` methods are defined internally; you shouldn't need to override them.

The `protoCitiesTextPicker` uses the `protoMapTextPicker` as its proto.

Slot descriptions

`label`	A string to be displayed as the picker label.
`labelFont`	Optional. The font for the label; the default setting is `tsSize(10) + tsBold`.
`entryFont`	Optional. The font for the text picker line; the default setting is `editFont10`.

`indent` Optional. If not supplied, `protoCitiesTextPicker` calculates it based on the length of label.

`params` A frame with slots that are used internally.

In order to choose a default city specify a default slot in the `params` frame of the form:

`default` Optional. Specifies a default city with a value of the form:
[*country-symbol*, *city-name*]

For example:

```
['Canada, "Calgary"]
```

You can find the appropriate symbol and name by using `GetCityEntry` to find the entry for the appropriate city, then using the `country` slot for the country symbol and the `name` slot for the city name. For example:

```
local c := GetCityEntry("Calgary")[0] ;
params.default := [c.countr, c.name] ;
```

PickActionScript

picker:`PickActionScript(`*newName*`)`

This method is called when the user taps the pop-up's close box. If you don't supply this method, there is no default action. If no item is selected because the user taps outside the list, the `PickCancelledScript` method is called instead.

newName The new city name selected by the user.

PickCancelledScript

picker:`PickCancelledScript()`

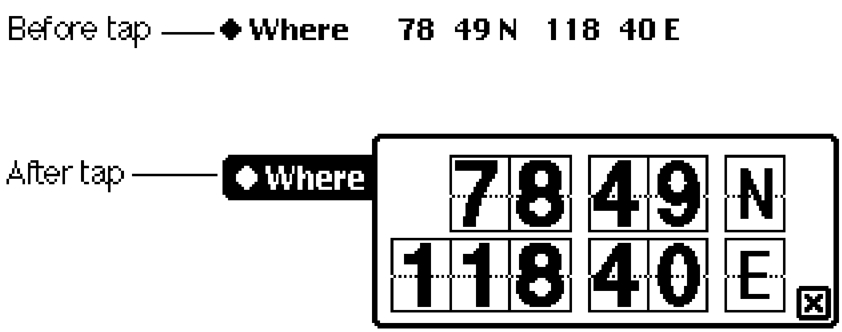
This method is called if the pop-up is cancelled when the user taps outside it; if you don't supply this method, there is no default action.

protoLongLatTextPicker

This proto displays a label picker with a text representation of longitude and latitude values. When the user taps the picker, the `longLatPicker` is displayed, allowing the user to select new values for longitude and latitude. When the user taps the pop-up's close box, the text next to the label is updated with the new values.

Figure 5-24 shows an example of `protoLongLatTextPicker` before and after it has been tapped.

Figure 5-24 Example of a text representation of longitude and latitude values



The `PopIt` and `TextSetup` methods are defined internally; you shouldn't need to override them.

The `protoLongLatTextPicker` uses the `protoTextPicker` as its proto; `protoTextPicker` is based on a view of the `clView` class.

Slot descriptions

`label`	The constant `kPopChar` followed by a string to be displayed as the picker label; "Where" is the default.
`latitude`	An integer specifying the latitude to display initially. See Chapter 19, "Built-in Applications and System Data,""Using Longitude and Latitude Values"

	(page 19-30) in *Newton Programmer's Guide* for information calculating this value.
`longitude`	An integer specifying the longitude to display initially.
`labelFont`	Optional. The font for the label; the default setting is `tsSize(10) + tsBold`.
`entryFont`	Optional. The font for the text picker line; the default setting is `editFont10`.
`indent`	Optional. The distance, in pixels, to indent the picker from the beginning of the line (the beginning of the text label). If you don't include this slot, the picker is placed 6 pixels to the right of the text label by default.
`worldClock`	A Boolean, must be non-`nil`.

PickActionScript

picker`:PickActionScript(`*long*, *lat*`)`

This method is called when an item is selected from the pop-up view. If you don't supply this method, there is no default action. If no item is selected because the user taps outside the pop-up view, the `PickCancelledScript` method is called instead.

long	The new longitude selected by the user.
lat	The new latitude selected by the user.

When the user picks new longitude or latitude value, the slots `longitude` and `latitude` are automatically updated.

PickCancelledScript

picker`:PickCancelledScript()`

This method is called if the pop-up view is cancelled when the user taps outside it; if you don't supply this method, there is no default action.

Date, Time, and Location Pop-up Views

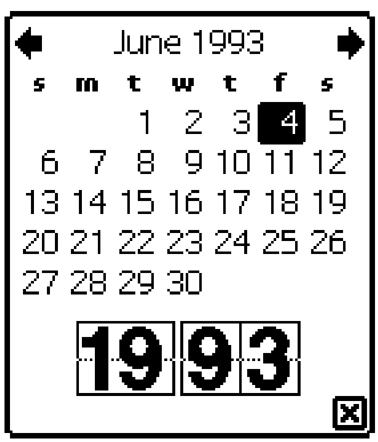
These protos let the user specify dates, times, and locations using graphical pop-up views.

protoDatePopup

This proto lets the user choose a single date. To provide selection of multiple dates, use the `protoMultiDatePopup` proto. The user confirms the selected date by tapping the close box; tapping outside the pop-up view cancels the pop-up view.

Figure 5-25 shows the result of a single-date selection.

Figure 5-25 Example of a single date selection



New

popup:`New(`*initialDate*`,` *bbox*`,` *callbackContext*`)`

This method is called to open the pop-up view.

initialDate	An array containing one element, an integer representing the initial date to display as selected (as returned by the `Time` function).
bbox	A bounding box for the pop-up view. This box is only suggested; generally, you would use `:GlobalBox()`.
callbackContext	The name of the view to which callback messages should be sent. Specify `self` if you define these methods in the pop-up view

PickActionScript

callbackContext:`PickActionScript(`*selectedDate*`)`

This method is called when the user taps the close box.

selectedDate	An array containing a single date.

PickCancelledScript

callbackContext:`PickCancelledScript()`

This method is called if the pop-up view is cancelled by tapping outside it.

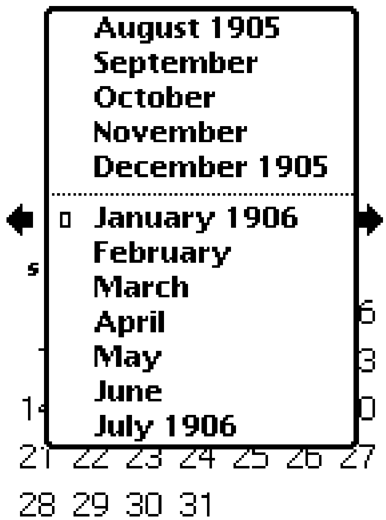
protoDatePicker

This proto facilitates the selection of a date. Use this proto when the desired date is likely to be relatively close to the current date, because it's not easy to change the year quickly. Figure 5-26 shows a date picker.

Figure 5-26 Example of a date picker

Tapping either arrow scrolls to the prior/next month; tapping a day selects the day; tapping the spaces before the first or after the last day of a month selects the appropriate day in the prior/next month; and tapping the month/year banner at the top displays a pop-up menu to change the month. The pop-up menu appears in Figure 5-27.

Figure 5-27 Example of a pop-up menu



The following slots and methods are used internally:

`MonthChangedScript`, `SetTitle`, `ViewSetupDoneScript`

These are listed so that you don't inadvertently override them.

Slot description

`selectedDates` An array containing one element, an integer representing the selected date (as returned by the `Time` function). You can supply an initial date to display here as well; if you don't, the current date is used.

To change the selected date programmatically, supply the new date in this slot and call the `Refresh` method.

DateChanged

picker:`DateChanged(`*array*`)`

This method is called when a date is selected, to give you a chance to take some action. The return value is ignored.

array An array containing a single element, the selected date.

Refresh

picker:`Refresh()`

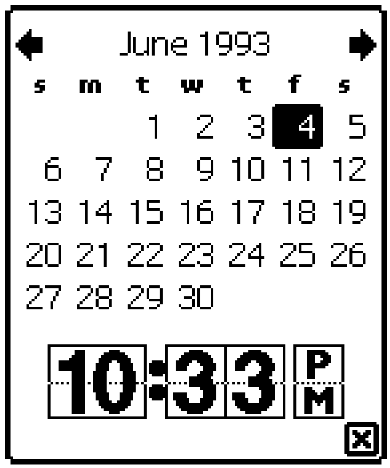
To change the selected date programmatically, supply a new date in the `selectedDates` slot and call this method to update the view.

protoDateNTimePopup

This proto lets the user choose a single date and time. The user confirms the selection by tapping the close box; tapping outside the pop-up view cancels it.

Figure 5-28 shows the result of a date and time selection.

Figure 5-28 Example of a single date and time selection



New

popup`:New(`*dateNTime*`,` *increment*`,` *bbox*`,` *callbackContext*`)`

This method is called to open the pop-up view.

dateNTime — An array containing one element, an integer, representing the initial date and time to display as selected (as returned by the `Time` function).

increment — An increment value that determines the granularity for the pop-up view. The value "1" specifies one minute; try "15", "30", and "60".

bbox — A bounding box for the pop-up view. This box is only suggested; generally, you would use `:GlobalBox()`.

callbackContext — The name of the view to which callback messages should be sent. Specify `self` if you define these methods in the pop-up view.

NewTime

callbackContext:`NewTime(`*dateNTime*`)`

This method is called whenever the time is changed.

dateNTime The new date and time.

PickActionScript

callbackContext:`PickActionScript(`*dateNTime*`)`

This method is called when the user taps the close box.

dateNTime The selected date and time.

PickCancelledScript

callbackContext:`PickCancelledScript()`

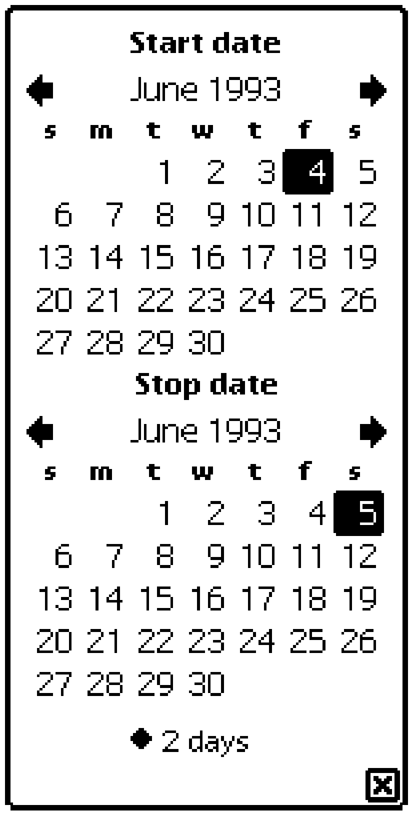
This method is called if the pop-up view is cancelled when the user taps outside it; if you don't supply this method, there is no default action.

protoDateIntervalPopup

This proto lets the user specify an interval of dates by selecting a start and stop date. The user confirms the selection by tapping the close box; tapping outside the pop-up view cancels it.

Figure 5-29 shows the result of selecting a start and stop date.

Figure 5-29 Example of a date interval pop-up view



The `protoDateIntervalPopup` is based on the `protoGeneralPopup` proto. It has the following two child views declared in itself:

- `start` uses the `protoDatePicker` proto and implements the starting date section of the pop-up view.
- `stop` uses the `protoDatePicker` proto and implements the ending date section of the pop-up view.

New

popup`:New(`*initialDates*`,` *bbox*`,` *callbackContext*`)`

This method is called to open the pop-up view.

initialDates	An array with two values (as returned by the `Time` function) specifying the initial range of dates to display as selected.
bbox	A bounding box for the pop-up view. This box is only suggested; generally you would use `:GlobalBox().`
callbackContext	The name of the view to which callback messages should be sent. Specify `self` if you define these methods in the pop-up view.

NewTime

callbackContext`:NewTime(`*startDate*`,` *stopOrMax*`)`

This method is called each time the user changes the selection.

startDate	The new start date.
stopOrMax	The new stop date, or the maximum time if ongoing. Note that the maximum time is defined as the constant `kMaximumTime := 0x1FFFFFFF`.

PickActionScript

callbackContext`:PickActionScript(`*startDate*`,` *stopOrMax*`)`

This method is called when the user taps the close box.

startDate	The new start date.
stopOrMax	The new stop date, or the maximum time if ongoing. Note that the maximum time is defined as the constant `kMaximumTime := 0x1FFFFFFF.`

PickCancelledScript

callbackContext`:PickCancelledScript()`

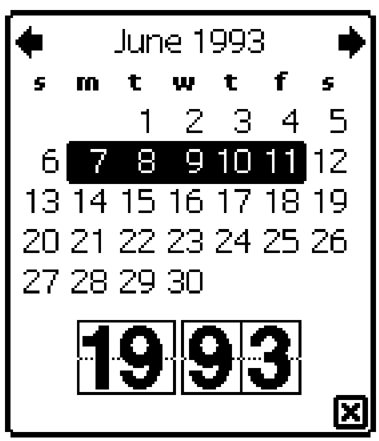
This method is called if the pop-up view is cancelled by tapping outside it.

protoMultiDatePopup

This proto lets the user specify a range of dates. To select a single date, use the `protoDatePopup`. The user confirms the selected range by tapping the close box; tapping outside the pop-up view cancels it.

Figure 5-30 shows the result of selecting a range of dates.

Figure 5-30 Example of a multidate pop-up view



New

popup`:New(`*initialDates*`,` *bbox*`,` *callbackContext*`)`

This method is called to open the pop-up view.

initialDates An array specifying a range of dates to display as selected. These dates have to be in sequence. For example, one day after another (for example, Tuesday, Wednesday, and Thursday) or the same day of the week (for example, first, second, and third Tuesday of the month).

bbox A bounding box for the pop-up view. This box is only suggested; generally, you would use `:GlobalBox()`.

callbackContext The name of the view to which callback messages should be sent. Specify `self` if you define these methods in the pop-up view.

PickActionScript

callbackContext`:PickActionScript(`*selectedDates*`)`

This method is called when the user taps the close box.

selectedDates An array containing the selected range of dates.

PickCancelledScript

callbackContext`:PickCancelledScript()`

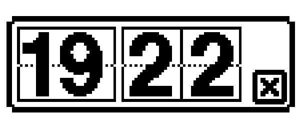
This method is called if the pop-up view is cancelled when the user taps outside it; if you don't supply this method, there is no default action.

protoYearPopup

This proto lets the user specify a year. The user confirms the selected range by tapping the close box; tapping outside the pop-up view cancels it.

Figure 5-31 shows the result of selecting a year.

Figure 5-31 Example of a year pop-up view



New

popup`:New(`*initialYear*`, `*bbox*`, `*callbackContext*`)`

This method is called to open the pop-up view.

initialYear	The year to display initially, specified as an integer (for example, "1995").
bbox	A bounding box for the pop-up view. This box is only suggested; generally, you would use `:GlobalBox()`.
callbackContext	The name of the view to which callback messages should be sent. Specify `self` if you define these methods in the pop-up view.

NewYear

callbackContext`:NewYear(`*year*`)`

This method is called each time the user changes the selection.

year	The new year, specified as a year.

DoneYear

callbackContext`:DoneYear(`*year*`)`

This method is called when the user taps the close box.

year	The selected year, specified as a year.

PickCancelledScript

callbackContext`:PickCancelledScript()`

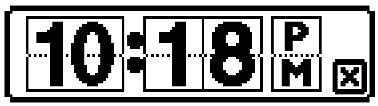
This method is called if the pop-up view is cancelled when the user taps outside of it; if you don't supply this method, there is no default action.

protoTimePopup

This proto permits setting a time with a digital clock. The user confirms the selection by tapping the close box; tapping outside the pop-up view cancels it.

Figure 5-32 shows how the digital clock appears.

Figure 5-32 Example of a time pop-up view



New

popup`:New(`*time*`,` *increment*`,` *bbox*`,` *callbackContext*`)`

This method is called to open the pop-up view.

time An array containing one element, an integer representing the initial time to display as selected (as returned by the `Time` function).

increment An increment for the pop-up view that determines the granularity for the pop-up view. The value "1" specifies one minute; try "15", "30", and "60".

bbox A bounding box for the pop-up view. This box is only suggested; generally, you would use `:GlobalBox().`

callbackContext The name of the view to which callback messages should be sent. Specify `self` if you define these methods in the pop-up view.

NewTime

callbackContext`:NewTime(`*time*`)`

This method is called whenever the time is changed.

time The new time.

PickActionScript

callbackContext`:PickActionScript(`*time*`)`

This method is called when the user taps the close box.

time The selected time.

PickCancelledScript

callbackContext`:PickCancelledScript()`

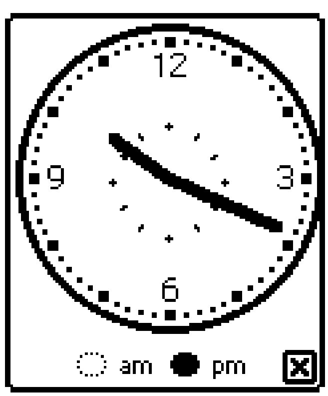
This method is called if the pop-up view is cancelled by tapping outside it.This method is called if the pop-up view is cancelled by tapping outside it.

protoAnalogTimePopup

This proto permits setting a time with an analog clock. The user confirms the selection by tapping the close box; tapping outside the pop-up view cancels it.

Figure 5-33 shows how the analog clock appears.

Figure 5-33 Example of an analog time pop-up view



New

popup:`New(`*time*`,` *increment*`,` *bbox*`,` *callbackContext*`)`

This method is called to open the pop-up view.

time	An array containing one element, an integer representing the initial time to display as selected (as returned by the `Time` function).
increment	An increment for the pop-up view that determines the granularity for it. The value "1" specifies one minute; try "15", "30", and "60".
bbox	A bounding box for the pop-up view. This box is only suggested; generally, you would use `:GlobalBox().`
callbackContext	The name of the view to which callback messages should be sent. Specify `self` if you define these methods in the pop-up view.

NewTime

callbackContext:`NewTime(`*time*`)`

This method is called whenever the time is changed.

time	The new time.

PickActionScript

callbackContext:`PickActionScript(`*time*`)`

This method is called when the user taps the close box.

time	The selected time.

PickCancelledScript

callbackContext:`PickCancelledScript()`

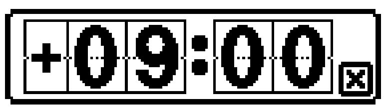
This method is called if the pop-up view is cancelled when the user taps outside it; if you don't supply this method, there is no default action.

protoTimeDeltaPopup

This proto lets the user choose a time period, or delta. The user confirms the selection by tapping the close box; tapping outside the pop-up view cancels it.

Figure 5-34 illustrates this time choice option.

Figure 5-34 Example of a time delta pop-up view



New

popup`:New(`*initialDelta*`,` *params*`,` *bbox*`,` *callbackContext*`)`

This method is called to open the pop-up view.

initialDelta
: An integer representing the initial delta time. A value of "1" specifies one minute, and the sign of the value specifies whether the delta is positive (+) or negative (–).

params
: A frame containing the following slots:

`increment`	An increment value that determines the granularity for the pop-up view. The value "1" specifies one minute; try "15", "30", and "60".
`minValue`	Optional. A minimum delta value.
`maxValue`	Optional. A maximum delta value.

bbox
: A bounding box for the pop-up view. This box is only suggested; generally, you would use `:GlobalBox()`.

callbackContext
: The name of the view to which callback messages should be sent. Specify `self` if you define these methods in the pop-up view.

PickActionScript

callbackContext`:PickActionScript(`*delta*`)`

This method is called when the user taps the close box.

delta The selected delta time.

PickCancelledScript

callbackContext`:PickCancelledScript()`

This method is called if the pop-up view is cancelled by tapping outside it.

protoTimeIntervalPopup

This proto lets the user choose a time interval by specifying a start and stop time. The user confirms the selection by tapping the close box; tapping outside the pop-up view cancels it.

Figure 5-35 illustrates a time interval selection.

Figure 5-35 Example of a time interval pop-up view

New

popup`:New(`*initialTimes*`,` *increment*`,` *bbox*`,` *callbackContext*`)`

This method is called to open the pop-up view.

initialTimes	An array with two values (as returned by the `Time` function) specifying the initial range of start and stop times.
increment	An increment value that determines the granularity for the pop-up view. The value "1" specifies one minute; try "15", "30", and "60".
bbox	A bounding box for the pop-up view. This box is only suggested; generally, you would use `:GlobalBox().`
callbackContext	The name of the view to which callback messages should be sent. Specify `self` if you define these methods in the pop-up view.

PickActionScript

callbackContext`:PickActionScript(`*startTime*`,` *stopTime*`)`

This method is called when the user taps the close box.

startTime	The selected start time.
stopTime	The selected stop time.

PickCancelledScript

callbackContext`:PickCancelledScript()`

This method is called if the pop-up view is cancelled by tapping outside it.

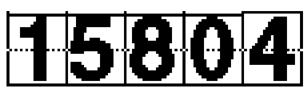
Number Pickers

This section describes the protos used to display pickers with numbers.

protoNumberPicker

This proto is used to display a picker from which the user can select a number. Figure 5-36 shows an example.

Figure 5-36 Example of a number picker



The following slots are of interest:

Slot descriptions

`minValue` Required. The minimum value in the list.

`maxValue` Required. The maximum value in the list.

`value` Required. The initial and currently selected value.

`showLeadingZeros`
Optional. Set this slot to non-`nil` to show leading zeros; for example, to show "007" with the two leading zeros.

This proto is based on a view of the `clPictureView` class. It has one child view, for each digit in the number; these views implement the picker functionality of the proto.

PrepareForClick

picker:`PrepareForClick()`

This method is called before a click on an individual digit is processed. The `value` slot is updated accordingly.

ClickDone

picker:`ClickDone()`

This method is called after a click on an individual digit is processed. The `value` slot is updated accordingly. You can override this method and check the `value` slot to determine the selected value.

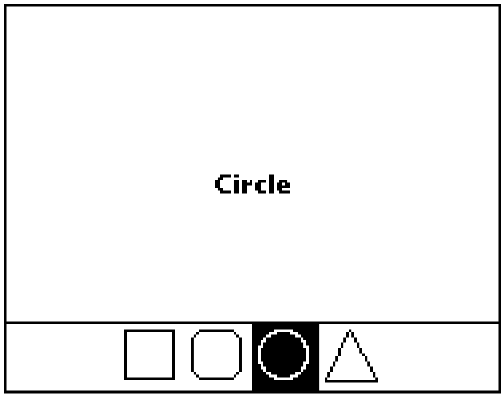
Picture Picker

This section describes the protos used to create a picture as a picker.

protoPictIndexer

This proto is used to create a view with a horizontal array of pictures, one of which the user can tap. When the user taps a picture, it is highlighted, and the system sends the `IndexClickScript` to signal which picture was selected. Figure 5-37 shows a typical array of pictures from which a user might make a selection.

Figure 5-37 Example of an indexed array of pictures



The following methods are defined internally: `ViewSetupDoneScript`, `ViewDrawScript`, `ViewClickScript`, `Hilite`, `Unhilite`, and `TrackPictHilite`. If you need to use one of these methods, be sure to call the inherited method also; for example,

```
inherited:?ViewSetupDoneScript()
```

or the proto may not work as expected.

The `protoPictIndexer` is based on a view of the `clPictureView` class.

Slot descriptions

`viewBounds`	Set to the size and location in which you want the view to appear.
`viewJustify`	Optional. The default setting is `vjCenterH + vjCenterV + vjParentFullH + vjParentBottomV`.
`viewFormat`	Optional. The default setting is `vfFillWhite`.
`icon`	The bitmap serving as the picture. This picture should be a single bitmap containing multiple objects or symbols that are all of the same width and arranged next to each other in a vertical row.

`iconBBox` — A bounds frame giving the bounds of the bitmap within the view. (The view can be bigger than the bitmap.) The width is the only important dimension calculated from the bounds frame. The width is used to calculate the size of the active rectangle; that is, the rectangle used to differentiate and highlight each of the objects in the bitmap. The width of the bitmap is divided into equal rectangles (width÷`numIndices`) that extend the full height of the view.

`numIndices` — The number of objects or symbols in the bitmap.

`curIndex` — This slot is set to the index of the currently selected item in the bitmap. Note that the first item has an index of zero. This slot must initially be set to an integer.

IndexClickScript

picker`:IndexClickScript(`*index*`)`

This method is called whenever the user taps the bitmap.

index — The index of the item that was chosen from the pop-up array. Note that the first item has an index of zero.

Here is an example of a template using `IndexClickScript`:

```
indexView := {...
_proto: protopictindexer,
viewBounds: {top: -25, left: 0, right: 0, bottom: 0},
viewJustify: vjCenterH+vjCenterV+vjParentFullH+
             vjParentBottomV,
viewFormat: vfFillWhite+vfPen(1)+vfFrameBlack,
icon: shapesBitmap, //square, roundrect, circle, triangle
iconBBox: {top: 0, left: 0, right: 100, bottom: 0},
numIndices: 4,
IndexClickScript: func(currIndex)
   begin
   SetValue(theText, 'text, shapeNameArray[currIndex]);
   end,
```

```
// set highlight to first item on entry
curIndex: 0,
shapeNameArray: ‘["Square", "Oval", "Circle", "Triangle"]
...}
```

Overview Protos

The protos in this section are used to create overviews of data; they include some protos specifically designed to display names from the Names soup.

protoOverview

This proto provides a framework for doing an overview view of data in an application. Each item in the overview has one line; the user can scroll the list and pick individual items or multiple items in the list.

Each entry in the list is a set of shapes is created by the client application. Figure 5-38 is an example of a `protoOverview` list.

Figure 5-38 Example of an overview list

Note that the `ViewClickScript` and `ViewDrawScript` methods are used internally in the `protoOverview` and should not be overridden.

Slot descriptions

`autoDeselect` Optional. If you set this to `true`, the item the user picks in the overview does not remain highlighted when the pen leaves it. Otherwise, when the pen leaves the item, it remains highlighted.

`viewBounds` Set to the size and location where you want the overview to appear.

`viewFlags` The default is `vVisible + vApplication + vClickable`.

`cursor` Optional. You probably need this if you want to use `protoOverview` directly, rather than using `protoSoupOverview`. This contains a cursor-like object that performs the same functions as a soup cursor. See "Using protoOverview" (page 6-24) in *Newton Programmer's Guide* for details.

`lineHeight` Optional. The default is 32, which specifies the height in pixels of each item in the overview.

`selectIndent` Optional. Specifies the left margin within which selection highlighting and behavior occur. If an item is tapped within this margin, the default `HitItem` method calls the `SelectItem` method with the item index. The default, if you don't supply this slot, is 18.

`viewFont` Optional. The default setting is `systemFont10Bold`.

`nothingCheckable`
Optional. If you don't want checkboxes at all, set this slot to non-`nil`. None of the list items will be indented and the vertical line down the left side of the list will be removed.

SetupAbstracts

overview`:SetupAbstracts(`*cursor*`)`

A method that should be called from the `ViewSetupChildrenScript` as the instantiator.

cursor A cursor or cursor-like object.

Abstract

overview`:Abstract(`*item*`, `*bounds*`)`

This method should return a shape or shape list representing an item in the overview. It is passed two parameters, the first an item obtained from the cursor-like object passed to `SetupAbstracts`, the second a bounds frame within which the returned shape should be placed.

The *bounds* value is not automatically offset by the `selectIndent` value. Therefore, you should use `selectIndent` rather than *bounds*`.left` to make certain that the shape returned fits in the frame.

item The item returned from the cursor-like object that was passed to the `SetupAbstracts` method.

bounds A bounds frame within which the returned shape should be placed.

CheckState

overview`:CheckState(`*entry*`)`

If the `nothingCheckable` slot is set to `nil`, the `SetUpAbstracts` method calls `CheckState` for each entry. You can override `CheckState` to return one of the following values:

Value	Meaning
`'notCheckable`	Cannot be checked; don't put a checkbox here.
`nil`	Can be checked, but isn't.
`true`	Can be checked, and is.

`CheckState` returns `nil` by default (checkable, but not checked). If checkboxes are specified, they are centered vertically based on the value in the `lineHeight` slot.

entry A soup entry.

HitItem

overview`:HitItem(`*hitIndex*`, `*xcoord*`, `*ycoord*`)`

A method that is called when an item is tapped. The default method returns non-`nil` if it handled the tap; that is, if it determined it should select the item (if the tap was within the `selectIndent` margin). In general, you should first call `inherited:?HitItem`, and handle the tap yourself only if the inherited method returns `nil`.

hitIndex The index to the item in the list (the first one being zero).

xcoord The X coordinate of the tap, relative to the left edge of the item that was tapped.

ycoord The Y coordinate of the tap, relative to the top edge of the item that was tapped.

Note that *hitIndex* is relative to the displayed items, not the total items. You need to track what the real "top" index is, as shown in the following example:

```
func(hitIndex, xcoord, ycoord)
begin
   if xcoord < selectIndent then
   inherited:HitItem(hitIndex, xcoord, ycoord) ;
   else begin
      hitIndex := hitIndex + saveIndex;
      print("hit item: " & hitIndex) ;
      :Dirty(); // refresh the view
   end ;
end
```

Notice that this code assumes you have a value `saveIndex` that can be added to the the `hitIndex` to find the index of the actual item.

IsSelected

overview:`IsSelected(`*item*`)`

item	The entry that the user tapped.

Return `true` if the item is selected (the checkbox is checked in the overview). Note that selected is different from highlighted or hit.

Scroller

overview:`Scroller(`*numItems*`)`

Scrolls the contents of the overview. The default method does nothing. If overridden, `Scroller` should cause the `SetupAbstracts` method to be called again, for example, by calling `RedoChildren`.

numItems	The number of items to scroll; a negative value means “scroll upwards.”

SelectItem

overview:`SelectItem(`*hitIndex*`)`

`SelectItem` is called each time the checkbox for an item is tapped. You must provide this method if `SelectIndent` is greater than 0. It should implement a way of remembering selected items, so the user can file or route the selected items at a later time and also perform whatever record keeping is required to toggle the selected state of the item at *hitIndex*. The *hitIndex* is relative to the displayed items, not the total items; you need to track what the real “top” index is.

hitIndex An integer identifying an item relative to top of displayed items.

ViewSetupChildrenScript

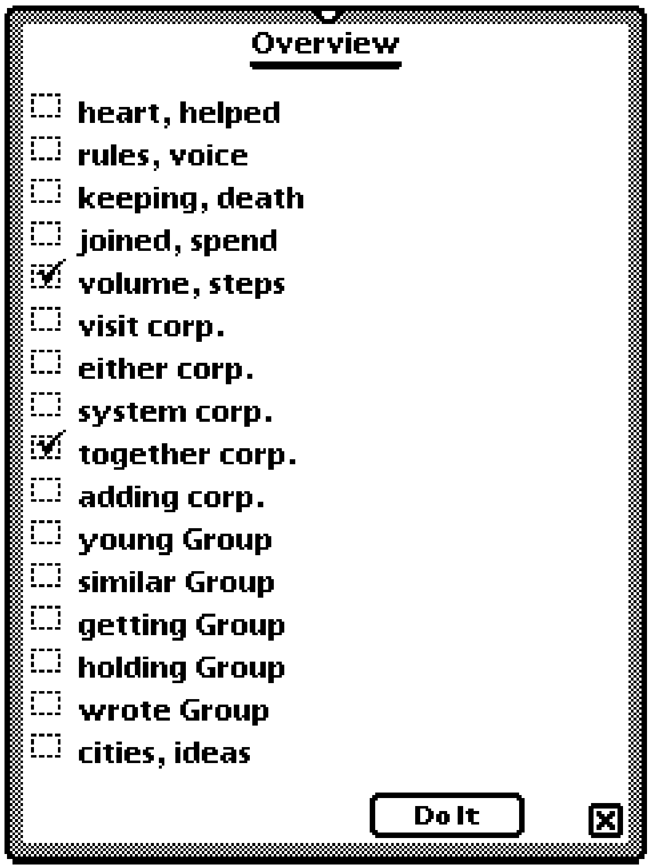
overview:`ViewSetupChildrenScript()`

You must provide this method. You must send the `SetupAbstracts` message from this script. Note that `SetupAbstracts` is expecting a cursor or cursor-like object. See “Using protoOverview” (page 6-24) in *Newton Programmer’s Guide*, for a discussion of how to create a cursor-like object.

protoSoupOverview

This proto is similar to `protoOverview`, but is designed to work with data that consists of soup entries. It expects each overview item to be a soup entry whether or not the cursor itself is an ordinary soup cursor. Figure 5-39 shows an example of this proto.

Figure 5-39 Example of a soup entry proto



A default `ViewSetupChildrenScript` method calls `SetupAbstracts`, passing the cursor in the `cursor` slot. If you override this method, you should call the inherited method once you've set up the cursor slot.

Slot descriptions

`cursor` Required. Set this slot to a cursor describing your entries. Initialized in `ViewSetupFormScript`.

All other slots in `protoSoupOverview` are the same as in `protoOverview`.

Scroller

overview:`Scroller(`*numItems*`)`

This optional method scrolls the overview, with respect to the specified cursor (in the `cursor` slot). The default is to take an integer and move the cursor forward (positive) or backward (negative). If you try to move the cursor forward past the end, the last item is returned. If you try to move the cursor backward before the first item, the first item is returned.

numItems The number of items to scroll; a negative value means to scroll upwards.

SelectItem

overview:`SelectItem(`*index*`)`

This method remembers selected items, doing the right thing with respect to the specified cursor (in the `cursor` slot). It keeps a list of the selected items by getting entry aliases for them, hence the need for the items to be real soup entries.

index The index of an item in the overview. The first item is 0.

Abstract

overview:`Abstract(`*entry*`, `*bounds*`)`

This required method should return a shape or shape list representing an item in the overview.

entry The entry returned from the cursor that was passed to the `SetupAbstracts` method.

bounds A bounds frame within which the returned shape should be placed.

IsSelected

overview:`IsSelected(`*entry*`)`

This method returns non-`nil` if the specified entry is currently selected.

entry An entry from the cursor.

ForEachSelected

overview:`ForEachSelected(`*function*`)`

This method calls the specified function once for each entry that is currently selected. The function is passed one argument, the entry.

function A function object.

protoListPicker

This proto provides a scrollable list of items, from either a soup or an array (or both), from which one or many items may be chosen. The list is built from a soup, using a cursor. By default, this proto queries the “Names” soup, but you can change it to query a different soup.

The selections are intended to be persistent, so enough information from soup entries is maintained to allow the selection to be displayed even if the soup is removed.

The `protoListPicker` proto is based on a view of the `clView` class.

The `viewFlags`, `viewBounds`, `viewJustify`, and `viewFormat` slots can be overridden at will. The `ViewScrollUpScript` and `ViewScrollDownScript` methods are provided for the developer to invoke.

The following slots and methods are used internally: `ViewSetupChildrenScript`, `ViewDrawScript`, `ViewQuitScript`, `fOpenEditView`, `nowShowing`, `fBorder`, `cursor`, `myQuerySpec`, `fCurrentKey`, `MarkCursorPosition`, `filterLabels`, `SetupFiltering`, `SetupCursor`, `RedoCursor`, `GetTargetInfo`, `FilterChanged`, `folderTabs`, `AZtabs`, and `listBase`.

See "Using protoListPicker" (page 6-26) in *Newton Programmer's Guide* for an example of this proto and a discussion of using it.

Slot descriptions

`declareSelf` Set to `'pickBase`.

`defaultJustification`
The default is `vjParentFullH + vjParentTopV`

`viewFlags` The default is `vVisible + vApplication +vClickable`.

`viewBounds` Set to the size and location where you want the list of scrollable items to appear.

`lineHeight` Optional. Set to the height, in pixels, of each line in the list. The default setting is the maximum of the font height and the checkmark height.

`listFormat` Optional. Specify `viewFormat` flags to be used for the `viewFormat` slot of the list child view. The default setting is `vfFrameGray + vfPen(1)`.

`pickerDef` Required. A frame used to determine the overall behavior of the list picker. This frame should be based on `protoNameRefDataDef` or `protoPeopleDataDef`. For an example, see "Using the Data Definitions Frame in a List Picker" (page 6-29) in *Newton Programmer's Guide*

`selected` Required. An array of references. Set this slot in the `ViewSetupFormScript` method if you want the list to be displayed with one or more items preselected.

Note that the name reference data definition contains the `_unselected` slot, which can be used to override the preselection of individual items (even though they are in the `selected` array).

While the list picker is open, the selected list is not valid until the picker's `ViewQuitScript` has run. Any operations on the data should be postponed, either by using the `'postQuit` deferral mechanism, or by calling the inherited `ViewQuitScript` method *before* your own operations.

`soupToQuery` Optional. A string specifying the union soup to query, or a function that returns a soup. This slot overrides any soup specified in the data definition. By default, no soup is queried.

`querySpec` Passed to the query routine. The `tagSpec` slot is replaced internally, and the `validTest` may be enhanced internally to allow the checkbox filtering and folder support. This slot overrides any `querySpec` specified in the data definition.

`suppressNew` Optional. If this slot is present and its value is non-`nil`, the New button is not drawn.

`suppressScrollers`
Optional. If this slot is present and its value is non-`nil`, the up and down scroll arrows are not drawn.

`suppressAZTabs` Optional. If this slot is present and its value is non-`nil`, the a-z tabs are not drawn.

`suppressFolderTabs`
Optional. If this slot is present and its value is non-`nil`, the folder tabs are not drawn.

`suppressSelOnlyCheckbox`
Optional. If this slot is present and its value is non-`nil`, the Selected Only checkbox is not drawn.

`suppressCloseBox`
Optional. If this slot is present and its value is non-`nil`, the close box is not drawn.

`suppressCounter`
Optional. Suppresses the text at the bottom right indicating how many items are selected

`reviewSelections`
Optional. If present and non-`nil`, and if `singleSelect` is `nil`, when the picker is opened with preselected items, the Selected Only checkbox is checked.

`readOnly` Optional. If present and non-`nil`, constrains the interface so that the currently selected list can be viewed but not changed. All taps in the body of the picker are ignored, the New button and Selected Only checkbox are hidden, and the checkboxes are suppressed.

`dontPurge` Optional. If present and non-`nil`, prevents unselected name references from being stripped out of the selected array when the picker is closed. You may also specify this slot in the data definition.

`soupChangeSymbol`
The symbol to use in the `RegSoupChange` message; by default, its `'listpicker`.

The list picker automatically registers notification of soup change in the soup it will query. By default, only the `SoupEnters` and `SoupLeaves` messages are handled. To handle any other messages, or to override the default behavior for the `SoupEnters` or `SoupLeaves` change types, add a slot whose name is the `changeType` you wish to support, and make its value a function of a `soupName` and the `changeData`. This function will be called when the soup notification is received with that `changeType`.See Table 9-1 (page 9-15) for a list of available `changeType` values.

SoupEnters

picker`:SoupEnters(`*soupName*`,` *changeData*`)`

Called when the list picker is notified that the soup has changed and the *changeType* is `'soupEnters`. This means that the soup has become available. By default, redisplays the cursor contents.

soupName The name of the soup that has become available.

changeData The soup itself.

SoupLeaves

picker`:SoupLeaves(`*soupName*`,` *changeData*`)`

Called when a soup becomes unavailable. This method synchronizes the cursor in case it was pointing to an entry removed with a soup and then refreshes the list.

soupName The name of the soup that has become unavailable.

changeData The soup itself. Note that you shouldn't use this soup, since this message means it is no longer available.

SetNowShowing

picker:`SetNowShowing(`*value*`)`

Sending this message is equivalent to tapping the Selected Only button.

value A symbol, where `'all` means show all entries and `'selected` means show only selected entries.

GetSelected

picker:`GetSelected(`*activeOnly*`)`

This method returns a clone of the selected array.

activeOnly A Boolean which, if non-`nil`, returns an array that is stripped of any `_unselected` name references.

protoNameRefDataDef

The `protoListPicker` proto is driven in large part by the data definition specified in the `pickerDef` slot. The `protoNameRefDataDef` proto is provided for creating your own data definitions.

Figure 5-40 (page 5-106) shows an example of a `protoListPicker` whose data definition is based on `protoPeopleDataDef`.

All calls to methods in the `pickerDef` slot are handled by sending the message to the frame itself, so the methods described below can use inherited functions and store data in the frame as needed.

Slot descriptions

`name` The name that appears in the top-left corner of the picker. The default value in Newton devices with English ROMs is “Names”.

`class` A symbol specifying the class to which all name references should be set; the default value is `'nameRef`.

`entryType` When a soup entry is created, its class should be set to this type. The `makeNameRef` routine should respect this slot.

`columns` An array of column specifications; for details, see "Column Specifications" (page 5-3); for an example, see "Specifying Columns" (page 6-29) in *Newton Programmer's Guide*. The default is a single, full-width column whose `fieldPath` is `'name`.

`singleSelect` Optional. If this slot is present and its value is non-`nil`, only a single item at a time can be selected from the list. (Selecting additional items deselects the original.)

Do not pre-load the `selected` slot with multiple selected name references and then specify `singleSelect`.

`soupToQuery` A string specifying the union soup to query or a function returning a soup. All data displayed is retrieved from this soup.

`querySpec` Passed to the query routine. The `tagSpec` slot is replaced internally, and the `validTest` may be enhanced internally to allow the checkbox filtering and folder support. By default all Names entries are displayed.

`validationFrame`

A validation frame acceptable to the `ValidityCheck` system global function. Used by the default `ValidityCheck` method. The default value is `nil`.

MakeCanonicalNameRef

dataDef`:MakeCanonicalNameRef(`*object*`,` *dataClass*`)`

Creates and returns a name reference with no application-specific slots. This method should not be overridden, but can be called if needed.

object An entry, an alias, a name reference, a frame, or `nil`.

dataClass Optional. The class of the entry.

MakeNameRef

dataDef:`MakeNameRef(`*object*`, `*dataClass*`)`

Creates a name reference with one additional slot `name` (by calling `MakeCanonicalNameRef`). Overrides of this method should generally call `MakeCanonicalNameRef` and fill in the slots that are needed.

If you are using `protoListPicker` to browse an array, this method should be overridden to add the slots returned by `MakeCanonicalNameRef` to the items in the array. To remove these slots, use the `PrepareToAdd` method.

object	An entry, an alias, a name reference, a frame, or `nil`.
dataClass	Optional. The class of the entry. If this is not specified, it's taken from the data definition.

Get

dataDef:`Get(`*object*`, `*fieldPath*`, `*format*`)`

Returns a value from the specified object, retrieved from the column specification.

object	An entry, an alias, a name reference, a frame, or `nil`.
fieldPath	A symbol uniquely identifying the field that should be displayed in this column. This symbol is used by the list picker to retrieve the data, and (in most cases and certainly the default case) is the actual path in the entry to the data field desired. However, it is possible to use the symbol purely as a marker—for example if the particular data required is a calculated aggregate of a number of data fields—as long as all the routines in the data definition that use this symbol are overridden to recognize this usage.
format	Determines the value returned; possible values are `'text`, `'sortText`, or `nil`. If `nil`, the actual field is desired. If `'text`, a text representation is requested. The value`'sortText` should be used only for the first

column in the `columns` array. For example, assume the following is defined:

```
local aName := '{first: "Cindy", last: "Peters"};
```

The result of calling the default `Get` method

```
[:Get(aName, 'name, format)]
```

depends on the value of the format parameter:

`'text`	"Cindy Peters"
`'sortText`	"Peters, Cindy"
`nil`	{first: "Cindy", last: "Peters"}

If the first column specification has `fieldPath = 'fruitType`, the overridden `Get` function should support `'sortText` for `'fruitType`, but all other fields need only support `nil` and `'text`.

GetPrimaryValue

dataDef`:GetPrimaryValue(`*object*`, `*format*`)`

Called by the default `Get` method to retrieve the data. The default method returns `nil`.

object An entry, an alias, a name reference, a frame, or `nil`.

format Determines the value returned; see `Get` method for details.

HitItem

dataDef`:HitItem(`*tapInfo*`, `*context*`)`

Called when the user taps in the picker. This method should return either a reference to a view opened as a result of the tap, or `nil`. If a view is opened, all tap processing by the list picker is suppressed until the data definition passes control back to the list picker by calling `context:Tapped(action)`; described on page 5-102.

tapInfo	A frame containing the following slots:	
	`nameRef`	The name reference that was tapped.
	`tapIndex`	The visible index of the name reference, or `nil` for "new."
	`bbox`	The bounding box for the cell that was tapped.
	`fieldPath`	The `fieldPath` for the column tapped, or `'new` if it was the New button.
	`editPaths`	All columns for this list.
	`popup`	Used in pop-up processing.
context	The view handling the tap.	

MakePopup

dataDef`:MakePopup(`*object*`, `*fieldPath*`)`

Returns `nil` or an array suitable for passing to the `PopupMenu` method. If the value of an item in the pop-up view is different from the `item` slot, the slot `value` should hold the proper value. If the item is to open the editor, the `value` slot should be the symbol `'openeditor`. This method is called by the list picker to determine when to precede a column with a diamond character. If you override the default `HitItem` method, this method should return non-`nil` to get the diamond character.

If an array is returned, it is popped up by `PopupMenu`. If `nil` is returned, the `HandleTap` method is called.

object — An entry, alias, a name reference, a frame, or `nil`.

fieldPath — A symbol uniquely identifying the field that should be displayed in this column. This symbol is used by the list picker to retrieve the data, and (in most cases and certainly the default case) is the actual path in the entry to the data field desired.

Tapped

context:`Tapped(`*action*`)`

Call this method from the `HitItem` method to indicate that a tap has been handled.

action	A symbol indicating what action to take in response to a tap. The following values can be specified:	
	`'select`	Select the item.
	`'toggle`	Toggle between selected and unselected state.
	`nil`	Do nothing.

New

dataDef:`New(`*tapInfo*`, `*context*`)`

Called when the user taps the New button. This method should return either a reference to a view opened as a result of the tap, or `nil`. If a view is opened, all tap processing by the list picker is suppressed until the data definition passes control back to the list picker by calling `context:Tapped(action)`.

If a `validationFrame` slot is provided, the default `New` method opens a label input line slip (as in the default editing for an item) allowing editing of a new entry with one child view for each column in the picker.

tapInfo	A frame containing the following slots:	
	`nameRef`	The name reference that was tapped.
	`tapIndex`	The visible index of the name reference, or `nil` for "new."
	`bbox`	The bounding box for the cell that was tapped.
	`fieldPath`	The field path for the column tapped, or `'new` if it was the New button.
	`editPaths`	All columns for this list.
	`popup`	Used in pop-up processing.

context	The view handling the tap.

DefaultOpenEditor

dataDef:`DefaultOpenEditor(`*tapInfo*`, `*context*`, `*why*`)`

You can call this method to open an edit view, for editing an existing record or in response to a tap on the New button.

The `DefaultOpenEditor` method causes a call to either `DefaultEditDone` or `DefaultNewDone` when the edit slip is closed.

tapInfo	A frame containing the following slots:	
	`nameRef`	The name reference that was tapped.
	`tapIndex`	The visible index of the name reference, or `nil` for `'new`.
	`bbox`	The bounding box for the cell that was tapped.
	`fieldPath`	The field path for the column tapped, or `'new` if it was the New button.
	`editPaths`	All columns for this list.
	`popup`	Used in pop-up processing.
context	The view handling the tap.	
why	A symbol that can be either `'edit` or `'new`.	

OpenEditor

dataDef:`OpenEditor(`*tapInfo*`, `*context*`, `*why*`)`

You can add this method and call it instead of `DefaultOpenEditor` if you need more flexibility than is provided by `DefaultOpenEditor`. You also need to draw the layout for each editor you need.

The arguments and return value are as per `DefaultOpenEditor`. See "Validation and Editing in protoListPicker" (page 6-31) in *Newton Programmer's Guide* for an example.

tapInfo	A frame containing the following slots:	
	`nameRef`	The name reference that was tapped.

`tapIndex`	The visible index of the name reference, or `nil` for "new."
`bbox`	The bounding box for the cell that was tapped.
`fieldPath`	The field path for the column tapped, or `'new` if it was the New button.
`editPaths`	All columns for this list.
`popup`	Used in pop-up processing.

context The view handling the tap.

why A symbol that can be either `'edit` or `'new`.

NewEntry

dataDef`:NewEntry(`*nameRef*`,` *label*`)`

Returns a new soup entry, filled in as much as possible from the name reference passed in, and with the tags slot set appropriately so that the entry is in the current folder. The new entry's class slot is given the value specified by the `cardType` slot in the data definition.

nameRef Holds the new soup information.

context The view handling the tap.

Note

If the soup doesn't exist, this method fails silently. ◆

ModifyEntry

dataDef`:ModifyEntry(`*nameRef*`,` *fieldPath*`)`

Returns the modified entry. Sets the field named by *fieldPath* in the underlying soup entry for the name reference. It then calls `EntryChangeXmit` on the entry.

nameRef The name reference for the entry that underwent the modifications.

fieldPath The array of the paths into the *nameRef* that changed.

Validate

dataDef:`Validate(`*nameRef*`,` *pathArray*`)`

You can add this method if you want to deal with nested soups or otherwise need more flexibility than you get when you use the `ValidateFrame` slot. This should return an array of paths that failed, or an empty array.

nameRef Name reference to validate.

pathArray Array of paths to validate in the name reference.

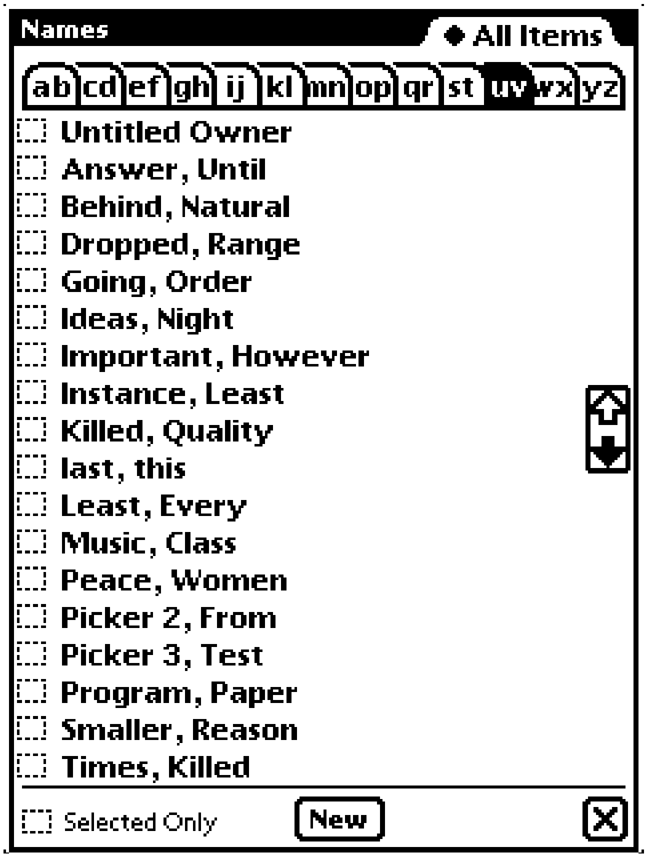
Validate each path in *pathArray* in the given *nameRef*. Accumulate a list of paths that are not valid and return them. See "Validation and Editing in protoListPicker" (page 6-31) in *Newton Programmer's Guide* for an example.

protoPeopleDataDef

The `protoPeopleDataDef`, which is based on the `protoNameRefDataDef`, is the basis of the built-in data definitions used by `protoPeoplePicker` and `protoMeetingplacePicker`.

Figure 5-40 shows an example of a `protoListPicker` whose data definition is based on `protoPeopleDataDef`.

Figure 5-40 A `protoListPicker` based on `protoPeopleDataDef`



Slot descriptions

`entryType` When a soup entry is created, its class should be set to this type. The `makeNameRef` routine should respect this slot. The default value is `'person`.

`soupToQuery` A string specifying the union soup to query or a function returning a soup. All data displayed is retrieved from this soup. By default the Names soup is queried.

`primaryPath` Optional. Symbol used to indicate that a specific column is the primary path. The primary path is treated specially in that the data displayed can be retrieved from multiple source slots; that is, the primary path for a card is the `name`, but for a company card the name data comes from the `company` slot. The mapping of where the data comes from is specified by the `primaryPathMapper`.

`primaryPathMapper`
Optional. A frame where each slot maps an entry class to the slot from which the data for the primary path should be retrieved. So, for example, the `primaryPathMapper` for the cardfile is

```
{person: name,
owner: name,
company: company,
group: group,
worksite: place,}
```

`superSymbol` (Used exclusively to support routing.) Used as usual for `dataDefs`. However, if the `superSymbol` is `'groupTransport`, the list picker type defined by this `nameRef` is available as one of the routing choices in the group card in the Names application. The name displayed in that application is the value of the `name` slot in the data definition.

`routePath` (Used exclusively to support routing.) Used by the `GetRoutingInfo` function to determine which `nameRef` slot contains the routing information.

The `protoPeopleDataDef` uses the methods described in the following sections. The additional methods `GetRoutingInfo`, `GetItemRoutingFrame`, `GetRoutingTitle`, and `PrepareForRouting` are used exclusively to support routing. They can be ignored if the data definition is not intended for routing or can be overridden if necessary.

Equivalent

dataDef:`Equivalent(`*nameRef1*`,` *nameRef2*`,` *pathArray*`)`

This method compares the data in two name references and returns an array of all paths that contain nonequivalent (in terms of what is displayed) values.

The default method handles strings and immediates; anything more sophisticated should be overridden here.

nameRef1	A name reference.
nameRef2	A name reference.
pathArray	An array of paths.

If you are using the default editing methods with a slot containing a frame, you need to override this method as well as provide a `validationFrame` (or override the `Validate` method). The `ModifyEntry` method is not responsible for deciding if an entry should be modified; when it is called, all the paths specified in the `fieldPath` parameters have been changed and should be entered properly in the appropriate Names soup entry.

Validate

dataDef:`Validate(`*nameRef*`,` *pathArray*`)`

This method returns an array of invalid paths.

nameRef	A name reference.
pathArray	An array of paths.

ModifyEntryPath

dataDef:`ModifyEntryPath(`*nameRef*`,` *entry*`,` *path*`)`

This method handles the modification of currently defined Names soup entries. For nonprimary paths, it sets `entry.(path) := nameRef.(path)`. For the primary path (phone numbers, e-mail addresses and so on), it sets the `sortOn` and `class` slots correctly.

In other words, you should override `ModifyEntry` as appropriate, iterate across the paths, write through those for which `entry.(path) := nameRef.(path)` isn't sufficient (other than the primary path), and call the inherited `ModifyEntryPath` for all the others.

nameRef	A name reference.
entry	A Names soup entry.
nameRef	A field path.

GetRoutingInfo

dataDef`:GetRoutingInfo(`*object*`)`

This method retrieves all the routing information for an item. By default, this method just calls `GetRoutingFrame` on the item. However, if the item is a group, this method iterates across each member, as returned by `Get(`*item*, *routePath*, *nil*), and recursively calls `GetRoutingInfo` for each member.

object	An entry, alias, name reference, frame, or `nil`.

GetItemRoutingFrame

dataDef`:GetItemRoutingFrame(`*item*`)`

This method is required for transport name references. It is called by the `GetRoutingInfo` method to convert the specific routing information into a form acceptable by the transport.

entry	The name reference of the entry from which to get the routing information.

GetRoutingTitle

dataDef`:GetRoutingTitle(`*objects*`,` *width*`,` *font*`)`

Similar to the `GetRoutingInfo` method, `GetRoutingTitle` is called by the transport code to create a string to display as the target of the transport. The string that is displayed is retrieved from the `primaryPath` slot.

objects	A name reference, an array of name references, or `nil`.

width The maximum length of the string, as specified in number of pixels.

font The font in which the string is rendered.

PrepareForRouting

dataDef`:PrepareForRouting(`*nameRef*`,` *fieldPath*`,` *format*`)`

This method is called to strip any information that is context specific (aliases, for instance) from the specified name reference.

object A name reference.

protoPeoplePicker

This proto implements a picker showing names from the Names application, along with associated phone numbers, fax numbers, or email addresses. In cases where several choices are possible, the picker allows selection using a pop-up selector. The proto also allows the user to add new entries, or additional information for existing entries.

This proto works with the data definition registry, using predefined data definitions to implement the picker behavior.

Slot descriptions

`class` A symbol specifying the type of data to display, and the data definition used to display it. You can specify the following values:

`|nameRef.people|`	names
`|nameRef.phone|`	phone numbers
`|nameRef.fax|`	fax numbers
`|nameRef.email|`	e-mail addresses

`selected` This slot is inherited by `protoListPicker` and contains an array of name references for selected items. These items may have been selected from the picker or added by the user. Note that some clean-up is conducted when the `ViewQuitScript` of `protoListPicker` is called, so the `selected` array should be used only after this executes (in other words, in a deferred send).

An array of name references may be passed in to the picker when it is first opened to establish defaults for the current item selections.

All other behavior is provided by the data definition; see `protoNameRefDataDef` for details.

protoPeoplePopup

This proto is similar to `protoPeoplePicker`, but opens a pop-up view containing the picker (instead of having the picker embedded in the application).

Slot descriptions

`class` — A symbol specifying the type of data to display, and the data definition used to display it. You can specify the following values:

`|nameRef.people|`	names
`|nameRef.phone|`	phone numbers
`|nameRef.fax|`	fax numbers
`|nameRef.email|`	e-mail addresses

`selected` — This slot is inherited by `protoListPicker` and contains an array of name references for selected items. These items may have been selected from the picker, or added by the user. Note that some cleanup is conducted when the `ViewQuitScript` of `protoListPicker` is called, so the `selected` array should only be used after this executes (in other words, in a deferred send or `'postQuit` operation).

An array of name references may be passed into the picker when it is first opened to establish defaults for the current item selections.

`context` — Optional. The name of the view containing the `PickActionScript` method.

`options` — All slots in this frame are copied to the `protoListPicker` view, so anything that can be specified to `protoListPicker` can be specified in the `options` slot. You can override any slot in the pop-up view; for instance, the `suppressNew` slot.

PickActionScript

picker`:PickActionScript(`*selected*`)`

This method is called when the pop-up view is closed.

selected — The `selected` array.

All other behavior is provided by the data definition; see `protoNameRefDataDef` for details.

Roll Protos

These protos are used to implement roll views. A roll view consists of several discrete subviews, arranged vertically, one above the other. The roll can be viewed in overview mode, where each subview is represented by a single-line description. Any single view or all views can be expanded to full size.

protoRoll

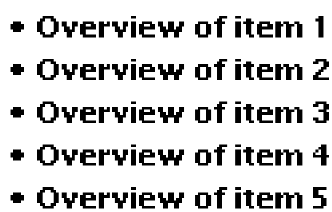
This proto is used to create a roll-like view that includes a series of individual items (other views) that the user can see either as a collapsed list of one-line overview descriptions or as full-size views. When an overview line is tapped, all the full-size views are displayed, with the one that was tapped shown at the top of the `protoRoll` view. Each view occupies the full width of the `protoRoll` and the views are arranged one above the other.

The user can then scroll through all the expanded views by using the universal scrollers (up and down arrows). The user can also tap the

Overview button (the dot between the up and down arrows) to get back to the overview list to select another item.

In the collapsed view, the items in the overview list are preceded by bullets. Figure 5-41 shows an example of this type of view.

Figure 5-41 Example of a rolled list of items



The following `protoRoll` methods are defined internally: `ViewSetupChildrenScript`, `ViewScrollUpScript`, `ViewScrollDownScript`, `ViewOverviewScript`, `GetOverview`, and `ShowItem`. If you need to use one of these methods, be sure to call the inherited method also (for example, `inherited:?ViewSetupChildrenScript()`), otherwise the proto may not work as expected.

The `protoRoll` is based on a view of the `clView` class. It has no predefined child views, though they are dynamically created at run time from the view templates you place in the `items` slot.

Slot descriptions

`viewFlags`	The default setting is `vApplication + vClipping`.
`viewBounds`	By default, the bounds are set to the entire screen, beginning 16 pixel lines down from the top. This would leave room for a title at the top if the `protoRoll` was placed inside a `protoApp`.
`items`	An array of templates that correspond to the items in the list. Each of these should use `protoRollItem` as

its proto. For details, see “protoRollItem” (page 5-119). Because this slot cannot usually be set until run time, you should set it in the `ViewSetupFormScript` method.

`allCollapsed` Optional. If this slot is set to a non-`nil` value, the roll is initially displayed in a collapsed state; that is, only the list of one-line overviews is displayed. If this slot is `nil`, the roll is initially displayed in an expanded state. The default is `nil`.

`index` This slot is used only when `allCollapsed` is set to `nil`; that is, when the roll is initially displayed in an expanded state. Items from the `items` array are displayed in the roll beginning with the item at this index.

`declareSelf` Must be set to `'roll`. This identifies the view that should receive scroll and overview events. This view must also be immediately enclosed by a parent view that has the `vApplication` view flag set, in order for scrolling and overview handling to operate properly.

Here is an example of a template using `protoRoll`:

```
myRoll := {...
_proto: protoRoll,
declareSelf: 'roll,
allCollapsed: true,
index: 0,
items: [
{_proto:protoRollItem,
height:50,
overview:"Overview of item 1",
viewBounds:{left:0,top:0,right:0,bottom:50},
stepChildren:[{_proto:protoStaticText,
            text:"This is the first test roll item",
            viewJustify: vjParentFullH + vjParentFullV,
            viewBounds:{left:0,top:0,right:0,bottom:0},
```

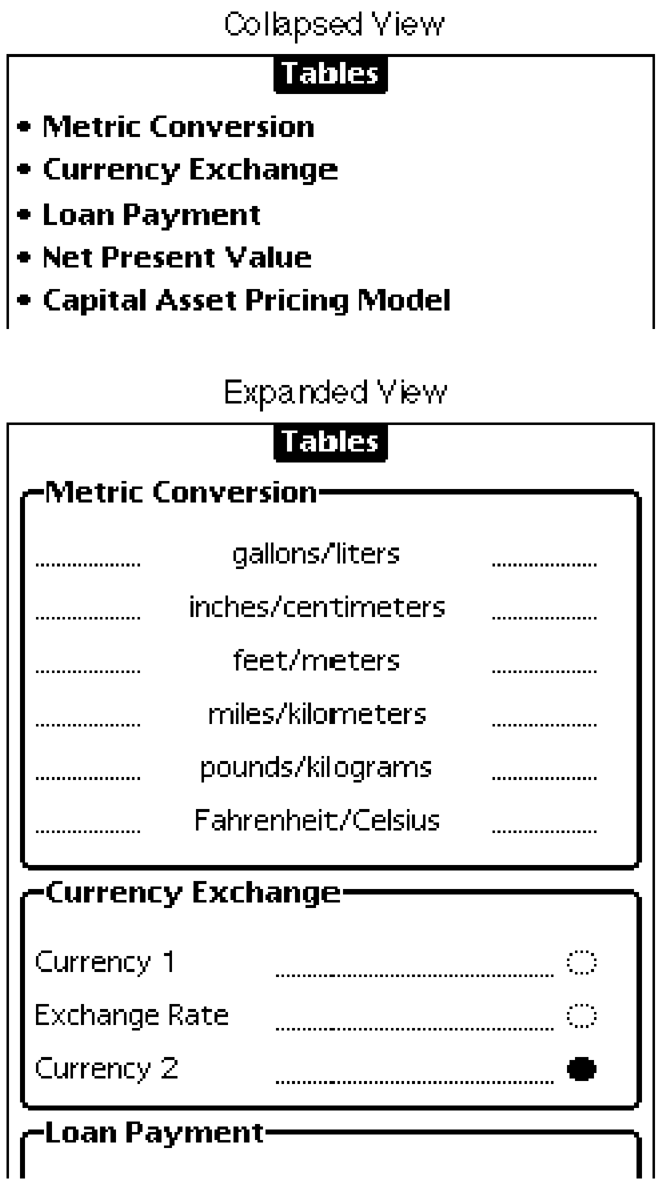
```
            viewfont:ROM_fontSystem12 }],
},
{_proto:protoRollItem,
height:200,
overview:"Overview of item 2",
viewBounds:{left:0,top:0,right:0,bottom:200},
stepChildren:[{_proto:protoStaticText,
            text:"This is the second test roll item",
            viewBounds:{left:0,top:0,right:0,bottom:0}
}],
},
{_proto:protoRollItem,
height:200,
overview:"Overview of item 3",
viewBounds:{left:0,top:0,right:0,bottom:200},
stepChildren:[{_proto:protoStaticText,
            text:"This is the third test roll item",
            viewBounds:{left:0,top:0,right:0,bottom:0}
}],
},
{_proto:protoRollItem,
height:50,
overview:"Overview of item 4",
viewBounds:{left:0,top:0,right:0,bottom:50},
stepChildren:[{_proto:protoStaticText,
            text:"This is the fourth test roll item",
            viewBounds:{left:0,top:0,right:0,bottom:0}
}],
}],
...};
```

protoRollBrowser

This proto is similar to `protoRoll`, except that the `protoRollBrowser` is an entirely self-contained application. It is based on the `protoApp` proto, so it has a title and a status bar. Also, it need not be contained in another view.

The `protoRollBrowser` works exactly like the `protoRoll` in other respects. Figure 5-42 shows an example of a `protoRollBrowser` view in its collapsed and expanded states:

Figure 5-42 Example of a collapsed and expanded rolled list of items



The `protoRollBrowser` uses the `protoApp` proto. A `protoRollBrowser` has the following three child views:

- A roll view, based on `protoRoll`. This view occupies most of the parent view, except for the title and status bar areas.
- A title, based on `protoTitle`.
- A status bar, based on `protoStatus`.

Slot descriptions

`viewBounds`	Set to the size and location where you want the roll to appear. By default it is centered horizontally within its parent view.
`viewJustify`	Optional. The default setting is `vjParentCenterH`.
`viewFormat`	Optional. The default setting is `vfFillWhite + vfFrameBlack + vfPen(1) + vfInset(1) + vfShadow(1)`.
`title`	A string that is the title. This title appears in a title bar at the top of the roll. (It uses `protoTitle` to create the title.)
`rollItems`	An array of templates that correspond to the items in the list. Each of these should use `protoRollItem` as its proto. Because this slot cannot usually be set until run time, you should set it in the `ViewSetupFormScript` method.
`rollCollapsed`	Optional. If this slot is set to a non-`nil` value, the roll is initially displayed in a collapsed state; that is, only the list of one-line overviews is displayed. If this slot is `nil`, the roll is initially displayed in an expanded state. The default is non-`nil`.
`rollIndex`	This slot is used only when `rollCollapsed` is set to `nil`; that is, when the roll is initially displayed in an expanded state. Items from the `items` array are displayed in the roll beginning with the item at this index.

`declareSelf` Do not change. This slot is set by default to `'base`. This identifies the view to be closed when the user taps the close box.

Here is an example of a template using `protoRollBrowser`:

```
myRollBrowser := {...
_proto: protoRollBrowser,
title: "My RollBrowser",
rollCollapsed: true,
declareSelf: 'base,
rollitems: [
{_proto:protoRollItem,
height:50,
overview:"Overview of item 1",
viewBounds:{left:0,top:0,right:0,bottom:50},
stepChildren:[{_proto:protoStaticText,
            text:"This is the first test roll item",
            viewJustify: vjParentFullH + vjParentFullV,
            viewBounds:{left:0,top:0,right:0,bottom:0},
            viewfont:ROM_fontSystem12 }],
},  // ... and so on
],
...};
```

protoRollItem

This proto is used for one of the views in a roll (based on `protoRoll` or `protoRollBrowser`). You should specify an array containing one or more views based on `protoRollItem`. Each item in the array represents one of the views in the roll.

The `protoRollItem` is based on a view of the class `clView`.

Note that the `protoRollItem` proto is not used by picking it from the view palette in NTK. You use this proto by writing a textual description of your template, referring to this proto in the `_proto` slot of your template frame.

You write one template frame for each item to be shown in the roll, and place them all in an array in the `items` slot of the roll. See `protoRoll` for an example.

Slot descriptions

`viewBounds`	Typically, you set the bounds to 0, 0, 0, `height`. The first three bounds parameters are not needed because the view is positioned below the previous child and fully horizontally justified within the roll. However, you can specify values other than zero to indent the view from the sides of its parent or to separate it from its preceding sibling, but keep in mind how the `viewJustify` setting affects the interpretation of the `viewBounds` values. For more information on the `viewJustify` slot, see "View Alignment" (page 3-13) in *Newton Programmer's Guide*.
`viewFormat`	Optional. The default setting is `vfFillWhite + vfFrameBlack + vfPen(1)`.
`viewJustify`	Optional. The default setting is `vjSiblingBottomV + vjParentFullH`.
`overview`	A string that is the one-line overview to be displayed for this view when the roll is collapsed and only the overview list is shown.
`height`	Set to the height of the view, in pixels.
`stepChildren`	An array containing one or more child views that belong to the view that is this particular roll item. These are shown when this item is expanded (tapped by the user, or scrolled to after the roll has already been expanded). Typically, each child view uses a proto and can include whatever slots are important for use with its particular proto.

View Classes

The following view class is used to display an expandable text outline.

Outline View (clOutline)

The `clOutline` view class is used to display an expandable text outline. Figure 5-43 shows an example.

Figure 5-43 Example of an expandable text outline

My First Heading
    First level 2 head
    Another level 2 head
        Wow—a third level!
Second main heading
Third main heading

The `clOutline` view class includes these features:

- Multilevel outline (up to 15 levels), with each outline level indented from the previous one.
- Headings that can be expanded (those that contain subheadings) are shown in bold automatically.
- Headings the user can expand to show subheadings by tapping the heading. Another tap on the heading collapses it, hiding its subheadings.
- Only one main heading can be expanded at a time. If the user taps a different heading, any other expanded heading is automatically collapsed, and the new heading is expanded.

Slot descriptions

`viewBounds` — Set to the size and location where you want the view to appear.

`browsers` — An array containing one frame item, `List`, which is itself an array of the items to be shown in the outline. Each outline item is a frame containing these slots:

- `level` — The outline level of this item. “1” specifies a top-level heading, “2” specifies a second-level heading, and so on. This slot can be omitted for top-level items; it defaults to level 1. You can use up to 15 levels.
- `name` — A string that is the text to be shown in the outline. Tabs are not allowed in the text.

`viewFont` — Specify the font to be used for the text in the outline. It’s best not to specify a bold font since bold is added automatically for headings that have subheadings. If you specify bold, all the text will be bold. The default font is `ROM_fontSystem10`.

`viewFlags` — The default setting is `vVisible` + `vClickable` + `vReadOnly`.

`viewFormat` — Optional. The default setting is `nil`.

`clickSound` — Optional. Specify a sound frame. This sound is played when the user taps any item in the outline.

OutlineClickScript

outline`:OutlineClickScript(`*index*`,` *unused*`)`

This method is called whenever the user taps an item in the outline. This function must return non-`nil`.

index — The index of the outline item in the `List` array (inside the `browsers` slot).

unused — Unused.

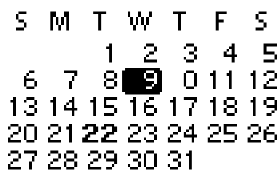
Here is an example of a view definition of the `clOutline` class:

```
myOutline := {...
   viewclass: clOutline,
   viewFlags: vVisible+vClickable+vReadOnly,
   viewBounds: {left: 25, top: 56, right: 220,
               bottom: 232},
   viewFont: ROM_fontsystem12,
   clickSound: ROM_flip,
   browsers: [{list: [
      {level:1, name:"My First Heading"},
      {level:2, name:"First level 2 head"},
      {level:2, name:"Another level 2 head"},
      {level:3, name:"Wow—a third level!"},
      {level:1, name:"Second main heading"},
      {level:2, name:"Section 2 subhead 1"},
      {level:2, name:"Section 2 subhead1"},
      {level:1, name:"Third main heading"},
      {level:2, name:"Last subhead"},
      ] }],
   OutlineClickScript: func(index, dummy)
      begin
      Print("You picked browser item " & index);
      true;
      end,
...};
```

Monthly Calendar View (clMonthView)

The `clMonthView` view class is used to display a monthly calendar. Figure 5-44 shows an example of a monthly calendar view.

Figure 5-44 Monthly calendar view



Selected days are highlighted with an inverted rounded rectangle. The current day is shown in bold, if it appears in the month that is displayed.

Here is an example of a view definition of the `clMonthView` class:

```
theMonth := {...
   viewclass: clMonthView,
   viewBounds: {left: 58, top: 82, right: 186,
                bottom: 178},
   viewflags: vVisible+vClickable,
   labelFont: ROM_fontSystem9Bold,
   datesFont: ROM_fontSystem9,
   selectedDates: nil,
   ViewSetupFormScript: func()
      begin
      self.selectedDates := [Time()];
      end,
...}
```

These slots are of interest for a view of the `clMonthView` class:

Slot descriptions

`viewBounds` Set to the size and location where you want the view to appear.

`selectedDates` Required. Initially, this slot must be set to an array containing a single element that is a time value. (For

	example, you can use the `Time` function to return the current time.) The month displayed is the month in which this time value occurs. If the user makes a selection of days in the month, this slot holds an array of time values, one for each of the days selected. All time values are represented as the number of minutes passed since midnight, January 1, 1904.
`year`	A read-only slot that holds an integer that is the year of the month shown.
`month`	A read-only slot that holds an integer that is the number of the month shown (January=1, . . . , December=12).
`viewFlags`	The default setting is `vVisible+vClickable`.
`viewFormat`	Optional. The default setting is `nil`.
`datesFont`	Optional. The font used for the day numbers. The current day's date is shown in bold. The default font is `ROM_fontSystem9`.
`labelFont`	Optional. The font used to label the days above the dates. If you omit this slot or set it to `nil`, the day labels are not shown. The default font from NTK is `ROM_fontSystem9Bold`.
`noSelection`	Optional. You should set this slot to `true` if you do not want the initial date highlighted (selected) when the month view is first displayed. The default is `nil`.
`singleDay`	Optional. You should set this slot to `true` to force single-day selection only (in which the user cannot select multiple days). The default is `nil`, meaning that multiple day selection is allowed.

Typically, the `selectedDates` slot resides in the parent view of the month view and is found through inheritance when the month view is instantiated. This allows the parent and its other child views to have access to the date selection from the month view.

The following methods are of interest in `clMonthView`.

MonthChangedScript

monthView:`MonthChangedScript()`

Called when the date selection changes.

This method lets you take an action when the date selection changes. The new selected dates are stored in an array in the `selectedDates` slot.

The return value of this method is ignored.

ViewSetupFormScript

monthView:`ViewSetupFormScript()`

Called before the month view is opened.

This method is set by default in NTK to the following line of code:

```
self.selectedDates := [Time()];
```

This code causes the view to display the current month when it is opened.

Pop-up Functions and Methods

The following functions and methods are used in creating pop-up views.

PopupMenu

view:`PopupMenu(`*pickItems*`, `*options*`)`

Creates a dynamic pop-up list view, or picker, from which one item can be selected.

`PopupMenu` returns the picker view that it creates.

pickItems An array of items that you want to appear in the picker list. The elements in the array appear with the first item at the top of the list, continuing down to the last item. If the list contains more items than can be shown on the screen at one time, the user can scroll it to see more items. For more details on the items you can specify in the picker list, see the section “Specifying the List of Items for a Popup” (page 6-37) in *Newton Programmer’s Guide*.

Often this is simply an array of strings to appear in the list.

If you list items that include icons, be aware that `PopupMenu` scales the items to the maximum of the icon height and text height. You can force this to a desired value (for all items, except separators) by adding this option slot to the *first* item:

```
view:PopupMenu([{item: "first one", fixedheight:
        22}...)
```

If you use icons in a list that can become large enough to scroll, you should specify the `fixedHeight` slot for *every* item.

If you find the indentation and placement of your icons and text are ragged, you can provide an indent slot for the first item, which forces every item to be indented correctly; for example:

```
view:PopupMenu([{item: "first one", indent: 28}...)
```

To insert a light or dark separator line between two items, place `'pickseparator` or `'picksolidseparator` in the item list.

To add a nonpickable item or place a mark next to an item, specify the item as a frame containing the following slots:

`item` The item string.

`pickable` Specify non-`nil` if you want the item to be pickable, or `nil` for not pickable. Nonpickable items appear in the list but are not highlighted and can’t be selected.

`mark` A character to be displayed next to the item. You can specify a character with either a dollar sign followed by the character code (`$\uFC0B`, for example, produces the check mark symbol in the Espy font) or one of the character constants (`kCheckMarkChar`, for example).

options A value that specifies where the pop-up menu appears.

There are a number of possible options. For button pop-up views, specify `nil`; the pop-up view is placed adjoining the view to which the `PopupMenu` message is sent (not obscuring the button).

You can also specify a frame with two slots—`left` and `top`—which define the top-left corner of the pop-up view. The new view is placed relative to the view from which `PopupMenu` is called (the parent). The left edge of the new rectangle is inset `left` pixels from the left side of the parent, and the top edge is inset `top` pixels from the top edge of the parent.

You can also provide a `'bounds` slot that is a bounds frame that specifies, in local coordinates, the rectangle next to which the pop-up view should appear.

When an item in a picker is selected, the system sends the `PickActionScript` message to the view identified by `self` (the view from which `PopupMenu` was called). You must define `PickActionScript` as a method that accepts one parameter. The parameter passed to `PickActionScript` is the array index of the item number selected in the list (the first item has an index of zero).

If no item is selected—that is, if the user taps outside the picker to close it—the `PickCancelledScript` message is sent to the view identified by `self`. If you want to handle this message, define a method that accepts no parameters, since none is passed.

`SetItemMark` and `GetItemMark` are two methods provided for picker views. You can use them within the `PickActionScript` method (or elsewhere) to set and get the mark for an item. You call these methods as follows:

popupView`:SetItemMark()`
popupView`:GetItemMark()`

where *popupView* is the view returned by the `PopupMenu` method. For details on these methods, see `protoPicker`.

The picker view created by `PopupMenu` is automatically closed after the user selects an item or taps outside the view.

Name Reference Functions

The following global routines are provided for working with name references.

IsNameRef

`IsNameRef(`*item*`)`

This function returns non-`nil` if the specified item is a name reference (as determined by the presence of an `_alias` slot).

AliasFromObj

`AliasFromObj(`*item*`)`

This function returns an alias, if possible. If the item is an alias, it is simply returned. If the item is a soup entry, an alias to it is created and returned. If the entry is a name reference, the alias to its entry is returned. In all other cases, `nil` is returned.

EntryFromObj

`EntryFromObj(`*item*`)`

This function returns an entry if possible. Basically, it looks for an entry alias and then tries to resolve the entry from it.

ObjEntryClass

`ObjEntryClass(`*item*`)`

This function returns the class of the entry returned by the `EntryFromObj` function.

Controls Reference

This chapter provides reference information for the control protos that you can use in your applications. You use the control protos to provide various user interface and view enhancement features in your applications. This chapter describes the following controls and other protos:

- horizontal and vertical scrollers
- boxes and buttons
- alphabetical selection tabs
- gauges and sliders
- time-setting displays
- special views
- view appearance enhancements
- status bars

Scroller Protos

Scrollers allow the user to move vertically or horizontally through a display that is bigger than the view. The Newton System Software provides a number of scrollers that allow users to scroll their views.

For an overview of using the scroller protos in your applications and a description of how to implement a simple scroller, see “Scroller Protos” (page 7-2) in *Newton Programmer’s Guide*.

protoHorizontal2DScroller

This proto is used to include both left/right and up/down scrollers, centered at the bottom of a view. Note that most units are expressed in terms of scrollable items (cells, lines, and so on) rather than pixels. The following figure shows the possible scrolling directions.

The `viewBounds` and `viewJustify` slots of `protoHorizontal2DScroller` are set up to center the scroller in its parent view. Change these slots only if you want the scroller in a different location.

Slot descriptions

`scrollView`	Optional. Messages are sent to this view; the default is the template. You usually set this slot in the `ViewSetupForm` script.
`scrollRect`	Optional. Extent of scrollable area, in units to scroll (lines, pixels, and so on).
`dataRect`	Optional. Extent of data in the view. This is often the same value as `scrollRect`.
`viewRect`	Optional. Extent of visible area.

`scrollAmounts` — Optional. An array of three numbers passed to you for scrolling: [line, page, double-tap]. The default is [1, 1, 1].

`pageThreshold` — Optional. The number of lines scrolled before scrolling in pages; the default is 5.

The following slots represent the current offset from the scrollable area. For example, if you scroll to the right, `xPos` is a positive value.

`xPos` — Current horizontal coordinate in the `scrollRect`.

`yPos` — Current vertical coordinate in the `scrollRect`.

The `protoHorizontal2DScroller` scroll arrows are handled for you, provided you specify `scrollRect`, `dataRect`, and `viewRect` correctly. If you want to get and set the arrows state, though, you can use the `GetArrow` and `SetArrow` methods, described on page 6-4.

ViewScroll2DScript

ScrollView`:ViewScroll2DScript(`*direction*`, `*extras*`)`

Is called when the user taps the scroll arrows. This method is required.

direction — A symbol indicating the direction to scroll. Use one of the following values: `'left`, `'right`, `'up`, or `'down`.

extras — A frame with the following slots:

- `count` — The number of calls to this method.
- `amount` — The amount scrolled in `scrollRect`.
- `axis` — The axis of scrolling, which is either `'horizontal` or `'vertical`.
- `unit` — The units in which to scroll.

While the pen is held down, the *extras* frame information is reused. This lets you attach state-specific slots to the *extras* frame, which you can reference in subsequent calls to this method.

Note

You usually call `RefreshViews` in your `ViewScroll2DScript`, which forces the view to redraw while the user has the pen down. ◆

ViewScrollDoneScript

scroller:`ViewScrollDoneScript()`

Is called after the user lifts the pen.

SetArrow

scroller:`SetArrow(`*direction*`,` *state*`)`

Is called when the user taps the scroll arrows. Sets the feedback state of an arrow.

direction A symbol indicating the arrow to change. Use one of the following values: `'left`, `'right`, `'up`, or `'down`.

state A symbol indicating the state of the arrow; use one of the following values: `'normal`, `'more`, or `'hilite`.

▲ **WARNING**

Do not set the `scrollRect`, `viewRect`, or `dataRect` slots in your `SetArrow` method. If you do, your changes can conflict with changes that the scroller proto is making. ▲

GetArrow

scroller:`GetArrow(`*direction*`)`

Returns the current state of the arrow direction.

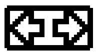
direction A symbol indicating the direction to scroll. Use one of the following: `'left`, `'right`, `'up`, or `'down`.

▲ **WARNING**

Do not set the `scrollRect`, `viewRect`, or `dataRect` slots in your `GetArrow` method. If you do, your changes can conflict with changes that the scroller proto is making. ▲

protoLeftRightScroller

This proto is used to include left/right scrollers, which are centered at the bottom of a view. The following is an example of a `protoLeftRightScroller` view:

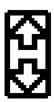


The `viewBounds` and `viewJustify` slots of `protoLeftRightScroller` are set up to center the scroller on the bottom edge of its parent view. Change these slots only if you want the scroller in a different location.

The slots and methods of `protoLeftRightScroller` are the same as those of `protoHorizontal2DScroller`. For their descriptions, see "protoHorizontal2DScroller" (page 6-2).

protoUpDownScroller

This proto is used to include up/down scrollers, centered at the right side of a view. The following is an example of a `protoUpDownScroller` view:



The `viewBounds` and `viewJustify` slots of `protoUpDownScroller` are automatically computed to center the scroller at the right of the view.

The slots and methods of `protoUpDownScroller` are the same as those of `protoHorizontal2DScroller`. For their descriptions, see "protoHorizontal2DScroller" (page 6-2).

protoHorizontalUpDownScroller

This proto is used to include horizontal up/down scrollers, centered at the right side of a view. The following is an example of a `protoHorizontalUpDownScroller` view:

The `protoHorizontalUpDownScroller` automatically centers itself at the bottom of the view; the `viewBounds` and `viewJustify` slots are set up for you.

The slots and methods of `protoHorizontalUpDownScroller` are the same as those of `protoHorizontal2DScroller`. For their descriptions, see "protoHorizontal2DScroller" (page 6-2).

Button and Box Protos

You use the protos described in this section to display text and picture buttons, checkboxes, and radio buttons. The Newton System Software provides a variety of button and box types for use in your applications. Each of these protos uses specific methods to control its behavior, as described in the description of each proto in this section.

For an overview of using the button and box protos in your applications and a description of how to implement a simple button, see "Button and Box Protos" (page 7-6) in *Newton Programmer's Guide*.

protoTextButton

This proto is used to create a rounded rectangle button with text inside it. The text is centered vertically and horizontally within the rectangle. The following is an example of a `protoTextButton` view:

The `ViewClickScript` method is used internally in the `protoTextButton` and should not be overridden. To handle a tap event, use the `ButtonClickScript` method; the `ViewClickScript` method sends the `ButtonClickScript` message to allow you to handle the event.

Note
Inking is automatically turned off when the button is tapped. ◆

The `protoTextButton` is based on a lightweight paragraph view, as described in "Lightweight Paragraph Views" (page 8-11) in *Newton Programmer's Guide*.

Slot descriptions

`viewBounds`	Set to the size and location where you want the button to appear.
`viewFlags`	The default setting is `vVisible + vReadOnly + vClickable`.
`text`	A string that is the text inside the button.
`viewFont`	Optional. The default font for the text is `ROM_fontSystem9Bold`.
`viewFormat`	Optional. The default setting is `vfFillWhite + vfFrameBlack + vfPen(2) + vfRound(4)`.
`viewJustify`	Optional. The default setting is `vjCenterH + vjCenterV + oneLineOnly`. To make a button with

multiple text lines, instead of `oneLineOnly`, use the `noLineLimits` flag.

`viewTransferMode`
Optional. The default transfer mode is `modeOr`.

The following is an example of a template that uses `protoTextButton`. This example prints "ouch" in the Inspector window when the user taps the button:

```
aButton := {...
_proto:     protoTextButton,
text:       "My Button",
ButtonClickScript: func()
   Print("ouch!");

// a handy way to fit a button around a string
ViewSetupFormScript: func()
   viewbounds := RelBounds( 150, 60,
                     StdButtonWidth(self.text), 13 );
...}
```

ButtonClickScript

button:`ButtonClickScript()`

Is called when the button is tapped. The value returned by `ButtonClickScript` is ignored.

ButtonPressedScript

button:`ButtonPressedScript()`

Is called repeatedly as long as the button is pressed (while the pen is held down within it). The value returned by `ButtonPressedScript` is ignored.

protoPictureButton

This proto is used to create a picture that is a button; that is, the user can tap the picture to cause an action to occur. The following is an example of a `protoPictureButton` view:

The `ViewClickScript` method is used internally in the `protoPictureButton` and should not be overridden. To handle a tap event, use the `ButtonClickScript` method; the `ViewClickScript` method sends the `ButtonClickScript` message to allow you to handle the event.

Note
Inking is automatically turned off when the button is tapped. ◆

The `protoPictureButton` is based on a view of the `clPictureView` class.

Slot descriptions

`viewBounds`	Set to the size and location where you want the button to appear.
`viewFlags`	The default is `vVisible + vReadOnly + vClickable`.
`icon`	The bitmap to be used as the button.
`viewFormat`	Optional. The default setting is `vfFillWhite + vfFrameBlack + vfPen(2) + vfRound(4)`. (The examples in the picture above have `viewFormat` set to zero.)
`viewJustify`	Optional. The default setting is `vjCenterH + vjCenterV`.

The following is an example of a template that uses `protoPictureButton`:

```
pictButton := {...
_proto:     protoPictureButton,
icon:       namesBitmap,
viewBounds: SetBounds( 2, 8, 34, 40 ),

ButtonClickScript: func()
   cardfile:Toggle()
...}
```

ButtonClickScript

button:`ButtonClickScript()`

Is called when the button is tapped. The value returned by `ButtonClickScript` is ignored.

ButtonPressedScript

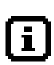
button:`ButtonPressedScript()`

Is called repeatedly as long as the button is pressed (while the pen is held down within it). The value returned by `ButtonPressedScript` is ignored.

protoInfoButton

This proto is used to include the information button in a view. Tapping the information button displays a picker containing information items, which include About, Help, and Prefs. The user can tap one of these items to see more information.

The following views show the information button with and without its picker view displayed:



The `ViewClickScript`, `ViewQuitScript`, `PickActionScript`, and `PickCancelledScript` methods are used internally in the `protoInfoButton` and should not be overridden.

The `protoInfoButton` uses the `protoPictureButton` as its proto.

Slot descriptions

`viewFlags` The default is `vVisible + vReadOnly + vClickable`.

`viewBounds` Optional. Set to the size and location where you want the information button to appear. If you do not set this slot, the information button appears five pixels to the right of its sibling in a 13x13 view. It is designed to be placed next to another button, for example in the status bar.

`viewJustify` Optional. The default setting is `vjParentLeftH + vjParentTopV + vjSiblingRightH + vjSiblingTopV + vjCenterH + vjCenterV`.

DoInfoAbout

button`:DoInfoAbout()`

Is sent to the information button view if the user selects the About menu item. The value returned by `DoInfoAbout` is ignored.

Note

The information button picker only displays the About item if you provide the `DoInfoAbout` method. ◆

DoInfoHelp

button`:DoInfoHelp()`

Is sent to the information button view if the user selects the Help menu item. The value returned by `DoInfoHelp` is ignored.

Note
The information button picker only displays the Help item if you provide the `DoInfoHelp` method. ◆

The Newton System Software also provides a number of functions for displaying help books. The `ShowManual` function, which is described in Chapter 26, "Utility Functions," displays the system Help book. The `OpenHelpTo`, `OpenHelpBook`, and `OpenHelpBookTo` display help books; these functions are described in the *Newton Book Maker User's Guide*.

DoInfoPrefs

button`:DoInfoPrefs()`

Is sent to the information button view if the user selects the Prefs menu item. The value returned by `DoInfoPrefs` is ignored.

Note
The information button picker only displays the Prefs item if you provide the `DoInfoPrefs` method. ◆

GenInfoAuxItems

button`:GenInfoAuxItems()`

Returns an array of items to display in the information button picker. You override this method to define the items you want to appear in the information button picker (the auxilliary items). For more information about the array that you return from this method, see "protoPicker" (page 5-13).

DoInfoAux

button`:DoInfoAux(`*items*`, ` *index*`)`

Is sent to the information button view if the user selects one of the auxiliary items defined by your `GenInfoAuxItems` method.

items — The array of auxiliary items returned by the `GenInfoAuxItems` method.

index — The index of the selected item in the *items* array.

protoOrientation

This proto is available on Newton platforms that support changing the screen orientation so that data on the screen can be displayed facing different ways.

The appearance and operation of this proto varies depending on the type of Newton ROM. On Newton devices with two available orientations—landscape and portrait—this proto presents a `protoTextButton` with the label “Rotate,” which lets the user change between the two modes. On other devices it presents a `protoPopupButton` offering a list of possible orientations.

If you override the default `viewBounds` or `viewJustify` values, you should check the `protoOrientation.viewBounds` value in your `ViewSetupFormScript` method to ensure that the height and width are correct.

When the user changes the orientation, the `screenOrientation` slot of the user configuration that is maintained by the Newton System Software is updated with the selected orientation. In addition, the `ReOrientToScreen` message is sent to all children of the root view; this message is described in “Views” (page 3-1).

Note that the `ButtonClickScript` method is used internally in the `protoOrientation` and should not be overridden.

The `protoOrientation` uses the `protoTextButton` (page 6-7) as its proto.

Slot descriptions

`viewFlags`	The default is `vVisible + vReadOnly + vClickable`.
`viewBounds`	Set to the size and location where you want the orientation button to appear.
`viewJustify`	Optional. The default setting is `vjCenterH + vjParentBottomV + vjParentCenterH`.

protoRadioCluster

This proto is used to group a series of radio buttons into a cluster where only one can be "on" at a time. You must add the individual radio buttons as child views to the radio cluster view.

There is no visual representation of a `protoRadioCluster` view by itself. It serves only as a container for child views based on `protoRadioButton` or `protoPictRadioButton`. See `protoRadioButton` (page 6-16) for an example of what this proto looks like.

The `protoRadioCluster` is based on a view of the `clView` class. The proto itself has no child views; instead, you add individual buttons to the cluster as child views. You can add these buttons, which use either `protoRadioButton` (page 6-16) or `protoPictRadioButton` (page 6-18), by moving the buttons into the cluster.

Slot descriptions

`viewBounds`	Set to the size and location where you want the radio button cluster to appear.
`clusterValue`	Optional. You can specify which button is initially selected by storing its `buttonValue` in this slot. During execution, this slot holds the current value of the radio button cluster by storing the `buttonValue` of the selected radio button. The default initial value is `nil` (no button selected).

The following is an example of a template that uses `protoRadioCluster` and three radio buttons based on `protoRadioButton` (page 6-16):

```
textFaceCluster := {...
_proto:     protoRadioCluster,
viewBounds: SetBounds( 70, 17, 130, 77),
ViewSetupFormScript: func()
   fontFrame := GetUserConfig('userFont);
   clusterValue := fontFrame.face;
ClusterChanged:      func()
   fontFrame.face := clusterValue;
```

```
   SetUserConfig('userFontFace, fontFrame),
...}

child1 :={
_proto:      protoRadioButton,
viewBounds: SetBounds( 0, 0, 60, 20),
text:        "Bold",
buttonValue:'bold
...}

child2 := {...
_proto:      protoRadioButton,
viewBounds: SetBounds( 0, 20, 60, 40),
text:        "Underline",
buttonValue:'underline
...}

child3 := {...
_proto:      protoRadioButton,
viewBounds: SetBounds( 0, 40, 60, 60),
text:        "Plain",
buttonValue:'plain
...}
```

InitClusterValue

cluster`:InitClusterValue(`*buttonValue*`)`

Initializes a radio button cluster. You can pass a button value to set a particular button, or `nil` to initialize the cluster with no buttons set. This method does not send the `ClusterChanged` method.

buttonValue The button value, or `nil` for no buttons set.

ViewSetupFormScript

cluster:`ViewSetupFormScript()`

Sets an initially selected item, the value of which has been calculated at run time. You calculate the value and then set `clusterValue` from within this method.

ClusterChanged

cluster:`ClusterChanged()`

Is called whenever the value of the radio cluster changes (that is, when a different radio button is "turned on") to allow you to perform any necessary processing. The value returned by `ClusterChanged` is ignored.

SetClusterValue

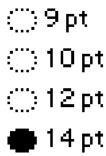
cluster:`SetClusterValue(`*buttonValue*`)`

Programmatically changes the selected radio button in a cluster. This method performs several tasks, including giving the user "undo" capability for the change, updating the screen appropriately, and calling the `ClusterChanged` method.

buttonValue The button value of the button you want to change.

protoRadioButton

This proto creates a radio button child view of a radio button cluster. Radio button clusters are described in "protoRadioCluster" (page 6-14). A radio button is a small oval bitmap that is either empty or contains a solid bull's-eye when it is selected. It is labeled to the right with a text label, as shown in the following view:



The following methods are defined internally: `ViewSetupDoneScript`, `ViewClickScript`, and `RadioClickScript`. If you need to use one of these methods, you must call the inherited method also (for example, `inherited:?ViewSetupDoneScript()`).

Note
Inking is automatically turned off when the button is tapped. ◆

The `protoRadioButton` uses `protoCheckbox` as its proto. For more information, see "protoCheckbox" (page 6-24).

IMPORTANT
A radio button based on `protoRadioButton` must be a child view of a view based on `protoRadioCluster`. You cannot create stand-alone buttons with this proto. ▲

Slot descriptions

`viewBounds`	Set to the size and location where you want the radio button to appear.
`viewFormat`	Optional. The default setting is `vfNone`. You usually do not want any frame or fill because the radio button provides all the required visual information.
`text`	A string that is the radio button text label.
`buttonValue`	The value of the cluster view when this radio button is selected. Each button in the cluster should have a unique `buttonValue`. When this button is selected,

this value is stored in the `clusterValue` slot of the parent radio button cluster.

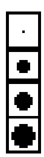
You should use a symbol or immediate for this value, since strings and other structured objects may fail the equivalence test (because internal comparisons are based on pointer equality, not content equality).

`viewValue` The current value of the radio button. When the button is unselected, this is set to `nil`. When the button is selected, this is set to the value in `buttonValue`.

This slot is initialized to `nil`. If you want this button to be initially selected, set `viewValue` to `buttonValue`.

protoPictRadioButton

This proto is used to create a picture radio button child view of a radio button cluster. Radio button clusters are described in “protoRadioCluster” (page 6-14). A picture radio button is a small boxed view that contains a picture. You typically place several of these in a horizontal or vertical row, from which the user chooses one. The following is an example of a vertical picture radio button cluster view:



The following methods are defined internally: `ViewClickScript` and `UpdateBitmap`. If you need to use one of these methods, you must call the inherited method also (for example, `inherited:?ViewClickScript()`).

The `protoPictRadioButton` uses `protoPictureButton` as its proto. For more information, see “protoPictureButton” (page 6-9).

IMPORTANT

A radio button based on `protoPictRadioButton` must be a child view of a view based on `protoRadioCluster`. You cannot create standalone picture buttons with this proto. ▲

Slot descriptions

`viewBounds` Set to the size and location where you want the picture radio button to appear.

`viewFormat` Optional. The default setting is `vfFillWhite + vfFrameBlack + vfPen(2) + vfRound(4)`. To simply frame the view, as shown in the example illustration, use this setting: `vfFillWhite + vfFrameBlack + vfPen(1)`.

`viewJustify` Optional. The default setting is `vjCenterH + vjCenterV`.

`icon` The bitmap to be used as the button picture.

`buttonValue` The value of the cluster view when this picture radio button is selected. Each button in the cluster should have a unique `buttonValue`. When this button is selected, this value is stored in the `clusterValue` slot of the parent radio button cluster.

You should use a symbol for this value, since strings and other structured objects may fail the equivalence test (because internal comparisons are based on pointer equality, not content equality).

`viewValue` The current value of the radio button. When the button is unselected, this is set to `nil`. When the button is selected, this is set to the value in `buttonValue`.

This slot is initialized to `nil`. If you want this button to be initially selected, set `viewValue` to `buttonValue`.

ViewDrawScript

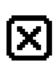
button:ViewDrawScript()

Highlights the radio button when the button is selected. You must supply this method. One style of highlighting is to draw an inner black border, as shown in the following example:

```
pictRadio := {...
_proto:     protoPictRadioButton,
               // override frame
viewFormat: vfFillWhite+vfFrameBlack+vfPen(1),
icon:       myPict,
buttonValue:3,
ViewDrawScript: func()
   begin       // if button is selected then highlight it
   if viewValue then
      :DrawShape(MakeRect(0,0,15,15), nil);
   end,
...}
```

protoCloseBox

This proto allows the user to close the view. This is the close box that you commonly see in views on the Newton screen. When the user taps the close box, the view is closed. The following is an example of a `protoCloseBox`:



Note
The `protoCloseBox` and `protoLargeCloseBox` are similar, with two differences: 1) `protoCloseBox` is a slightly smaller icon, and 2) the frame for `protoCloseBox` is part of the bitmap. ◆

The `protoCloseBox` uses `protoPictureButton` as its proto. For more information, see “protoPictureButton” (page 6-9).

Slot descriptions

`viewFlags` The default is `vVisible + vReadOnly + vClickable`.

`viewBounds` Set to the size and location where you want the close box to appear. If you do not set this slot, the close box defaults to the lower-right corner of its instantiator’s view. (The bitmap is placed at -14, -14 from the lower-right corner.)

`viewJustify` Optional. The default setting is `vjParentBottomV + vjParentRightH`.

`viewFormat` Optional. The default setting is `vfNone`. Typically you don’t use any frame or fill since the close box picture provides the visual content.

IMPORTANT

The view that is to be closed by the close box must contain the following slot:

```
declareSelf: 'base
```

This is usually the application base view. ▲

The following is an example of a template that uses `protoCloseBox`:

```
printerPicker := {...
declareSelf: 'base,
...}

child := {...  // child of printPicker
_proto: protoCloseBox,

ButtonClickScript: func()
   begin
   :closeNetwork();
   inherited:?ButtonClickScript();
```

```
   end,
...}
```

ButtonClickScript

box:`ButtonClickScript()`

Sends the `Close` message to the view identified as `base`. You need to redefine this method if you want to perform additional operations before the view is closed. For example, you might need to close down a communications connection when the view is closed.

If you do redefine this method, you must call the inherited method by sending the message `inherited:?ButtonClickScript()`.

protoLargeCloseBox

This proto is a picture button that contains an “X” icon that allows the user to close the view. When the user taps the icon, the view is closed. The following is an example of a `protoLargeCloseBox` view:

Note
The `protoLargeCloseBox` and `protoCloseBox` (page 6-20) are very similar, with two differences: 1) `protoLargeCloseBox` is a slightly larger icon, and 2) the frame for `protoLargeCloseBox` is not part of the bitmap, but is controlled by the `viewFormat` flags. ◆

The `protoLargeCloseBox` uses `protoPictureButton` as its proto. For more information, see “protoPictureButton” (page 6-9).

Slot descriptions

`viewFlags`	The default is `vVisible + vReadOnly + vClickable`.
`viewBounds`	Set to the size and location where you want the close button to appear. If you do not set this slot, the close button defaults to the lower-right corner of its instantiator's view. (The bitmap is placed at -18, -18 from the lower-right corner.)
`viewJustify`	Optional. The default setting is `vjParentBottomV + vjParentRightH + vjCenterH + vjCenterV`.
`viewFormat`	Optional. The default setting is `vfFillWhite + vfFrameBlack + vfPen(2) + vfRound(4)`.

IMPORTANT

The view that is to be closed by the close box must contain the following slot:

```
declareSelf: 'base
```

This is usually the application base view. ▲

The following is an example of a template that uses `protoLargeCloseBox`:

```
closer := {...
_proto: protoLargeCloseBox,
// no need to define anything else
...}
```

ButtonClickScript

box`:ButtonClickScript()`

Sends the `Close` message to the view identified as `base`. You need to redefine this method if you want to perform additional operations before the view is closed. For example, you might need to close down a communications connection when the view is closed.

If you do redefine this method, you must call the inherited method by sending the message `inherited:?ButtonClickScript()`.

protoCheckbox

This proto is used to create a checkbox, which is a small dotted box that can include a check mark. Each checkbox is labeled to the right with a text label. When the user taps the checkbox, a check is drawn in it. If the user taps a checked box, the check is removed. The following is an example of a checkbox view:

The following methods are defined internally: `ViewSetupDoneScript`, `ViewClickScript`, `ViewChangedScript`, and `UpdateBitmap`. If you need to use one of these methods, you must call the inherited method also (for example, `inherited:?ViewSetupDoneScript()`).

Note
Inking is automatically turned off when the checkbox is tapped. ◆

The `protoCheckbox` implements the checkbox icon portion of the proto. It has one child view, a lightweight paragraph view that implements the text label portion of the proto. Lightweight paragraph views are described in "Lightweight Paragraph Views" (page 8-11) in *Newton Programmer's Guide*.

The `protoCheckbox` is based on the `protoCheckBoxIcon` internal proto, which is based on a view of the `clPictureView` class. The `protoCheckBoxIcon` identifies itself as the `base` view (`declareSelf: 'base`).

Slot descriptions

`viewBounds`	Set to the size and location where you want the checkbox to appear.
`viewFormat`	Optional. The default setting is `vfNone`. You don't typically use any frame or fill since the checkbox provides all the necessary visual content.
`viewFont`	Optional. The default font for the text label is `ROM_fontSystem9`.
`text`	A string that is the checkbox text label.
`buttonValue`	Optional. The value that you want for the view when the checkbox is checked. The default value is non-`nil`. You should use a symbol or immediate value for this value, since strings and other structured objects may fail the equivalence test (because internal comparisons are based on pointer equality, not content equality).
`viewValue`	The current value of the checkbox. When the button is unchecked, this is set to `nil`. When the button is selected, this is set to the value in `buttonValue`.

The following is an example of a template that uses `protoCheckBox`:

```
notifier :={
_proto:         protoCheckBox,
viewBounds:     SetBounds( 40, 30, 200, 45),
buttonValue:    true,
text:           "Play Notify Sound"
...}
```

ValueChanged

checkBox`:ValueChanged()`

Is called whenever the value of the checkbox changes, to allow you to do additional processing. The value returned by `ValueChanged` is ignored.

ToggleCheck

checkBox`:ToggleCheck()`

Programmatically toggles the check mark in the checkbox: if the check mark was displayed, it is erased; if it was not shown, it is displayed. The checkbox is redrawn appropriately. The `ToggleCheck` method always returns non-`nil`.

protoRCheckbox

This proto creates a checkbox with label text to its left. This is exactly like `protoCheckbox`, except that the `protoRCheckbox` places the checkbox to the right of the text, and `protoCheckbox` places the checkbox to the left of the text. The following is an example of a `protoRCheckbox` view:

The following methods are defined internally: `ViewSetupDoneScript`, `ViewClickScript`, `ViewChangedScript`, and `UpdateBitmap`. If you need to use one of these methods, you must call the inherited method also (for example, `inherited:?ViewSetupDoneScript()`).

Note

Inking is automatically turned off when the checkbox is tapped. ◆

The `protoRCheckbox` implements the checkbox icon portion of the proto. It has one child view, a lightweight paragraph view that implements the text label portion of the proto. Lightweight paragraph views are described in “Lightweight Paragraph Views” (page 8-11) in *Newton Programmer’s Guide*.

The `protoRCheckbox` is based on the `protoCheckBoxIcon` internal proto, which is based on a view of the `clPictureView` class. The `protoCheckBoxIcon` identifies itself as the `base` view (`declareSelf: 'base`).

The following is an example of a template that uses `protoRCheckBox`:

```
rightCheckView :={
_proto:         protoRCheckBox,
viewBounds:     SetBounds( 40, 30, 200, 45),
buttonValue:    true,
text:           "Right Checkbox"
...}
```

Slot descriptions

`viewBounds` Set to the size and location where you want the checkbox to appear.

`viewFormat` Optional. The default setting is `vfNone`. You don't typically use any frame or fill since the checkbox provides all the necessary visual content.

`viewFont` Optional. The default font for the text label is `ROM_fontSystem9`.

`text` A string that is the checkbox text label.

`indent` Optional. The number of pixels to indent the checkbox to the right of the text. The default indent is 16.

`buttonValue` Optional. The value that you want for the view when the checkbox is checked. The default value is non-`nil`.

You should use a symbol or immediate value for this value, since strings and other structured objects may fail the equivalence test (because internal comparisons are based on pointer equality, not content equality).

`viewValue` The current value of the checkbox. When the button is unchecked, this is set to `nil`. When the button is selected, this is set to the value in `buttonValue`.

ValueChanged

checkBox`:ValueChanged()`

Is called whenever the value of the checkbox changes, to allow you to do additional processing. The value returned by `ValueChanged` is ignored.

ToggleCheck

checkBox`:ToggleCheck()`

Programmatically toggles the check mark in the checkbox. If the check mark was displayed, it is erased; if it was not shown, it is displayed. The checkbox is redrawn appropriately. The `ToggleCheck` method always returns non-`nil`.

Selection Tab Protos

You can use the protos described in this section to display alphabetic selection tabs on the screen.

For an overview of using the selection tab protos in your applications, see "Selection Tab Protos" (page 7-11) in *Newton Programmer's Guide*.

protoAZTabs

This proto is used to include alphabetical tabs, arranged horizontally, in a view. The following is an example of a `protoAZTabs` view:

PickLetterScript

tabs`:PickLetterScript(`*letter*`)`

Is called when the user taps a tab.

letter — The letter that was tapped.

The following example shows a `pickLetterScript` method:

```
pickLetterScript: func(theLetter)
   begin
   setValue(theText,'text);
   end
```

SetLetter

tabs:`SetLetter(`*newLetter*, *val*`)`

Sets which letter is the currently selected letter and updates the highlighting.

newLetter The letter to select and highlight.

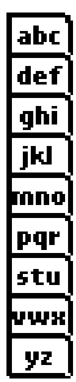
val Must be `nil`. Reserved for future use.

The following example shows a use of the `SetLetter` method:

```
   // set myProtoAZTabs to the letter "C"
myProtoAZTabs:SetLetter($c, nil);
```

protoAZVertTabs

This proto is used to include alphabetical tabs, arranged vertically, in a view. The following is an example of a `protoAZVertTabs` view:



PickLetterScript

tabs:`PickLetterScript(`*letter*`)`

Is called when the user taps a tab.

letter The letter that was tapped.

SetLetter

tabs:`SetLetter(`*newLetter*, *val*`)`

Sets which letter is the currently selected letter and updates the highlighting.

newLetter The letter to select and highlight.

val Must be `nil`. Reserved for future use.

Gauges and Slider Protos

You can use the slider protos described in this section to present a gauge view that indicates the current progress in relation to the entire operation.

For an overview of using the slider protos in your applications and a description of how to implement a simple slider, see "Gauge and Slider Protos" (page 7-12) in *Newton Programmer's Guide*.

clGaugeView

The `clGaugeView` class is used to display objects that look like analog bar gauges.

Note

Although the `clGaugeView` class remains available for compatibility purposes, you should use the `protoGauge` instead. See "protoGauge" (page 6-35) for more information about `protoGauge`. ◆

The following is an example of a `clGaugeView` view:

On interactive gauges, the end of the gauge indicator bar contains a small diamond-shaped active area called a knob. The user can drag the knob to move the indicator bar to a new position.

The following example is a view definition of the `clGaugeView` class:

```
soundGauge := {...
   viewClass:  clGaugeView,
   viewBounds: {left:80, top:20, right:180, bottom:28},
   viewFlags:  vVisible+vClickable,
   gaugeDrawLimits: true,
   minValue:   0,          // must be an integer
   maxValue:   11          // must be an integer
   viewSetupFormScript: func()
         self.viewvalue := GetVolume(),
   viewChangedScript: func(slot, context)
      begin
         SetVolume(viewValue);
         :SysBeep(); //play it so they can hear new level
      end,
...}
```

Slot descriptions

`viewBounds` Set to the size and location where you want the view to appear.

`viewValue` Set this slot to give the gauge an initial value. If you need to calculate the initial value at run time, set this slot in your `ViewSetupFormScript`. This value must be an integer between `minValue` and `maxValue`, inclusive. During execution, the `viewValue` slot stores

	the current gauge setting by interpolating between `minValue` and `maxValue`.
`viewFlags`	The default setting is `vVisible + vClickable`. To make a gauge that is read-only, set the `vReadOnly` flag (and not `vClickable`). For read-only gauges, the diamond-shaped knob is not drawn on the gauge.
`viewFormat`	Optional. The default setting is `nil`.
`minValue`	Optional. The minimum gauge value. This is the value of `viewValue` when the gauge reads fully to the left side. The default is `0`, which you can change if you wish.
`maxValue`	Optional. The maximum gauge value. This is the value of `viewValue` when the gauge reads fully to the right side. The default is `100`, which you can change if you wish.
`gaugeDrawLimits`	Optional. The default is non-`nil`, which displays the gray background. If you set this slot to `nil`, the gray background is not displayed.

ViewChangedScript

view`:ViewChangedScript(`*slot*, *view*`)`

Is called whenever the value of the gauge view changes. This method is called repeatedly as the gauge knob is dragged. You can dynamically track the changes the user is making to the gauge indicator by examining the value of the `viewValue` slot in this method. The value returned by `ViewChangedScript` is ignored.

slot	The name of the slot that changed.
view	The view.

ViewFinalChangeScript

view`:ViewFinalChangeScript(`*valueBefore*, *valueAfter*`)`

Is called after the user lifts the pen from moving the gauge knob. If the user moved the gauge but reset it to its original value, this method is not called. The value returned by `ViewFinalChangeScript` is ignored.

valueBefore	The initial `viewValue` of the gauge before it was changed.
valueAfter	The final `viewValue` of the gauge after it was changed.

protoSlider

This proto is used to create a user-settable gauge view, which looks like an analog bar gauge with a draggable diamond-shaped knob. The following is an example of a `protoSlider` view:

If you want to have a read-only gauge, use the `protoGauge` instead of `protoSlider`.

The following methods are defined internally: `ViewChangedScript` and `ViewFinalChangeScript`. If you need to use one of these methods, be sure to call the inherited method also (for example, `inherited:?ViewChangedScript()`).

▲ **WARNING**
You cannot dynamically change the value of the `minValue` and `maxValue` slots at run time, except within the `ViewSetupFormScript` method. If you need to change the value of these slots, you must close the view, change the values, then reopen the view. ▲

The `protoSlider` uses the `protoGauge` as its proto. For more information, see "protoGauge" (page 6-35).

The following is an example of a template that uses `protoSlider`:

```
SoundSetter := {...
_proto:         protoSlider,
viewBounds:     RelBounds( 12, -21, 65, 9),
viewJustify:    vjParentBottomV,
```

```
maxValue:      4,          // must be an integer

ViewSetupFormScript: func()
   self.viewValue := GetVolume(),

ChangedSlider: func()
   begin
   SetVolume(viewValue);
   :SysBeep();
   end,
...}
```

Slot descriptions

`viewBounds` Set to the size and location where you want the gauge to appear.

`viewValue` Set this slot to give the gauge an initial value. If you need to calculate the initial value at run time, set this slot in your `ViewSetupFormScript`. This value must be an integer between `minValue` and `maxValue`, inclusive. During execution, the `viewValue` slot stores the current gauge setting by interpolating between `minValue` and `maxValue`.

`minValue` Optional. The minimum gauge value. This is the value of `viewValue` when the gauge reads fully to the left side. The default is `0`, which you can change if you wish.

`maxValue` Optional. The maximum gauge value. This is the value of `viewValue` when the gauge reads fully to the right side. The default is `100`, which you can change if you wish.

`gaugeDrawLimits`
Optional. The default is non-`nil`, which displays the gray background. If you set this slot to `nil`, the gray background is not displayed.

ViewSetupFormScript

slider:`ViewSetupFormScript()`

Allows you to perform any processing that is required before the view is instantiated, including setting the initial value of the `viewValue` slot. This method is required; however, you can simply define it as `nil` if you do not need to perform any actions.

ChangedSlider

slider:`ChangedSlider()`

Is called after the user lifts the pen from moving the slider knob. You can access the current gauge setting in the `viewValue` slot. If the user moved the gauge but reset it to its original value, this method is not called. This method is required; however, you can simply define it as `nil` if you do not need to perform any actions. The value returned by `ChangedSlider` is ignored.

TrackSlider

slider:`TrackSlider()`

Is called whenever the value of the `viewValue` slot changes. It is provided so you can dynamically track the changes as the slider is moved and take action based on the current value. The `TrackSlider` method is called repeatedly as the gauge knob is dragged. The value returned by `TrackSlider` is ignored.

protoGauge

You use this proto to create a read-only gauge view, which looks like an analog bar gauge. The following is an example of a `protoGauge` view:

If you want to let the user set the gauge, use the `protoSlider`.

Note that you cannot change the value of the minValue and maxValue slots at run time, except within the ViewSetupFormScript method. If you need to change the value of these slots, you must close the view, change the values, then reopen the view.

The protoGauge is based on a view of the clGaugeView class, which is described in “clGaugeView” (page 6-30).

The following is an example of a template that uses protoGauge:

```
PercentSolvedGauge := {...
_proto:         protoGauge,
viewBounds:     RelBounds( 157, -21, 55, 9),
viewJustify:    vjParentBottomV,
maxValue:       problemsFrame.numberOfProblems,
viewvalue:      problemsFrame.numberSolved,
ViewSetupFormScript: func()
   nil,
...}
```

Slot descriptions

viewBounds	Set to the size and location where you want the gauge to appear.
viewValue	Set this slot to give the gauge an initial value. If you need to calculate the initial value at run time, set this slot in your ViewSetupFormScript. This value must be an integer between minValue and maxValue, inclusive. During execution, the viewValue slot stores the current gauge setting by interpolating between minValue and maxValue.
minValue	Optional. The minimum gauge value. This is the value of viewValue when the gauge reads fully to the left side. The default is 0, which you can change if you wish.
maxValue	Optional. The maximum gauge value. This is the value of viewValue when the gauge reads fully to the right

side. The default is `100`, which you can change if you wish.

`gaugeDrawLimits`
: Optional. The default is non-`nil`, which displays the gray background. If you set this to `nil`, the gray background is not displayed.

ViewSetupFormScript

gauge`:ViewSetupFormScript()`

Allows you to perform any processing that is required before the view is instantiated, including setting the initial value of the `viewValue` slot. This method is required; however, you can simply define it as `nil` if you do not need to perform any actions.

protoLabeledBatteryGauge

This proto is used to create a read-only gauge view that graphically shows the amount of power remaining in the system battery. The gauge is updated every 10 seconds. If the Newton is plugged in and the battery is charging, a charging symbol appears instead of the gauge. The following is an example of a `protoLabeledBatteryGauge` view:

The following methods are defined internally: `ViewSetupDoneScript`, `ViewSetupChildrenScript`, `ViewIdleScript`, and `ReadBattery`. If you need to use one of these methods, you must call the inherited method also (for example, `inherited:?ViewSetupDoneScript()`).

The `protoLabeledBatteryGauge` uses an internal proto, `protoBatteryGauge`, as its prototype. The `protoBatteryGauge` is based on a view of the `clView` class and has two children: the gauge or charging symbol, and the label.

The following is an example of a template that uses `protoLabeledBatteryGauge`:

```
BatteryGauge := {
_proto:     protolabeledbatterygauge,
viewBounds: {left: 58, top: 106, right: 186,
            bottom: 130},
// no other slots needed
};
```

Slot description

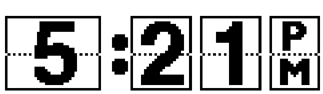
`viewBounds` Set to the size and location where you want the gauge to appear. The gauge fills the entire width of the view.

Time Protos

You can allow the user to specify dates and times with the protos described in this section. For an overview of using the time protos in your applications and a description of how to implement a simple time setter, see “Time Protos” (page 7-14) in *Newton Programmer’s Guide*.

protoDigitalClock

This proto displays a digital clock that can be used to set the time. The user can change the time by tapping each digit. Tapping on the upper part of the digit increments it to the next number; tapping the lower part decrements it. The following is an example of a `protoDigitalClock` view:



Slot descriptions

`viewFlags`	For future compatibility, you must set the `vClickable` flag. (Note, however, that clicks are processed by the children of this proto.)
`viewBounds`	The clock size is fixed at 119x28 pixels.
`viewJustify`	The default setting is `vjParentLeftH + vjParentTopV`.
`increment`	The amount to increment or decrement for each tap. The default is `1`.
`time`	Required. The time to which the clock should be set, expressed in the number of minutes elapsed since midnight, January 1, 1904. When the time is changed, this slot is updated with the currently set time. Note that a `time` slot must be set, either here or somewhere above this proto in the inheritance hierarchy.
`wrapping`	Set to non-`nil` (the default value) to wrap around day boundaries.
`midnite`	Set to non-`nil` if the value `0` should indicate midnight tomorrow (in other words, the end of the current day). The default value is `nil`, which means that `0` indicates midnight today (the beginning of the current day).

Refresh

clock:`Refresh()`

Updates the appearance of the clock. You can call this method when the system time is changed by some external event. For example, if there are two clocks present and the user changes the time in one clock, you should send the `Refresh` message to the second clock.

TimeChanged

clock:`TimeChanged()`

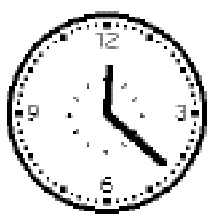
Is called when the time is changed, to allow you to perform any required actions in response to that event. The value returned by `TimeChanged` is ignored.

protoNewSetClock

This proto displays an analog clock that can be used to set the time. There are four ways to change the time with this proto:

- Either hand can be dragged around to the correct position.
- Tapping the rim of the clock changes the minutes. However, if the tap is within two degrees of the location pointed to by the minute hand, it is interpreted as an attempt to drag the minute hand.
- Tapping the inner circle of dots sets the hours. If the tap is within two degrees of the hour hand, it is interpreted as an attempt to drag the hour hand.
- A line can be drawn from the center of the clock face to either the border (to set the minutes) or the inner dial (to set the hours).

The following is an example of a `protoNewSetClock` view:



The following slots and methods are used internally: `ViewSetupFormScript`, `ViewSetupChildrenScript`, `ViewSetupDoneScript`, `ViewDrawScript`, `ViewClickScript`, `tickSound`, `tockSound`, `cuckooSound`, `tickTock`, `hours`, `minutes`, `icon`, `fIconAsShape`, `DrawHand`, `diff`, `FastEnoughAtaN`, `atanTable`, `sinTable`, `distance`.

The `protoNewSetClock` is based on a view of the `clView` class.

Slot descriptions

`viewBounds` Set to the size and location where you want the clock to appear. By default, the bounds are `{left: 30, top: 30, right: 146, bottom: 146}`. The height and width should be equal and a multiple of 29 to make the clock face appear its best.

`viewJustify` The default setting is `vjParentLeftH + vjParentTopV`.

`time` Optional. The time to which the clock should be set, expressed in the number of minutes elapsed since midnight, January 1, 1904. If you don't include this slot, the clock is set to the current time. When the time is changed, this slot is updated with the currently set time.

`annotations` Optional. An array of four strings to be used as minute annotations around the clock face, beginning with the number at the top of the clock and proceeding clockwise. For example, the strings [`"N"`, `"E"`, `"S"`, `"W"`] would decorate the clock like a compass. If you don't specify this slot, the following annotations are used: [`"12"`, `"3"`, `"6"`, `"9"`].

`suppressAnnotations`
Optional. If this slot exists (with any value), the four minute annotations around the clock face are not drawn.

`exactHour` Optional. If non-`nil`, the hour hand clings exactly to the hour markers. If `nil`, the hour hand adjusts between the minutes appropriately, according to the minutes. By default, this is `nil`. You rarely need to set this slot.

Refresh

clock`:Refresh()`

Updates the appearance of the clock. You can call this method when the system time is changed by some external event. For example, if there are two clocks present and the user changes the time in one clock, you should send the `Refresh` message to the second clock.

TimeChanged

clock:`TimeChanged()`

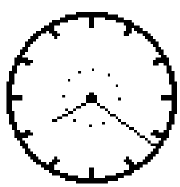
Is called when the time is changed, to allow you to perform any actions required to respond to that event. The value returned by `TimeChanged` is ignored.

protoSetClock

This proto creates an analog clock with which the user can set a time. The user sets the hour by tapping the location in the inner circle where the hour hand should be positioned and the location in the outer circle where the minute hand should be positioned.

Note
The `protoSetClock` has been replaced by the `protoNewSetClock`, which you should use instead. The `protoSetClock` remains available for compatibility of older applications. ◆

The following is an example of a `protoSetClock` view:



The following methods are defined internally: `ViewDrawScript`, `ViewStrokeScript`, `Diff`, `Distance`, `DrawHand`, `DrawHilite`, and `FastEnoughAtan`. If you need to use one of these methods, you must call the inherited method also (for example, `inherited:?ViewDrawScript()`).

The `protoSetClock` is based on a view of the `clPictureView` class.

The following is an example of a template that uses `protoSetClock`:

```
clock := {...
_proto:  protoSetClock,
hours:   nil,  // updated when a new time is set
minutes: nil,  // updated when a new time is set

TimeChanged: func()
   begin
      // do this so the old hands are erased...
   self:Dirty();
      // insert your code in place of the following line
   print("H:" && hours && "M:" && minutes);
   end,

ViewSetupFormScript: func() // show the current time
   begin
   local t     :=Time();
   self.hours  :=(t DIV 60) MOD 24;
   self.minutes:=t MOD 60;
   end,
...};
```

Slot descriptions

`viewBounds`	The clock size is fixed at 64x64 pixels.
`viewFlags`	The default setting is `vVisible + vClickable + vStrokesAllowed`.
`viewFormat`	Optional. The default setting is `vfNone`.
`hours`	Initially set to `nil`. This slot is updated with the new hour when the user sets the hour hand.
`minutes`	Initially set to `nil`. This slot is updated with the new minute time when the user sets the minute hand.

TimeChanged

clock`:TimeChanged()`

Is sent to the view when the user changes the time on the clock. To ensure that the clock redraws properly, you should at least include the following code:

```
self:Dirty()
```

protoAMPMCluster

This proto is used to include A.M. and P.M. radio buttons in a view. The following is an example of a `protoAMPMCluster` view:

The `bounds` must be 70 pixels wide and 15 pixels high. To use this proto, define a parent (a `clView`, for instance), declare a `protoNewSetClock` to the parent as `'setter`, and add a `protoAMPMCluster` to the same parent.

The `protoAMPMCluster` uses `protoRadioCluster` as its proto; `protoRadioCluster` is based on a view of the `clView` class.

The following is an example of a template that uses `protoAMPMCluster`:

```
picker := RDefChild(myTimePopup, 'setter, RDefTemplate( {
   _proto: protoNewSetClock,
   // ...
   }));

RDefChild(myTimePopup, 'ampmButtons, {
   _proto:        protoAMPMCluster,
   viewBounds:    SetBounds(0, 2, 70, 20),
   viewJustify:   vjCenterH
                  + vjSiblingCenterH + vjSiblingBottomV,

   });
```

Slot descriptions

`time` Required. This slot must be set, either here or somewhere above this proto in the inheritance hierarchy.

Special View Protos

You can use the protos in this section to provide special-purpose views in your applications. For an overview of using the special view protos in your applications, see “Special View Protos” (page 7-16) in *Newton Programmer’s Guide*.

protoDragger

This proto creates a view that the user can move around the screen by dragging it with the pen. This view has a rounded matte frame with a small control at the top-center of the frame. The user drags the view by “grabbing” the small control. The view can be dragged only within the bounds of its parent view. The following is an example of a `protoDragger` view:

The proto defines no contents for the draggable view. You need to add your own contents by adding child templates to it.

By default, `protoDragger` does not support scrolling or overview. If you want the view to support scrolling or overview (that is, to handle the scroll arrows and overview button), set the `vApplication` bit in the `viewFlags` slot, and provide the appropriate methods (`ViewScrollUpScript`, `ViewScrollDownScript`, and `ViewOverviewScript`) to handle scroll and overview messages.

The `protoDragger` is based on a view of the `clView` class and does not have any child views.

The following example is a template using `protoDragger`:

```
dragger:= {...
_proto:     protoDragger,
viewBounds: {left:-5, top:104, right:168, bottom:162},
viewJustify:vjParentCenterH,
viewflags:  vVisible+vClickable,
...};

theText := {...  // child of the draggable view
_proto:     protostatictext,
text:       "I'm draggable. . ",
viewBounds: {left:40, top:24, right:144, bottom:48},
viewfont:   simpleFont12+tsBold,
...};
```

Slot descriptions

`viewBounds` Set to the size and location where you want the view to appear.

`viewFlags` The default setting is `vClickable`. Although you can add other view flags, you must not remove `vClickable`.

`viewFormat` Optional. The default setting is `vfFillWhite + vfFrameDragger + vfPen(7) + vfInset(1) + vfRound(5)`.

`noScroll` Optional. This slot holds a message that is used in an error alert if the scroll arrows are tapped and you have not provided a `ViewScrollUpScript` or `ViewScrollDownScript` method to handle the event. This error occurs only if the `vApplication` flag is set for this view (it is not set by default), and it is receiving scroll events. The default message is, “This application

	does not support scrolling," which you can change if you want.
`noOverview`	Optional. This slot holds a message that appears in an error alert if the overview button is tapped and you have not provided a `ViewOverviewScript` method to handle the event. This error occurs only if the `vApplication` flag is set for this view (it is not set by default), and it is receiving overview events. The default message is, "This application does not support Overview," which you can change if you want.

protoDragNGo

This proto is identical to `protoDragger`, except that it includes a close box in the lower-right corner of the view. The following is an example of a `protoDragNGo` view:

The proto defines no contents for the view. You can add your own contents by adding child templates to it.

The `protoDragNGo` is based on the `protoDragger`, which is based on a view of the `clView` class. It is provided with one child view: a close box based on the `protoCloseBox` proto (page 6-20).

The following example is a template using `protoDragNGo`:

```
dragngoView:= {...
_proto:      protoDragNGo,
viewBounds: {left:-2, top:98, right:158, bottom:170},
viewJustify:vjParentCenterH,
viewflags:  vVisible+vClickable,
```

```
...};

theText := {...  // child of the dragngo view
_proto:     protostatictext,
text:       "Drag'n Go view",
viewBounds: {left:22, top:30, right:134, bottom:54},
viewfont:   simpleFont12+tsBold,
...};
```

Slot descriptions

`viewBounds`	Set to the size and location where you want the floater to appear.
`viewFlags`	The default setting is `vClickable`. Although you can add other view flags, you must not remove `vClickable`.
`viewFormat`	Optional. The default setting is `vfFillWhite + vfFrameDragger + vfPen(7) + vfInset(1) + vfRound(5)`.
`noScroll`	Optional. This slot holds a message that appears in an error alert if the scroll arrows are tapped and you have not provided a `ViewScrollUpScript` or `ViewScrollDownScript` method to handle the event. This error occurs only if the `vApplication` flag is set for this view (it is not set by default), and it is receiving scroll events. The default message is, “This application does not support scrolling,” which you can change if you want.
`noOverview`	Optional. This slot holds a message that appears in an error alert if the overview button is tapped and you have not provided a `ViewOverviewScript` method to handle the event. This error occurs only if the `vApplication` flag is set for this view (it is not set by default), and it is receiving overview events. The default message is, “This application does not support Overview,” which you can change if you want.

protoDrawer

This proto creates a view that acts like the Extras Drawer. When the user opens the drawer, the view slides up from the bottom and a drawer-opening sound plays. When the user closes the drawer, a drawer-closing sound plays.

The `protoDrawer` has no content defined. You must add child views to it. This proto is based on a view of the `clView` class.

The following is an example of a template that uses `protoDrawer`:

```
myDrawer := {...
_proto: protoDrawer}  // nothing else needed for drawer
// add children to drawer
```

Slot descriptions

`viewBounds`	Set to the size and location where you want the view to appear.
`viewFlags`	The default setting is `vFloating + vApplication`.
`viewFormat`	Optional. The default setting is `vfPen(2) + vfFrameBlack`.
`viewEffect`	Optional. The default setting is `fxDrawerEffect`.
`showSound`	Optional. The default setting is `ROM_draweropen`.
`hideSound`	Optional. The default setting is `ROM_drawerclose`.

protoFloater

This proto creates a draggable view that floats above all other nonfloating sibling views within an application. This proto is identical to `protoDragger`, except that `protoFloater` is horizontally centered within its parent view, has an opening view effect, and has the `vFloating` flag set in the `viewFlags` slot.

Note
For the base view of an application, it is recommended that you use `protoDragger` instead of `protoFloater`. The floating property interferes with some system services for applications. ◆

The proto defines no contents for the floating view. You can add your own contents to the floater by adding child templates to it.

By default, `protoFloater` does not support scrolling or overview. If you want your floater to support scrolling or overview (that is, to handle the scroll arrows and overview button), set the `vApplication` bit in the `viewFlags` slot, and provide the appropriate methods (`ViewScrollUpScript`, `ViewScrollDownScript`, and `ViewOverviewScript`) to handle scroll and overview messages.

The `protoFloater` is based on the `protoDragger` (page 6-45).

Slot descriptions

`viewBounds`	Set to the size and location where you want the floater to appear.
`viewFlags`	The default setting is `vFloating` + `vClickable`. Although you can add other view flags, you must not remove `vFloating` or `vClickable`.
`viewJustify`	Optional. The default setting is `vjParentCenterH`.
`viewFormat`	Optional. The default setting is `vfFillWhite + vfFrameDragger + vfPen(7) + vfInset(1) + vfRound(5)`.
`viewEffect`	Optional. The default effect is `fxZoomOpenEffect`.
`noScroll`	Optional. This slot holds a message that appears in an error alert if the scroll arrows are tapped and you have not provided a `ViewScrollUpScript` or `ViewScrollDownScript` method to handle the event. This error occurs only if the `vApplication` flag is set for this view (it is not set by default), and it is receiving scroll events. The default message is, “This application does not support scrolling,” which you can change if you want.

`noOverview`	Optional. This slot holds a message that appears in an error alert if the overview button is tapped and you have not provided a `ViewOverviewScript` method to handle the event. This error occurs only if the `vApplication` flag is set for this view (it is not set by default), and it is receiving overview events. The default message is, "This application does not support Overview," which you can change if you want.

protoFloatNGo

This proto is identical to `protoFloater`, except that it includes a close box in the lower-right corner of the floating view.

Note
For the base view of an application, it is recommended that you use `protoDragNGo` instead of `protoFloatNGo`. The floating property interferes with some system services for applications. ◆

The proto defines one child, the close box, for the floating view. You can add your own contents to the floater by adding child templates to it.

The `protoFloatNGo` is based on the `protoFloater` (page 6-49), which is based on the `protoDragger` (page 6-45). It is provided with one child view that is a close box based on the `protoCloseBox` (page 6-20).

Slot descriptions

`viewBounds`	Set to the size and location where you want the floater to appear.
`viewFlags`	The default setting is `vFloating` + `vClickable`. Although you can add other view flags, you must not remove `vFloating` or `vClickable`.
`viewJustify`	Optional. The default setting is `vjParentCenterH`.
`viewFormat`	Optional. The default setting is `vfFillWhite + vfFrameDragger + vfPen(7) + vfInset(1) + vfRound(5)`.

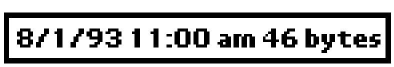
`viewEffect` Optional. The default effect is `fxZoomOpenEffect`.

`noScroll` Optional. This slot holds a message that appears in an error alert if the scroll arrows are tapped and you have not provided a `ViewScrollUpScript` or `ViewScrollDownScript` method to handle the event. This error occurs only if the `vApplication` flag is set for this view (it is not set by default), and it is receiving scroll events. The default message is, “This application does not support scrolling,” which you can change if you want.

`noOverview` Optional. This slot holds a message that appears in an error alert if the overview button is tapped and you have not provided a `ViewOverviewScript` method to handle the event. This error occurs only if the `vApplication` flag is set for this view (it is not set by default), and it is receiving overview events. The default message is, “This application does not support Overview,” which you can change if you want.

protoGlance

This proto creates a text view that closes itself automatically after it has been shown for a brief period of time. The `protoGlance` view also closes immediately if the user taps the view. The following is an example of a `protoGlance` view:



The following methods are defined internally: `ViewSetupDoneScript`, `ViewClickScript`, and `ViewIdleScript`. If you need to use one of these methods, be sure to call the inherited method also (for example, `inherited:?ViewClickScript()`).

The `protoGlance` is based on a lightweight paragraph view, as described in “Lightweight Paragraph Views” (page 8-11) in *Newton Programmer’s Guide*.

A `protoGlance` view is typically hidden until the user performs an action such as tapping a button. After the button is tapped, the `Open` message is sent to the `protoGlance` view to cause it to show itself.

The following example is a template using `protoGlance` (and the button that opens the glance view):

```
myGlance := {...
_proto:            protoglance,
text:              "Just a glance...",
viewIdleFrequency:5000,
viewfont:          ROM_fontSystem9Bold,
viewJustify:       vjCenterV+vjCenterH,
...};

showGlance := {... // Button that opens the glance view
_proto:            prototextbutton,
text:              "Show it",
ButtonClickScript: func()
   myGlance:Open(),
   };
```

Slot descriptions

`viewBounds` Set to the size and location where you want the view to appear.

`viewJustify` Optional. The default setting is `vjCenterV + vjLeftH`.

`viewFormat` Optional. The default setting is `vfFillWhite + vfPen(2) + vfFrameBlack + vfInset(1)`.

`viewFont` Optional. The default font is `ROM_fontSystem9Bold`.

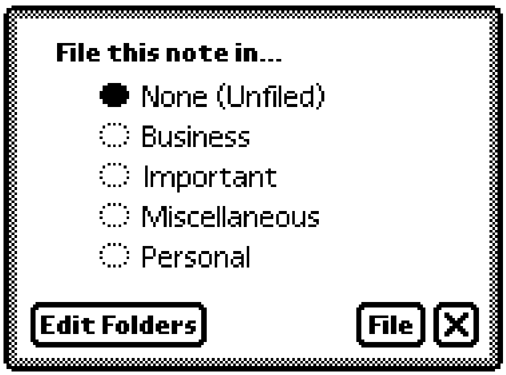
`viewEffect` Optional. The default effect is `fxRight + fxRevealLine`.

`viewIdleFrequency`
Optional. The length of time that the view is to remain open, in milliseconds. The default is `3000` milliseconds (three seconds). Specify an integer greater than zero.

`text`	The text string to display in the view.

protoStaticText

This proto is used for static text. It defines a one-line paragraph view that is read-only and left-justified. The following is an example of a `protoStaticText` view:



The `protoStaticText` is based on a view of the `clParagraphView` class.

The following is an example of a template that uses `protoStaticText`:

```
heading := {...
_proto:         protoStaticText,
viewbounds:     RelBounds( 30, 15, 170, 50),
viewJustify:    vjCenterH+vjTopV,
viewFont:       ROM_fontSystem10,
text:           "Pick your favorite color:",
...}
```

Slot descriptions

`viewBounds`	Set to the location where you want the text to appear.
`viewFlags`	The default setting is `vVisible + vReadOnly`. You do not usually need to change this setting.
`text`	A string that is the text you want to display.

`viewFont`	Optional. The default font is `ROM_fontSystem9Bold`. This slot is ignored if the `styles` slot is present.
`viewJustify`	Optional. The default setting is `vjLeftH + oneLineOnly`.
`viewFormat`	Optional. The default is `vfNone`.
`viewTransferMode`	Optional. The default transfer mode is `modeOr`.
`tabs`	Optional. An array of as many as eight tab-stop positions, in pixels. For example: [`10`, `20`, `30`, `40`].
`styles`	Optional. If multiple font styles are used for the text, this is an array of alternating run lengths and font information. The first element is the run length (in characters) of the first style run, and the second element is the font style of that run. The third element is the run length of the second style run, and so on. All the run lengths must add up to the total text length. If the text is all in a single font, the font in the `viewFont` slot specifies the font style, and the `styles` slot is not needed. For information on how to specify a font in the `styles` array, see the section "Specifying a Font" in Chapter 8, "Text and Ink Input and Display."

View Appearance Protos

You can use the protos described in this section to add to the appearance of your views in certain ways. For an overview of using the view appearance protos in your applications, see "View Appearance Protos" (page 7-18) in *Newton Programmer's Guide*.

protoBorder

This is simply a view filled with black, to use as a border, a line, or a black rectangle. The following is an example of a `protoBorder` view:

The `protoBorder` is based on a view of the `clView` class.

The following is an example of a template that uses `protoBorder`:

```
theBorder := {...
_proto:     protoBorder,
viewBounds: SetBounds( 0, 0, 0, 2 ), // 2-pixel high line
viewJustify:vjParentFullH,
...}
```

Slot descriptions

`viewBounds`	Set to the size and location where you want the border to appear.
`viewFlags`	The default setting is `vVisible`.
`viewFormat`	Optional. The default setting is `vfFillBlack`.

protoDivider

This proto is used to create a divider bar that extends the whole width of its parent view. The divider bar consists of a text string near the left end of a thick line, as shown in the view below:

The `ViewSetupChildrenScript` method is defined internally. If you need to use this method, you must call the inherited method also (for example, `inherited:?ViewSetupChildrenScript()`).

The protoDivider is based on a view of the clView class. It has the following two child views declared in itself:

- divider. This child view uses the protoBorder (page 6-56). It is used for the solid black line.
- dividerText. The text label on the divider bar. This child view is based on a lightweight paragraph view, as described in "Lightweight Paragraph Views" (page 8-11) in *Newton Programmer's Guide*.

The following is an example of a template that uses protoDivider:

```
protoCoverBorder := {...
_proto:       protoDivider,
viewFont:     ROM_fontSystem18Bold,
title:        "COVER SHEET",
}
```

Slot descriptions

viewBounds	Set to the size and location where you want the divider bar to appear. By default, the divider extends the entire width of its parent view (see viewJustify).
viewFlags	The default setting is vVisible + vReadOnly. You do not usually need to change this setting.
viewFont	Optional. The default font is ROM_fontSystem9Bold.
viewJustify	Optional. The default setting is vjParentFullH.
viewFormat	Optional. The default setting is vfNone. In most cases you won't need to change this.
title	A string that is the text on the divider bar.
titleHeight	Optional. The height of the divider view defaults to the height of the font used. For a taller divider view, set this slot to a greater value.

protoTitle

This proto is used to create a title centered at the top of a view. The following view shows a `protoTitle` that has its `titleIcon` slot filled in:

The `ViewSetupFormScript` method is defined internally. If you need to use this method, you must call the inherited method also (for example, `inherited:?ViewSetupFormScript()`).

The `protoTitle` is based on a lightweight paragraph view, as described in "Lightweight Paragraph Views" (page 8-11) in *Newton Programmer's Guide*.

The following is an example of a template that uses `protoTitle`:

```
myTitle := {...
_proto:        protoTitle,
titleHeight:   18,
title:         "Preferences"
...}
```

Slot descriptions

`viewJustify`	Optional. The default setting is `vjParentCenterH + vjParentTopV + vjCenterV + vjCenterH`.
`viewFormat`	Optional. The default setting is `vfFillBlack + vfRound(3)`.
`viewFont`	Optional. The default font is `ROM_fontSystem10Bold`.
`title`	A string that is the title.
`titleIcon`	Optional. A bitmap frame (like the frame returned from `GetPictAsBits`). See *Newton 2.0 User Interface Guidelines* for icon size guidelines.
`titleHeight`	Optional. The height of the title view (black rectangle) defaults to the height of the font used. If you want a taller title view, set this slot to a greater value.

`viewTransferMode`
Optional. The default transfer mode is `modeBic`.

Status Bar Protos

You can use the protos described in this section to display a status bar at the bottom of a view. For an overview of using the status bar protos in your applications, see “Status Bar Protos” (page 7-19) in *Newton Programmer’s Guide*.

protoStatus

This proto is used to create a status bar at the bottom of a view. The status bar includes a large close button at the right side and an analog clock at the left side. If the user taps the analog clock, a digital clock is displayed for three seconds. The following is an example of a `protoStatus` view:

Note
The new status bar protos `newtStatusBarNoClose` and `newtStatusBar`, are the preferred way to add a status bar to your applications. These protos, which are described in “NewtApp Reference” (page 3-1), simplify adding buttons and automate hiding the close box when your application is moved into the background. ◆

The `viewJustify` flags for this view are set so that the status bar always appears at the bottom of its parent view and always occupies the full width of the parent view. Instantiators are not required to set any slots. However, the application base view in which the `protoStatus` view is included must include the following slot:

```
declareSelf: 'base
```

This identifies the view that gets closed when the close box in the status bar is tapped.

The `protoStatus` is based on another proto called `protoStatusBar`, described in the next section, which contains two child views.

The `protoStatus` itself has one child view, the close button that appears at the right side of the view. The close button is based on the `protoLargeCloseBox` (page 6-22).

The following is an example of a template that uses `protoStatus`:

```
theStatus := {_proto: protostatus} // nothing else needed

// base app must include this slot:
declareSelf: 'base
```

protoStatusBar

This proto is used to create a status bar at the bottom of a view. It is identical to `protoStatus`, except that it does not include a close button. The following is an example of a `protoStatusBar` view:

Note
The new status bar protos `newtStatusBarNoClose` and `newtStatusBar`, are the preferred way to add a status bar to your applications. These protos, which are described in "NewtApp Reference" (page 3-1), simplify adding buttons and automate hiding the close box when your application is moved into the background. ◆

The `viewJustify` flags for this view are set so that the status bar always appears at the bottom of its parent view and always occupies the full width of the parent view. Instantiators are not required to set any slots.

The `protoStatusBar` is based on a view of the `clView` class. The `protoStatusBar` contains the following two child views:

- A small round analog clock that appears at the left side of the view. The clock is based on a view of the `clView` class.
- A digital clock display that slides out from the analog clock when the user taps the analog clock. This view is hidden automatically after three seconds. This view is based on the `protoGlance` proto (page 6-52).

Text and Ink Input and Display Reference

This chapter provides reference information for all the constants, data structures, methods, and functions available for your working with text in your applications.

Text Constants and Data Structures

This section describes the constants and data structures you can use in your applications to work with text.

Text Flags

The text flags listed below are view flags you can use to specify information about text in views.

Constant	Value
`vWidthIsParentWidth`	`(1 << 0)`
`vNoSpaces`	`(1 << 1)`
`vWidthGrowsWithText`	`(1 << 2)`
`vFixedTextStyle`	`(1 << 3)`
`vFixedInkTextSTyle`	`(1 << 4)`
`vExpectingNumbers`	`(1 << 9)`

Constant descriptions

`vWidthIsParentWidth`
The view's width is the same as that of its parent view.

`vNoSpaces` Do not insert spaces between words.

`vWidthGrowsWithText`
Causes the right horizontal boundary of the view to extend only as far as the widest line of text in the paragraph. This flag can only be used for paragraph views that are children of an edit view.

`vFixedTextStyle`
The font family, point size, and style of the `viewFont` are applied to all recognized text in the paragraph.

`vFixedInkTextSTyle`
The font point size and style of the `viewFont` are applied to all ink words in the paragraph.

`vExpectingNumbers`
Causes ink words to be scaled based on the assumption that they represent numbers rather than lowercase letters. Use for numeric fields in which the `vNumbersAllowed` flag is not set.

Font Constants for Use in Frames

This section describes the constants you can use to specify fonts in font frames.

Font Family Constants

Use these font family constants to specify the font family in NewtonScript font frames.

Symbol	Font Family
`'espy`	Espy (system) font
`'geneva`	Geneva font
`'newYork`	New York font
`'handwriting`	Casual (handwriting) font

Font Face Constants

Use these font face constants to specify the font face in NewtonScript font frames.

Constant	Value
`kFaceNormal`	`0x000`
`kFaceBold`	`0x001`
`kFaceItalic`	`0x002`
`kFaceUnderline`	`0x004`
`kFaceOutline`	`0x008`
`kFaceSuperScript`	`0x080`
`kFaceSubScript`	`0x100`

Constant descriptions

`kFaceNormal`	Plain font face
`kFaceBold`	Bold font face
`kFaceItalic`	Italic font face
`kFaceUnderline`	Underlined font face
`kFaceOutline`	Outlined font face
`kFaceSuperScript`	Superscripted font face
`kFaceSubScript`	Subscripted font face

Font Constants for Packed Font Integer Specifications

This section describes the constants you can use to specify font information in packed font integer specifications.

Built-in Fonts

The built-in font constants allow you to use a single integer value to specify one of the fonts built into the Newton system, including the font family, font face, and font size.

Constant	Value
`ROM_fontsystem9`	9216
`ROM_fontsystem9bold`	1057792
`ROM_fontsystem9underline`	4203520
`ROM_fontsystem10`	10240
`ROM_fontsystem10bold`	1058816
`ROM_fontsystem10underline`	4204544
`ROM_fontsystem12`	12288
`ROM_fontsystem12bold`	1060864
`ROM_fontsystem12underline`	4206592
`ROM_fontsystem14`	14336
`ROM_fontsystem14bold`	1062912

Constant	Value
ROM_fontsystem14underline	4208640
ROM_fontsystem18	18432
ROM_fontsystem18bold	1067008
ROM_fontsystem18underline	4212736
simpleFont9	9218
simpleFont10	10242
simpleFont12	12290
simpleFont18	18434
fancyFont9 or userFont9	9217
fancyFont10 or userFont10	10241
fancyFont12 or userFont12	12289
fancyFont18 or userFont18	18433
editFont10	10243
editFont12	12291
editFont18	18435

Constant descriptions

`ROM_fontsystem9`
: 9-point, plain face, Espy font

`ROM_fontsystem9bold`
: 9-point, boldface, Espy font

`ROM_fontsystem9underline`
: 9-point, underline face, Espy font

`ROM_fontsystem10`
: 10-point, plain face, Espy font

`ROM_fontsystem10bold`
: 10-point, boldface, Espy font

`ROM_fontsystem10underline`
: 10-point, underline face, Espy font

`ROM_fontsystem12`
: 12-point, plain face, Espy font

`ROM_fontsystem12bold`
: 12-point, boldface, Espy font

`ROM_fontsystem12underline`
: 12-point, underline face, Espy font

`ROM_fontsystem14`
: 14-point, plain face, Espy font

`ROM_fontsystem14bold`
: 14-point, boldface, Espy font

`ROM_fontsystem14underline`
: 14-point, underline face, Espy font

`ROM_fontsystem18`
: 18-point, plain face, Espy font

`ROM_fontsystem18bold`
: 18-point, boldface, Espy font

`ROM_fontsystem18underline`
: 18-point, underline face, Espy font

`simpleFont9` 9-point, plain face, Geneva font

`simpleFont10` 10-point, plain face, Geneva font

`simpleFont12` 12-point, plain face, Geneva font

`simpleFont18` 18-point, plain face, Geneva font

`fancyFont9 (userFont9)`
: 9-point, plain face, New York font

`fancyFont10 (userFont10)`
: 10-point, plain face, New York font

`fancyFont12 (userFont12)`
: 12-point, plain face, New York font

`fancyFont18 (userFont18)`
: 18-point, plain face, New York font

`editFont10` 10-point, plain face, handwriting font

`editFont12` 12-point, plain face, handwriting font

`editFont18` 18-point, plain face, handwriting font

Font Family Constants

Use these font family constants to specify the family ID in a packed integer font specification.

Constant descriptions

(none)	The Espy (system) font
`tsFancy`	The New York font
`tsSimple`	The Geneva font
`tsHWFont`	The Casual (handwriting) font

Font Face Constants for Packed Integer Font Specifications

Use these font face constants to specify the font face in a packed integer font specification.

Constant	Value
`tsPlain`	`0`
`tsBold`	`1048576`
`tsItalic`	`2097152`
`tsUnderline`	`4194304`
`tsOutline`	`8388608`
`tsSuperScript`	`134217728`
`tsSubScript`	`268435456`

Constant descriptions

`tsPlain`	Plain font face
`tsBold`	Bold font face
`tsItalic`	Italic font face
`tsUnderline`	Underlined font face
`tsOutline`	Outlined font face
`tsSuperScript`	Superscripted font face
`tsSubScript`	Subscripted font face

Keyboard Constants

This section describes the constants you can use with keyboard views.

Keyboard Registration Constants

When you register a keyboard, you can specify these flags to define how the keyboard is used.

Constant	Value
`kKbdUsesKeyCodes`	1
`kKbdTracksCaret`	2
`kKbdforInput`	4

Constant descriptions

`kKbdUsesKeyCodes`
: The keyboard is key code–based, which means that the system has to redraw the view whenever the Shift, Option, or another modifier key is pressed on this or any other key code–based view. This is because a single key map is used for all keyboard views.

`kKbdTracksCaret`
: The `ViewCaretChangedScript` method of the keyboard view is called whenever the caret changes position.

`kKbdforInput`
: The insertion caret is activated when this keyboard opens, if the caret was not already active. Use this when your keyboard provides input capabilities.

Key Descriptor Constants

The key descriptor constants specify the appearance of each key in a keyboard.

Constant	Value
keySpacer	(1 << 29)
keyAutoHilite	(1 << 28)
keyInsetUnit	(1 << 25)
keyFramed	(1 << 23)
keyRoundingUnit	(1 << 20)
keyLeftOpen	(1 << 19)
keyTopOpen	(1 << 18)
keyRightOpen	(1 << 17)
keyBottomOpen	(1 << 16)
keyHUnit	(1 << 11)
keyHHalf	(1 << 10)
keyHQuarter	(1 << 9)
keyHEighth	(1 << 8)
keyVUnit	(1 << 3)
keyVHalf	(1 << 2)
keyVQuarter	(1 << 1)
keyVEighth	(1 << 0)

Constant descriptions

keySpacer	Nothing is drawn in this space; it is a spacer, not a key.
keyAutoHilite	Highlight this key when it is pressed.
keyInsetUnit	Inset this key's frame a certain number of pixels within its space. Multiply this constant by the number of pixels you want to inset, from 0–7.

`keyFramed`	The thickness of the frame around the key. Multiply this constant by the number of pixels you want to use for the frame thickness, a value in the range 0-3.
`keyRoundingUnit`	The roundedness of the frame corners. Multiply this constant by the number of pixels you want to use for the corner radius, from 0-15, zero being square.
`keyLeftOpen`	No frame line is drawn along the left side of this key.
`keyTopOpen`	No frame line is drawn along the top side of this key.
`keyRightOpen`	No frame line is drawn along the right side of this key.
`keyBottomOpen`	No frame line is drawn along the bottom side of this key.
`keyHUnit`	Used in a key dimensions formula to specify horizontal units.
`keyHHalf`	Defines a number of half-units.
`keyHQuarter`	Defines a number of quarter-units.
`keyHEighth`	Defines a number of eighth-units.
`keyVUnit`	Used in a key dimensions formula to specify vertical units.
`keyVHalf`	Defines a number of half-units.
`keyVQuarter`	Defines a number of quarter-units.
`keyVEighth`	Defines a number of eighth-units.

Note

See “Defining Keys in a Keyboard View” (page 8-30) in *Newton Programmer’s Guide* for more information about the `keyHUnit`, `keyHHalf`, `keyHQuarter`, `keyHEighth`, `keyVUnit`, `keyVHalf`, `keyVQuarter`, and `keyVEighth` constants. ◆

Keyboard Modifier Keys

Use the keyboard modifier key constants to determine which modifier keys have been pressed or when a character is “delivered” from a keyboard.

Constant	Value
`kIsSoftKeyboard`	`(1 << 24)`
`kCommandModifier`	`(1 << 25)`
`kShiftModifier`	`(1 << 26)`
`kCapsLockModifier`	`(1 << 27)`
`kOptionsModifier`	`(1 << 28)`
`kControlModifier`	`(1 << 29)`

Constant descriptions

`kIsSoftKeyboard`
: If `true`, the character was entered on a “soft” keyboard; if not, the character was entered on an external keyboard.

`kCommandModifier`
: If `true`, the Command key was in effect.

`kShiftModifier`
: If `true`, the Shift key was in effect.

`kCapsLockModifier`
: If `true`, the Caps Lock key was in effect.

`kOptionsModifier`
: If `true`, the Option key was in effect.

`kControlModifier`
: If `true`, the Control key was in effect.

Line Patterns

A line pattern, which you use for customizing the display of the ruling lines in an edit or paragraph view, is an 8-byte binary data structure with the class `'pattern`.

The bit pattern of the bytes defines which pixels are turned on in the line. A typical line pattern is defined as shown here:

```
myPattern := SetClass( Clone("\uAAAAAAAAAAAAAAAA"),
                             'pattern );
```

This code clones a string, which is already a binary object, and changes its class to `'pattern`. The string is specified with hex character codes whose binary representation creates the pattern. Each 2-digit hex code creates one byte of the pattern.

When the line is drawn, the first bit of the pattern is aligned with the first pixel of the line. The pattern is repeated as necessary.

The Rich String Format

The rich string format lets you embed ink data in a text string. The location of each ink word in the string is indicated by a placeholder character (`0xF700` or `0xF701`), and the data for each ink word is stored after the string terminator character at the end of the string. The final 32 bits in a rich string also have special meaning.

Text Views and Protos

This section describes the views and protos that you can use to display text and receive text input.

General Input View (clEditView)

The `clEditView` class is used to accept text input. The `clEditView` class contains no data. When it receives input it creates child views—a `clParagraphView` to hold text or ink text and a `clPolygonView` to hold graphics or raw ink.

You can also add pictures to `clEditView` views. To add a picture to a `clEditView`, you need to create an appropriate template and add that template to the view's `viewChildren`.

For a list of the features provided by `clEditView`, see "General Input Views" (page 8-6) in *Newton Programmer's Guide*. The same section provides an example of a template that defines a view of the `clEditView` class.

Slot descriptions

`viewBounds` Set to the size and location where you want the view to appear.

`viewFlags` The default setting is `vVisible`. You will most likely want to set additional flags to control the recognition behavior of the view.

`viewFormat` Optional. The default setting is: `vfFillWhite + vfFrameBlack + vfPen(1)`.

`viewLineSpacing`
Sets the spacing between the lines, in pixels.

`viewLinePattern`
Optional. Sets a custom pattern that is used to draw the lines in the view. In the `viewFormat` slot editor in NTK, you must also set the Lines item to Custom to signal that you are using a custom pattern. (This sets the `vfCustom<<vfLinesShift` flag in the `viewFormat` slot.)

Patterns are binary data structures, which are described in "Line Patterns" (page 7-11).

A view of this class can appear as a blank space. Normally, you want the view to contain a series of horizontal dotted lines, like lined writing paper, to show that the view accepts input. For information on how to create this effect, see "Creating the Lined Paper Effect in a Text View" (page 8-8) in *Newton Programmer's Guide*.

Child views that are automatically created by a `clEditView` have the `vNoScripts` flag set in their `viewFlags` slot, as described in "System

Messages in Automatically Created Views" (page 8-8) in *Newton Programmer's Guide*.

Functions and Methods for Edit Views

This section describes the messages that are sent to edit views. You can define methods for these messages.

EditAddWordScript

view:`EditAddWordScript(`*form*, *bounds*`)`

Is sent to an edit view when a new paragraph is about to be added to the edit view.

form	The paragraph template that is about to be added to the edit view.
bounds	The bounds of the written ink or typewritten character that has caused the new paragraph to be added.

You can use this script to modify the paragraph that is about to be added to the edit view. Your method must return the template to be added.

If you do not provide this method, or if you return *form* unchanged, the default action is taken: the system adds the paragraph view to the edit view in the usual manner at the usual location.

NotesText

`NotesText(`*childArray*`)`

Returns a string that represents all of the text in an edit view.

childArray	An array of child views of an edit view. You should use the `editView.viewChildren` slot.

The `NotesText` function creates a string in which distinct paragraphs are separated by carriage return characters. The `NotesText` function uses the location of each child view within the edit view to determine the order in which the strings are output.

If any of the child views contains ink, `NotesText` returns a rich string. If none of the views contains ink, `NotesText` returns a plain string.

You can use the `NotesText` function to export edit view text to a non-Newton computer or e-mail system.

Paragraph View (clParagraphView)

The `clParagraphView` class displays text or accepts text input. For a list of the features provided by `clParagraphView`, see "Paragraph Views" (page 8-10) in *Newton Programmer's Guide*. The same section provides an example of a template that defines a view of the `clParagraphView` class.

Slot descriptions

`viewBounds`	Set to the size and location where you want the view to appear.
`text`	A string that is the text currently contained in the view.
`viewFont`	Required, unless the `styles` slot is specified. The `viewFont` slot sets the font used to display text in the view. Note that if the view template itself does not contain this slot, it is inherited through proto inheritance only, not parent inheritance. See "Using Fonts for Text and Ink Display" (page 8-17) in *Newton Programmer's Guide* for a detailed description of how to specify a font. If the text in the view has multiple fonts, the `styles` slot is used to specify the font, instead of the `viewFont` slot.
`viewFlags`	The default setting is `vVisible`. You will most likely want to set additional flags to control the recognition behavior of the view. See the discussion of recognition flags in "Recognition" (page 9-1) in *Newton Programmer's Guide*.
`viewFormat`	Optional. The default setting is `vfFillWhite+vfFrameBlack+vfPen(1)+vfLinesGray`.
`viewJustify`	Optional. The default setting is `vjLeftH+vjTopV+vjParentLeftH+vjParentTopV+noLineLimits`.

Note that this view class does not support vertical justification of the view text for multiline text views. Therefore, the vertical justification flags (`vjCenterV`, `vjBottomV`, and `vjFullV`) apply only if the `oneLineOnly` flag is also set.

`tabs` Optional. An array of up to eight tab-stop positions, in pixels. For example: [10, 20, 30, 40]. These positions are pixel values, relative to the left boundary of the view.

`styles` Optional. An array of alternating run lengths and font information, if multiple font styles are used. The first element is the run length (in characters) of the first style run, and the second element is the font style of the first run. The third element is the run length of the second style run, and so on. For ink words, the length value is always `1`, and the style specification is a binary object that contains the ink data. All of the run lengths must add up to the total text length. If the text is all in a single font, the font in the `viewFont` slot specifies the font style, and the `styles` slot is not needed. For information on how to specify a font in the `styles` array, see "Text and Styles" (page 8-25) in *Newton Programmer's Guide*.

`textFlags` Optional. Can contain one or more of the text flags described in "Text Flags" (page 7-2).

`copyProtection` Specifies restrictions on copying the view by dragging it into another view or by using the clipboard. This slot applies only to views of the class `clParagraphView`. If this slot is not present, there are no copy restrictions. In this slot you can specify one or more copy protection attributes, which are represented by constants defined as bit flags. The copy protection attributes are listed and described in Table 7-1.

Table 7-1 `CopyProtection` constants

Constant	Value	Description
`cpNoCopies`	1	The view cannot be copied.
`cpReadOnlyCopies`	2	The view can be copied, but the copy cannot be modified.
`cpOriginalOnlyCopies`	4	The original view can be copied, but copies of it cannot. When a copy is made, the its `copyProtection` slot is changed to `1` (`cpNoCopies`) to prevent further copying.
`cpNewtonOnlyCopies`	8	The view can be copied, but on one Newton device only. Copies cannot be exported to a different Newton device.

Input Line Protos

An input line is just that, a single line in which the user can enter data. Protos are provided for input lines with and without an identifying label, and for regular and rich-text input. The use of input line protos is described in "Using Input Line Protos" (page 8-12) in *Newton Programmer's Guide*.

protoInputLine

This proto is used for a one-line input field that is indicated by a dotted line to write on. It defines a simple paragraph view that accepts any kind of text input and is left-justified, as described in "protoInputLine" (page 8-12) in *Newton Programmer's Guide*. The same section provides an example of a template using `protoInputLine`.

Slot descriptions

`viewBounds` — Set to the location where you want the input field to appear.

`viewFlags` — Set particular view flags to limit recognition, if desired. The default setting is `vVisible + vClickable + vGesturesAllowed + vCharsAllowed + vNumbersAllowed`. For more information about the recognition view flags, see "Recognition" (page 9-1) in *Newton Programmer's Guide*.

`text` — Optional. Set to a string that is the initial text, if any, to be shown in the input field. The default is no text. During run time, this slot holds the current text that exists in the input field.

`viewFont` — Optional. This sets the font for text the user writes in the input field. The default is `editFont12`.

`viewJustify` — Optional. The default setting is `vjLeftH + oneLineOnly`.

`viewFormat` — Optional. The default setting is `vfLinesGray`.

`viewTransferMode` — Optional. The default mode is `modeOr`.

`viewLineSpacing` — Optional. The line spacing is the height of the input line in pixels and it defaults to the setting of the parent view, or to 20, if there is no parent setting.

`viewLinePattern` — Optional. Sets a custom pattern for drawing the line in the view. A pattern is an 8-byte binary data structure with the class `'pattern`. For information about specifying a line pattern, see "Defining a Line Pattern" (page 8-9) in *Newton Programmer's Guide*.

`viewChangedScript` — Optional. This method is called whenever the value of the input field is changed.

`memory` — Used to reference a list of the last *n* items chosen. The value of this slot is a symbol that names the list. The symbol must incorporate your developer signature, as described in “protoLabelInputLine” (page 8-13) in *Newton Programmer’s Guide*.

The following additional methods are defined internally: `ViewSetupFormScript` and `ViewSetupDoneScript`. 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.

protoRichInputLine

This proto is the text and ink equivalent of the `protoInputLine`. The slot descriptions and discussion are exactly the same as for `protoInputLine`.

protoLabelInputLine

This proto is used for a one-line input field that includes a text label and can optionally feature a pop-up menu. See “protoLabelInputLine” (page 8-13) in *Newton Programmer’s Guide* for a description of how to use this proto.

Slot descriptions

`viewBounds` Set to the location where you want the view to appear. Note that the view should have a height equal to or greater than the value set for `viewLineSpacing`.

`entryFlags` Set particular flags to limit recognition, if desired. The setting you specify in this slot is used for the `viewFlags` slot of the input field. The default setting is `vVisible + vClickable + vGesturesAllowed + vCharsAllowed + vNumbersAllowed`. For more information about the recognition view flags, see “Recognition” (page 9-1) in *Newton Programmer’s Guide*.

`label` Set to a string that is the label text.

`labelFont` Optional. Sets the font used for the label. The default is `ROM_fontSystem9Bold`.

`labelCommands` Optional. If this slot is supplied, the picker feature is activated and the label is shown with a diamond to its left to indicate that it is a picker.

Specify an array of strings that should appear in a picker when the user taps the label. To include a thin, gray separator line, specify the symbol `'pickSeparator`. For a thicker black line, specify the symbol `'pickSolidSeparator`.

The currently selected item in the list, if there is one, is marked with a check mark to its left.

`curLabelCommand`
: Optional. If the `labelCommands` slot is supplied, this slot specifies which item in that array should be initially marked with a check mark. Specify an integer, which is used as an index into the `labelCommands` array. If you omit this slot, no item is initially marked with a check mark.

`indent`
: Optional. Set to the distance from the left edge of the view where the dotted input line should begin. The default is 4 pixels to the right of the label text. This slot is useful if you are specifying several labeled input fields in a column, and want all the dotted input lines to line up beneath one another. If you specify this slot, be sure to leave enough room for the label text.

`viewLineSpacing`
: Optional. The line spacing is the height of the input line in pixels and it defaults to the setting of the parent view, or to 20, if there is no parent setting.

`viewLinePattern`
: Optional. Sets a custom pattern that is used to draw the line in the view. A pattern is an 8-byte binary data structure with the class `'pattern`. For information about specifying a line pattern, see "Defining a Line Pattern" (page 8-9) in *Newton Programmer's Guide*.

`textSetup`
: Optional. This method is called when the view is instantiated to set an initial value in the input field. This method is passed no parameters and should return a string, which is set as the initial value in the input field.

If you don't supply this method, the input field is initially empty.

`updateText` Optional. You can call this method to programmatically change the value of the text in the input field. This action is reversible by the user with the Undo button. This method takes one parameter, a string that is the new value of the input field. Note that you don't normally need to call this method; the input field is updated automatically when the user writes in it.

`textChanged` Optional. This method is called whenever the value of the input field is changed. It is passed no parameters. If you don't supply this method, no default action occurs.

`setLabelText` Optional. You can call this method to dynamically change the label text after the view has already been opened. This method takes one parameter, a string that is the new label text.

`setLabelCommands`

Optional. You can call this method to dynamically set the `labelCommands` array. This method takes one parameter, an array of strings that should appear in the picker.

`labelClick` Optional. This method is called when the user taps the label. It is passed one parameter, the stroke unit that was passed to the `viewClickScript` method of the label. This message notifies the view, which gives you a chance to handle the event when the label is tapped. If you don't supply this method or choose not to handle the event, the default action is to display the picker, get the user's choice, enter the chosen text into the input line, and dirty the input line to cause a redraw.

This function must return either `true` or `nil`. If it returns `true`, the default action is not finished; the assumption is that you have handled the event yourself. If it returns `nil`, the default action is still performed after this method returns.

`labelActionScript`

Optional. This method is called when an item is chosen from the picker. It is passed one parameter, which is the index of the item selected from the `labelCommands` array. This message notifies the view, which gives you a chance to handle the event when an item is chosen from the picker. If you don't supply this method or choose not to handle the event, the default action is to set the text in the input line to the string that was chosen from the picker.

This function must return either `true` or `nil`. If it returns `true`, the default action is not finished; the assumption is that you have handled the event yourself. If it returns `nil`, the default action is still performed after this method returns.

Note that inking is automatically turned off when the label is tapped.

The `protoLabelInputLine` is based on a view of the `clView` class, and includes two child views: `labelLine` and `entryLine`. These views are described in "protoLabelInputLine" (page 8-13) in *Newton Programmer's Guide*.

protoRichLabelInputLine

This proto is the text and ink equivalent of the `protoLabelInputLine`. The slot descriptions and discussion are exactly the same as for `protoLabelInputLine`.

Text and Ink Display Functions and Methods

This section describes the functions and methods you can use in applications to display text and ink in views. For more information, see "Text and Ink in Views" (page 8-14) in *Newton Programmer's Guide*.

Functions and Methods for Measuring Text Views

This section describes the functions you can use to measure or predict the bounds of a text view.

There are two measurement functions: `TextBounds` and `TotalTextBounds`. The `TextBounds` function is more efficient, but is accurate only in limited circumstances. You can use the `TextBounds` function if the view meets the following conditions:

- it contains no tabs
- it uses a single font
- it uses fixed line spacing

If your view does not meet these conditions, use the `TotalTextBounds` function for measuring the bounds of the view.

TextBounds

`TextBounds(`*rStr*, *fontFrame*, *viewBounds*`)`

Computes the bounds of a text string within a view.

rStr — A string or rich string that does not contain any tabs or line breaks.

fontFrame — Either a standard font specification, or a frame that contains the following two slots:

- `font` — A font specification.
- `justification` — Optional. The text justification, which must be : `'left`, `'center`, or `'right`. The default value is `'left`.

viewBounds — A bounds frame in which either the `right` or `bottom` slot has a value of `0`.

The `TextBounds` function computes the bounds frame for a text string that is drawn using the supplied font specification. The `TextBounds` function modifies the slots in `viewBounds` to specify the bounds for *rStr*.

If the `right` value of the original bounds frame is 0, `TextBounds` computes how wide the bounds box needs to be for the text to fit into a specified height value, and stores that value into the `right` slot.

If the `bottom` value of the original bounds frame is `0`, `TextBounds` computes how tall the bounds box needs to be for the text to fit into a specified width value, and stores that value into the `bottom` slot.

If both the `right` and `bottom` values of the original bounds frame are `0`, the `width` and `height` slots are modified based on the explicit line breaks in *rStr*.

TotalTextBounds

`TotalTextBounds(`*paraSpec*`, `*editSpec*`)`

Predicts the bounds of a complex paragraph view, based on the text in the view.

paraSpec A paragraph view template that must contain the following slots: `text`, `viewFont`, and `viewBounds`. The `bottom` slot in `viewBounds` should have a value of `0.`

editSpec A template for the edit view in which the paragraph is to be enclosed. This can be `nil`.

You should include this parameter if you are going to create a paragraph view as the child of a `clEditView`, since the properties of the edit view affect the computation.

The `TotalTextBounds` function returns a bounds frame for a `clParagraphView` that encloses the specified text. The returned bounds frame contains the same `left`, `right`, and `top` values as the `viewBounds` slot of *paraSpec*. The `bottom` slot of the returned bounds frame is filled in with the appropriate height value for the paragraph view.

The text slot of the paragraph view can contain plain strings or rich strings.

Functions and Methods for Determining View Ink Types

This section describes the functions and methods you can use to determine whether a view accepts raw ink or ink words as input.

AddInk

`AddInk(`*edit*`,` *poly*`)`

Adds ink to an edit view.

edit — An edit view object.

poly — A polygon frame that can be expanded into a `clPolygonView` object. This frame contains two slots:

- `ink` — The ink data.
- `viewBounds` — The bounds box for the ink.

The `AddInk` function adds ink to an edit view. The ink is stored within the edit view as a polygon view.

ViewAllowsInk

`ViewAllowsInk(`*view*`)`

Determines if *view* accepts raw ink as input.

view — A view object.

The `ViewAllowsInk` function returns a non-`nil` value if view accepts raw ink as input. This function uses the view's recognition configuration and view flags to determine the return value.

Note
The value returned by the `ViewAllowsInk` function is not necessarily the same as the state of the Recognition menu. This is because a view that does not receive ink due to the Recognition menu setting can allow ink. ◆

ViewAllowsInkWords

`ViewAllowsInkWords(`*view*`)`

Determines if *view* accepts raw ink as input.

view A view object.

The `ViewAllowsInkWords` function returns a non-`nil` value if the view accepts ink words as input. This function uses the view's recognition configuration and view flags to determine the return value.

Font Attribute Functions and Methods

You can use the font attribute functions and methods to store or retrieve the settings stored in a font specification. For more information about using fonts in your text views, see "Using Fonts for Text and Ink Display" (page 8-17) in *Newton Programmer's Guide*.

FontAscent

`FontAscent(`*fontSpec*`)`

Returns the ascent, in pixels, of the font specified by *fontSpec*. The ascent is the vertical distance from the font baseline to the font ascent line.

fontSpec A font specification. This can be an integer, frame, or binary object specification of a font.

FontDescent

`FontDescent(`*fontSpec*`)`

Returns the descent, in pixels, of the font specified by *fontSpec*. The descent is the vertical distance from the font baseline to the font descent line.

fontSpec A font specification. This can be an integer, frame, or binary object specification of a font.

FontHeight

`FontHeight(`*fontSpec*`)`

Returns the maximum height, in pixels, of the font specified by *fontSpec*. This equals the font ascent plus the descent plus the leading.

fontSpec	A font specification. This can be an integer, frame, or binary object specification of a font.

FontLeading

`FontLeading(`*fontSpec*`)`

Returns the font leading, in pixels, of the font specified by *fontSpec*. This is the vertical distance from the font descent line to the ascent line of the next text line below it.

fontSpec	A font specification. This can be an integer, frame, or binary object specification of a font.

GetFontFace

`GetFontFace`*(fontSpec)*

Returns the face of the font specified by *fontSpec*. The face is returned as an integer value.

fontSpec	A font specification. This can be an integer, frame, or binary object specification of a font.

GetFontFamilyNum

`GetFontFamilyNum`*(fontSpec)*

Returns the family number for the font specified by *fontSpec*. Only the Espy, Geneva, Handwriting (Casual), and New York font families currently have numbers.

Returns `nil` if no number is available or if the font is an ink font.

fontSpec	A font specification. This can be an integer, frame, or binary object specification of a font.

GetFontFamilySym

`GetFontFamilySym`*(fontSpec)*

Returns the symbol representing the typeface of the font specified by *fontSpec*. The returned value is one of the font family symbols, as shown in Table 8-3 (page 8-18) in *Newton Programmer's Guide*.

Returns `nil` if the *fontSpec* is an ink font binary object.

fontSpec	A font specification. This can be an integer, frame, or binary object specification of a font.

GetFontSize

`GetFontSize`*(fontSpec)*

Returns the size of the font specified by *fontSpec*.

fontSpec	A font specification. This can be an integer, frame, or binary object specification of a font.

MakeCompactFont

`MakeCompactFont`*(family, size, face)*

Makes a new font specification from the supplied components.

family	Can be either a symbol or integer that specifies a font family.
size	The point size as an integer value.
face	The font face as an integer value.

Returns a font specification. If the font is a ROM font, a packed integer is returned.

SetFontFace

`SetFontFace`*(fontSpec, newFace)*

Sets the face of the font specified by *fontSpec* to the face specified by *newFace* and returns the altered *fontSpec*.

fontSpec	A font specification. This can be an integer, frame, or binary object specification of a font.
newFace	An integer, which specifies a font face.

Returns the altered *fontSpec*. If the font is a ROM font, a packed integer is returned.

If you specify the *fontSpec* as a frame, the returned frame is cloned from the input parameter *fontSpec*. If you specify the *fontSpec* as a binary object, the binary object itself is modified.

Note
You can replace the current values in a *fontSpec* only with your input specification. You cannot supplement the current values. For example, you cannot add the bold attribute to a font that already uses the underline attribute; instead, you must specify both attributes in your input specification. To combine existing values with new values, call the appropriate font attribute retrieval function (e.g., `GetFontFace`) and add in your new value(s). ◆

SetFontFamily

`SetFontFamily`*(fontSpec, newFamily)*

Sets the family of the font specified by *fontSpec* to the family specified by *newFamily* and returns the altered *fontSpec*.

fontSpec	A font specification. This can be an integer, frame, or binary object specification of a font.
newFamily	Can be either a symbol or integer that specifies a font family.

Returns the altered *fontSpec*. If the font is a ROM font, a packed integer is returned.

If you specify the *fontSpec* as a frame, the returned frame is cloned from the input parameter *fontSpec*. If you specify the *fontSpec* as a binary object, the binary object itself is modified.

SetFontParms

`SetFontParms` *(fontSpec, whichParms)*

Alters one or more components of a font specification. The *whichParms* parameter specifies which components of the *fontSpec* to alter.

`SetFontParms` returns a modified version of the font specification. If the specification can be packed into an integer (if the font is a ROM font), it returns a packed integer.

The returned value may be a modified version of the font passed in, or may be a modified clone of the original *fontSpec*. If possible, a packed integer is returned.

fontSpec — A font specification. This can be an integer, frame, or binary object specification of a font.

whichParms — A frame that specifies which components of the font spec to alter. The slots that can be used individually or in combination in this frame include:

- `size` — An integer representing the point size of the type. Usual values include: 9,10,12,14, and 18.
- `face` — An integer representing the font style attribute. The constants that you can use for font face values are shown in "Font Face Constants for Packed Integer Font Specifications" (page 7-7).
- `family` — A symbol or integer representing the typeface. Note that you cannot change the family of an ink font. The constants that you can use for font family numbers are

	shown in “Font Family Constants” (page 7-7)
`scale`	Applies only to ink fonts. An integer percentage of the original written ink size. When this slot is present, the `size` slot is ignored.
`penSize`	Applies only to ink fonts. An integer between 1 and 4.

If you specify the *fontSpec* as a frame, the returned frame is cloned from the input parameter *fontSpec*. If you specify the *fontSpec* as a binary object, the binary object itself is modified.

SetFontSize

`SetFontSize`*(fontSpec, newSize)*

Sets the size of the font specified by *fontSpec* to the size specified by *newSize* and returns the altered *fontSpec*.

fontSpec	A font specification. This can be an integer, frame, or binary object specification of a font.
newSize	The new font size, specified as an integer value.

Returns the altered *fontSpec*. If the font is a ROM font, a packed integer is returned.

If you specify the *fontSpec* as a frame, the returned frame is cloned from the input parameter *fontSpec*. If you specify the *fontSpec* as a binary object, the binary object itself is modified.

Rich String Functions and Methods

This section describes the functions and methods you can use to operate with rich strings. For a description of rich strings and the rich string format, see “Rich Strings” (page 8-22) in *Newton Programmer’s Guide*.

DecodeRichString

`DecodeRichString`*(richString, defaultFontSpec)*

Returns a frame containing two slots: `text` and a `styles`. These slots can be placed in a paragraph view for editing or viewing.

richString	A rich string that can contain text and ink words.
defaultFontSpec	The font specification for the text in *richString*. This is usually the same as the `viewFont` slot of the view in which the text is displayed.

Note
The `SetValue` function, which also decodes a rich string, is more efficient than the `DecodeRichString` function. ◆

ExtractRangeAsRichString

view:`ExtractRangeAsRichString`*(offset, length)*

Returns a rich string for the range of text specified from a paragraph view. This method can be used only on paragraph views.

offset	The beginning offset of the text range, specified as an integer value.
length	The number of characters in the range of text, specified as an integer value. Each ink word in the rich string counts as a single character.

GetRichString

view:`GetRichString()`

Returns either a rich string or a plain string that represents the text in the paragraph view to which the `GetRichString` message is sent. If the paragraph contains ink, `GetRichString` returns a rich string; if not, `GetRichString` returns a plain string.

IsRichString

`IsRichString`*(testString)*

Returns non-`nil` if the *testString* parameter is a rich string containing ink.

testString A rich string that can contain text and ink words.

MakeRichString

`MakeRichString`*(text, styleArray)*

Encodes the data from the *text* and *styleArray* parameters into a rich string.

text The text from the `text` slot of a paragraph view.

styleArray The array found in the `styles` slot of a view. The format of this array is described in the section "Text and Styles" (page 8-25) in *Newton Programmer's Guide*.

Returns a rich string that has the encoded information for the text and style array parameters.

StripInk

`StripInk(`*richString*`,` *replaceChar*`)`

Modifies *richString*, replacing every ink word placeholder in the string with *replaceChar*. If *replaceChar* is `nil`, the ink words are deleted.

richString The rich string to strip of ink word placeholders.

replaceChar The character to insert into *richString* in place of the ink word placeholders. Use `nil` to delete all ink words from the rich string.

Returns the modified string.

▲ **WARNING**
The `StripInk` function destructively modifies *richString*. ▲

Functions and Methods for Accessing Ink in Views

This section describes the functions you can use to determine if a view has ink in it and to access the ink in a paragraph view.

GetInkAt

`GetInkAt`*(para, index)*

Returns the next ink in the paragraph view specified by *para*.

para A paragraph view.

index The starting position of the search. If this value is `nil`, `GetInkAt` starts searching at the beginning of the paragraph text. If this value is an integer, `GetInkAt` starts searching at the next position after *index*.

The `GetInkAt` function returns a polygon view that contains the ink.

NextInkIndex

`NextInkIndex`*(para, index)*

Finds the next piece of ink within the paragraph view specified by *para*.

para A paragraph view.

index The starting position of the search. If this value is `nil`, `NextInkIndex` starts searching at the beginning of the paragraph text. If this value is an integer, `NextInkIndex` starts searching at the next position after *index*.

Returns the offset of the next ink in the paragraph. If `NextInkIndex` does not find ink, it returns `nil`.

To start checking at the beginning of the text, use `nil` as the value of *index*. To start checking for ink at offset *i*, use *i-1* as the value of *index*. To start checking at the next location in the text, use the value returned by the previous call to `NextInkIndex`.

ParaContainsInk

`ParaContainsInk`*(para)*

Determines if the paragraph view specified by *para* contains ink.

para A paragraph view.

If the paragraph view contains ink, `ParaContainsInk` returns the offset within the paragraph of the first piece of ink. If the paragraph view does not contain ink, `ParaContainsInk` returns `nil`.

PolyContainsInk

`PolyContainsInk`*(poly)*

Determines if the polygon specified by *poly* contains ink.

poly A polygon view.

Returns `true` if the polygon contains ink and `nil` if not.

Keyboards

This section describes the views, protos, and functions you can use in your applications to work with on-screen keyboards.

Keyboard View (clKeyboardView)

The `clKeyboardView` class displays keyboard-like arrays of buttons that can be pressed (tapped with the pen) to perform an action. To read about how to use this class, see “Keyboard Views” (page 8-26) in *Newton Programmer’s Guide*.

Slot descriptions

`_noRepeat` If present, indicates that keys do not repeat while held down.

`viewBounds` Set to the size and location where you want the view to appear.

`keyDefinitions` An array that defines the layout of the keys, as described in "The Key Definitions Array" (page 8-31) in *Newton Programmer's Guide*.

`viewFlags` The default setting is `vVisible + vClickable`.

`viewFormat` Optional. The default setting is `nil`.

`keyArrayIndex` Optional. Determines the array element to use for a key legend or result, allowing dynamic indexing into an array for legends or results. See "The Key Definitions Array" (page 8-31) in *Newton Programmer's Guide*.

`keyHighlightKeys`
Optional. An array of keys that to highlight on the displayed keyboard. Specify an array of `keyResult` items, as described in "The Key Definitions Array" (page 8-31) in *Newton Programmer's Guide*

`keyResultsAreKeycodes`
Optional. If true, indicates that integers specified as results are to be interpreted as key codes, and the corresponding character is returned. If `nil` (the default), integers are not converted to characters.

`keyReceiverView`
Optional. The view to which key commands (as a result of key presses) should be posted if no `keyPressScript` method exists. If the `keyReceiverView` slot is not found, the view identified by the symbol `'viewFrontKey` is used. This symbol evaluates at run time to the current key receiver view.

`keySound` Optional. A reference to a sound frame. The sound is played whenever a key is pressed. The default is no sound.

`keyPressScript` Optional. This method is called whenever a key is pressed. The key result of the key pressed is passed as a parameter to this method. If this method is not supplied, the key result is converted (if possible) into a sequence of characters, which are posted as key events to the key receiver view.

An example of a view definition of the `clKeyboardView` class, including the key definitions for the view, is shown in “Defining Keys in a Keyboard View” (page 8-30) in *Newton Programmer’s Guide*.

Keyboard Protos

This section provides reference information for the keyboard protos.

protoKeyboard

This proto creates a keyboard view that floats above all other views. It is centered within its parent view and appears in a location that won’t obscure the key-receiving view. For a description of how to use this proto, see “protoKeyboard” (page 8-28) in *Newton Programmer’s Guide*.

Slot descriptions

`saveBounds` — Set to the size and location where you want the keyboard view to appear. (This is used as the `viewBounds` value for the keyboard view.) Note that the keyboard view may be displayed above or below the location you specify, if it must be moved so as not to obscure the key-receiving view. (You can “freeze” it in place by using the `freeze` slot.)

`freeze` — Optional. If set to `true`, prevents automatic movement of the keyboard view. This slot is set to `nil` by default, allowing movement of the keyboard view so as not to obscure the key-receiving view, if it would be blocked by the bounds you specified for the keyboard.

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

This proto is used in conjunction with `protoKeypad` to implement a floating keyboard. It defines the parent view, and `protoKeypad` is a child view that defines the key characteristics.

The `protoKeyboard` itself uses the `protoFloater` proto, which is described in “Controls and Other Protos” (page 7-1) in *Newton Programmer’s Guide*.

protoKeypad

This proto defines key characteristics for a keyboard view (`clKeyboardView` class). For a description of how to use this proto, see “protoKeypad” (page 8-29) in *Newton Programmer’s Guide*.

Slot descriptions

`keyDefinitions` An array that defines the layout of the keys. Refer to the `clKeyboardView` description in “The Key Definitions Array” (page 8-31) in *Newton Programmer’s Guide*.

`viewFont` Optional. The default font is `ROM_fontSystem9Bold`.

`viewFormat` Optional. The default setting is `vfFillWhite`.

`keyArrayIndex` Optional. Set by this proto to zero.

`keyHighlightKeys`
Optional. Set by this proto to `nil`.

`keyResultsAreKeycodes`
Optional. Set by this proto to `true`.

`keyReceiverView`
Optional. Set by this proto to `'viewFrontKey`.

`keySound` Optional. Set by this proto to `typewriter`.

`keyPressScript` Optional. This method is called whenever a key is pressed. The result of the key press is passed as a parameter to this method. If this method is not supplied, the key result is converted (if possible) into a sequence of characters that are posted as key events to the key receiver view.

The `protoKeypad` is based on a view of the class `clKeyboardView`. For more information about the key slots listed above, refer to "Keyboard View (clKeyboardView)" (page 7-35).

Use this proto along with `protoKeyboard` to implement a floating keyboard. The view using the `protoKeypad` proto should be a child of the view using the `protoKeyboard` proto.

protoKeyboardButton

This proto is used to include the keyboard button in a view. For a description of how to use this proto, see "protoKeyboardButton" (page 8-29) in *Newton Programmer's Guide*.

Slot descriptions

`viewFlags` — The default is `vVisible + vReadOnly + vClickable`.

`viewBounds` — Set to the size and location where you want the keyboard to appear.

`viewJustify` — Optional. The default setting is `vjCenterH + vjCenterV`.

`defaultKeyboard` — Required. The symbol of the default keyboard to open. This value is not actually in the button view frame, but is found by inheritance.

Note that the `ViewClickScript`, `ButtonClickScript`, and `PickActionScript` methods are used internally in the `protoPictureButton` and should not be overridden.

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

protoSmallKeyboardButton

This proto is used to include the small keyboard button in a view. For a description of how to use this proto, see “protoSmallKeyboardButton” (page 8-30) in *Newton Programmer’s Guide*.

Slot descriptions

`viewFlags`	The default is `vVisible + vReadOnly + vClickable`.
`viewBounds`	Set to the size and location where you want the keyboard to appear.
`viewJustify`	Optional. The default setting is `vjCenterH + vjCenterV`.
`current`	Required. The symbol of the default keyboard to open. This value is not actually in the button view frame, but is found by inheritance.

The `protoSmallKeyboardButton` uses the `protoKeyboardButton` as its proto, and `protoKeyboardButton` uses the `protoPictureButton` as its proto.

Note that the `ViewClickScript`, `ButtonClickScript`, and `PickActionScript` methods are used internally in the `protoPictureButton` and should not be overridden.

protoAlphaKeyboard

This proto is used to include an alphanumeric keyboard in a view. For a description of how to use this proto, see “protoAlphaKeyboard” (page 8-30) in *Newton Programmer’s Guide*.

Slot descriptions

`viewBounds`	Set to the size and location where you want the keyboard to appear.
`viewJustify`	Optional. The default setting is `vjCenterH + vjCenterV`.

protoNumericKeyboard

This proto is used to include a numeric keyboard in a view. For a description of how to use this proto, see "protoNumericKeyboard" (page 8-30) in *Newton Programmer's Guide*.

Slot descriptions

`viewBounds` Set to the size and location where you want the keyboard to appear.

`viewJustify` Optional. The default setting is `vjCenterH + vjCenterV`.

protoPhoneKeyboard

This proto is used to include a phone keyboard in a view. For a description of how to use this proto, see "protoPhoneKeyboard" (page 8-30) in *Newton Programmer's Guide*.

Slot descriptions

`viewBounds` Set to the size and location where you want the keyboard to appear.

`viewJustify` Optional. The default setting is `vjCenterH + vjCenterV`.

protoDateKeyboard

This proto is used to include a time and date keyboard in a view. For a description of how to use this proto, see "protoDateKeyboard" (page 8-30) in *Newton Programmer's Guide*.

Slot descriptions

`viewBounds` Set to the size and location where you want the keyboard to appear.

`viewJustify` Optional. The default setting is `vjCenterH + vjCenterV`.

Keyboard Functions and Methods

This section describes the functions and methods you can use with keyboards in your Newton applications.

GetCaretBox

`GetCaretBox()`

Returns a bounds frame containing the global coordinates of the text insertion caret, if it is displayed. If there is a text selection in a view, the caret is positioned before the first character of the selection, though it may not be visible. If there is no text selection and the caret is not displayed, this function may still return a bounds frame giving the virtual position of the text caret. This is the last position of the caret when it was displayed, or the position where handwritten text would be inserted (usually immediately following existing text).

If there is no key-receiving view, `nil` is returned.

KeyboardInput

view:`KeyboardInput()`

Returns `true` if the view is the current key view (the view receiving keystrokes) and the keyboard is enabled (visible). Otherwise, this function returns `nil`.

This method applies only to views of the class `clEditView` and `clParagraphView`.

KeyIn

`KeyIn(`*keyCode*`, `*down*`)`

Allows you to programmatically change the state of the modifier keys (Caps Lock, Shift, and Option) on the alpha keyboard.

keyCode The physical keycode of the key whose state you want to change. Caps Lock is 0x39, Shift is 0x38, and Option is 0x3A.

down Specify `true` to cause the equivalent of a key press. Specify `nil` to release the key.

The key is highlighted on the alpha keyboard when it is pressed (*down* = `true`), and unhighlighted when it is released (*down* = `nil`). Note that if the keyboard is open, you must send it the `Dirty` message after changing the key state in order for the visual change to occur. This is not necessary if you use the `KeyIn` function to change the key state before opening the keyboard.

PostKeyString

`PostKeyString(`*view*`, `*keyString*`)`

Sends keystrokes to a view, as if they had been entered on a keyboard.

view The view to which to send keystrokes.

keyString A string containing the keystrokes to send.

This function always returns `nil`.

SetKeyView

`SetKeyView(`*view*, *offset*`)`

Sets the view that is to receive keyboard input from an on-screen keyboard and positions the caret at the specified offset in that view. Note that this function is only guaranteed to work with a `clParagraphView`. To place the caret in an edit view, you should use `SetCaretInfo` or `PositionCaret`.

view	The view to receive keyboard input. This must be a `clParagraphView`. Using `nil` for this value makes the caret disappear.
offset	The text caret is displayed at this character location. An offset of zero indicates the beginning of the view, an offset of one is after the first character, and so on.

Note that you may also call this function with only `nil` as the argument, to make the caret disappear. This function always returns `nil`.

Keyboard Registry Functions and Methods

If your application includes its own keyboard, you may need to use these functions. The system needs to know when keyboards are open, both for the purposes of the insertion caret and for keyboard-related callbacks.

KeyboardConnected

`KeyboardConnected()`

Returns non-`nil` if a keyboard is connected to the Newton.

OpenKeyPadFor

`OpenKeyPadFor`*(view)*

Opens a context-sensitive keyboard for the specified view.

The `OpenKeyPadFor` function first searches the proto chain to see if the view defines a keyboard in a `_keyboard` slot. If so, it opens the keyboard specified by that slot.

If the view does not define a keyboard, `OpenKeyPadFor` checks to see if the view allows only a single type of input for which the Newton system has a corresponding keyboard: date, time, phone number, or number. If so, it opens the appropriate keyboard.

If none of these other constraints is met, `OpenKeyPadFor` opens the `alphaKeyboard`.

view A view for which a context-sensitive keypad exists. Generally this should be the view that is returned by `GetKeyView`.

Note
The Newton System Software uses the `OpenKeypadFor` function to open a context-sensitive keyboard when the user double-taps on a view in which a `_keyboard` slot is defined. ◆

RegGlobalKeyboard

`RegGlobalKeyboard(`*kbdSymbol*`,` *kbdTemplate*`) // platform file function`

Installs a keyboard as the only alphanumeric keyboard. This replaces the built-in alpha keyboard view.

IMPORTANT
This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

`call kRegGlobalKeyboardFunc with (`*kbdSymbol*, *kbdTemplate*`);`

▲

kbdSymbol A unique identifier symbol for the keyboard view.

kbdTemplate A view template used to create the new keyboard. This template must include the the following slot:

`preAllocatedContext`
This slot must have the value `'alphaKeyboard.`

RegisterOpenKeyboard

view:`RegisterOpenKeyboard(`*flags*`)`

Notifies the system that a keyboard view is open and displays the insertion caret if necessary. You should call this method in your `ViewSetupDoneScript`.

flags Specifies how the keyboard is used. You can use a combination of the constants shown in the section "Keyboard Registration Constants" (page 7-8).

Note
Each keyboard prototype automatically calls the `RegisterOpenKeyboard` method. If you are using a keyboard prototype, you need not call this method. ◆

UnRegGlobalKeyboard

`UnRegGlobalKeyboard(`*kbdSymbol*, *kbdTemplate*`) // platform file function`

De-installs a keyboard that was installed by the `RegGlobalKeyboard` functions. This restores the built-in alpha keyboard view.

IMPORTANT
This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

`call kUnRegGlobalKeyabordFunc with (`*kbdSymbol*, *kbdTemplate*`);`
▲

kbdSymbol A unique identifier symbol for the keyboard view.

UnregisterOpenKeyboard

view:`UnregisterOpenKeyboard()`

Notifies the system that a keyboard view is no longer visible, which causes the insertion caret to be hidden, if necessary.

Note
The system automatically unregisters a keyboard when it is hidden or closed. ◆

Caret Insertion Writing Mode Functions and Methods

Use these functions to determine the setting of caret insertion writing mode or to set it yourself.

GetRemoteWriting

`GetRemoteWriting()`

Returns non-`nil` if caret insertion writing mode is currently enabled.

SetRemoteWriting

`SetRemoteWriting`*(newSetting)*

Sets the caret insertion writing mode preference. If *newSetting* is `nil`, caret insertion writing mode is disabled; otherwise, caret insertion writing mode is enabled.

newSetting Indicates the new setting (enabled or disabled) for caret insertion writing mode. If *newSetting* is `nil`, caret insertion writing mode is disabled; otherwise, it is enabled.

IMPORTANT
The caret insertion writing mode is a user preference that you should rarely change. The `SetRemoteWriting` method is meant to be called only from preferences or applications that serve a similar purpose. ▲

Insertion Caret Functions and Methods

This section describes the functions and methods you can use to retrieve information about or manipulate the insertion caret.

GetCaretInfo

```
GetCaretInfo()
```

Returns `nil` if there is no insertion caret. If there is an insertion caret, returns a frame with the following two slots:

`view` The view that owns the insertion caret. This can be either a `clParagraphView` or a `clEditView`.

`info` A frame whose contents depend on the type of view in which the caret is positioned.

If the caret is in a paragraph view, the slots are

`class`	`'paraCaret`
`offset`	The offset in characters of the caret position or the start of the selection, if there is one.
`length`	The length of the selection. The value of this slot is `0` if there is no selection.

If the caret is in an edit view and not inside any existing text, the slots are

`class`	`'editCaret`
`x`	The *x*-coordinate of the caret, in local coordinates.
`y`	The *y*-coordinate of the caret, in local coordinates.

If the caret is in a view that is more complex than a single paragraph, the slots are

`class`	`'hilite`

GetKeyView

`GetKeyView()`

Returns the view that owns the insertion caret.

Returns `nil` if there is no insertion caret.

Note

The insertion caret may have a defined view and offset even if it is not visible. The insertion caret is shown only when caret insertion writing mode is on, a keyboard is connected, or one or more keyboards are open on the screen. ◆

PositionCaret

view:`PositionCaret`*(x, y, playSound)*

Positions the caret at local coordinates within the view. You can use this method in an edit view.

x	The *x* position of the insertion caret in coordinates local to the view.
y	The *y* position of the insertion caret in coordinates local to the view.
playSound	If this value is non-`nil`, the system plays a sound when the caret is positioned.

▲ **WARNING**

You can use the `PositionCaret` method only with an edit view. ▲

SetCaretInfo

`SetCaretInfo(`*view, info*`)`

Restores the position of the insertion caret in a custom view that performs its own caret location management.

view — The view in which you want to modify the insertion caret information.

info — A frame in which you have specified the insertion caret information, using the same value types as are returned in the info parameter of the `GetCaretInfo` function, as described in "GetCaretInfo" (page 7-48).

▲ **WARNING**
You can use the `SetCaretInfo` function to restore the caret information for caret classes `'paraCaret` or `'editCaret`. You cannot use this function to restore caret information for caret class `'hilite`. The caret classes are described in "GetCaretInfo" (page 7-48). ▲

Application-Defined Methods for Keyboards

This section describes the keyboard-related methods you can define in order to perform keyboard-related actions at certain times.

ViewCaretChangedScript

view`:ViewCaretChangedScript`*(view, offset, length)*

Is sent to a registered keyboard view whenever the caret position or text selection has changed. Implement this method for a registered keyboard if you need to respond in some way to a change in the caret position or text selection.

view — The view in which the caret appears.

offset — Character offset of the insertion caret within the view, beginning with zero.

length The length of the text selection. If this value is `0`, there is no selection.

Input Event Functions and Methods

This section describes the methods that can use to handle and respond to input events in your applications.

Functions and Methods for Hit-Testing

This section describes the methods you can use to gather information about the location of user input in a paragraph view.

PointToCharOffset

view:`PointToCharOffset`*(x,y)*

Performs hit-testing for the character closest to the point specified by *x* and *y* in a paragraph view. The *x* and *y* values are specified as global point coordinates.

x,y Global point coordinates.

If the point (*x,y*) is within the paragraph margins, `PointToCharOffset` finds the character nearest to the point and returns its offset, measured from the beginning of the paragraph. If `PointToCharOffset` cannot find a character, it returns `-1`.

Note
This method works only for visible points in a paragraph view. You cannot hit-test an off-screen or clipped point. ◆

PointToWord

view:`PointToWord`*(x,y)*

Performs hit-testing for the word closest to the point specified by *x* and *y*. The *x* and *y* values are specified as global point coordinates.

x,y Global point coordinates.

If the point (*x,y*) is within the paragraph margins, `PointToWord` finds the word nearest to the point and returns a frame with two slots: `start` and `end`. The `start` slot specifies the offset from the beginning of the paragraph to the start of the word. The `end` slot specifies the offset from the beginning of the paragraph to the end of the word.

If `PointToWord` cannot find a word, it returns `nil`.

Note
This method works only for visible points in a paragraph view. You cannot hit-test an off-screen or clipped point. ◆

Functions and Methods for Handling Insertions

This section describes the methods and functions you can use to handle insertion events.

The Insert Specification Frame

Several methods in this section receive an input parameter that is an insert specification frame. This frame contains the following six slots:

`insertItems` The items to be inserted. This can be a single item or an array of items. Each item must be one of the valid item forms shown in Table 7-2 (page 7-53).

`addSpace` Optional. The value `true` adds a space between items unless `vNoSpaces` is set.

`undoable` Optional. If `true`, indicates that the insertion can be undone; otherwise, the insertion cannot be undone. The default value is `true`.

`insertOffset` Optional. The number of characters to offset the insertion from the beginning of the paragraph.

`replaceChars` Optional. Replaces this number of characters, starting at the insert offset. If no insert offset is specified, replaces the selection (if there is one).

`moveCaret` Optional. If `true`, indicates that the insertion caret should be moved to the position following the insertion; otherwise, the insertion caret is not moved. The default value is `true`.

Table 7-2 Valid items in an insert specification

Item type	Description	Example
string	Used for keyboard and plain text insertions.	`"hello"`

text and styles frame	Used for styled text.
	Note that if `styles` is not an array, it is assumed to be a single `fontSpec` that applies to all text.
		`{`
		`text: "hi there"`
		`styles: [len, fontSpec,`
		`         len, fontSpec, ...]`
		`}`

rich string	Used for rich string insertions.	
ink binary object	Used for ink words (class `'inkWord`).	
`correctInfo` frames	Used for handwritten words.	

For more information about `correctInfo` frames, see “Recognition” (page 9-1) in *Newton Programmer’s Guide*.

HandleInsertItems

view:`HandleInsertItems`*(insertSpec)*

Inserts one or more items into a paragraph.

You usually implement this method for paragraph views; however, you can implement it for a `clView` that has scripts set up to handle the `InsertItems` event.

insertSpec An insert specification frame, as described in the section "The Insert Specification Frame" (page 7-52).

Returns `nil`.

InsertItemsAtCaret

`InsertItemsAtCaret`*(insertSpec)*

Inserts one or more items into a paragraph at the caret position. The inserted items replace the selection, if there is one.

You usually implement this method for paragraph views; however, you can implement it for a `clView` that has scripts set up to handle the `InsertItems` event.

insertSpec An insert specification frame, as described in the section "The Insert Specification Frame" (page 7-52).

Note
You should not use the following insert specification frame slots for this method: `replaceChars`, `insertOffset`, and `moveCaret`. ◆

Functions and Methods for Handling Ink Words

This section describes the functions and methods you can use to work with ink words in your applications.

GetInkWordInfo

`GetInkWordInfo`*(inkWord)*

Returns information about an ink word.

inkWord	An ink word.

Returns a frame with the following slots:

`origWidth`	The width of the originally written ink word.
`origAscent`	The ascent of the originally written ink word.
`origDescent`	The descent of the originally written ink word.
`origXHeight`	The x-height of the originally written ink word.
`fontFace`	The font style of the ink word.
`scale`	The scaling percentage for the ink word.
`origPenSize`	The pen width used to display the word. This is the value defined in the Styles menu.
`origFontSize`	The font size of the originally written ink word.
`curFontSize`	The current font size of the ink word.
`curPenSize`	Unused. Do not rely on this value.
`curWidth`	The current (scaled) width of the ink word.
`curHeight`	The current (scaled) height of the ink word.
`curAscent`	The current (scaled) ascent of the ink word.
`curXHeight`	The current (scaled) x-height of the ink word.
`curDescent`	The current (scaled) descent of the ink word.

HandleInkWord

view`:HandleInkWord`*(strokeBundle)*

Hands a stroke bundle off to a view for processing.

strokeBundle Raw stroke data for the ink word. You need to convert this data to an ink word by calling the `StrokeBundleToInkWord` method, which is described in “StrokeBundleToInkWord” (page 8-89).

The view’s `ViewInkWordScript`, if any, is called as if the ink had been written by the user.

HandleRawInk

view:`HandleRawInk`*(strokeBundle)*

Sends a stroke bundle to a view for handling.

strokeBundle Raw data for the sketch ink, as described in “Recognition: Advanced Topics” (page 10-1).

The view’s `ViewRawInkScript`, if any, is called.

Application-Defined Methods for Handling Ink in a View

This section describes the messages that are sent for handling ink in a view.

ViewInkWordScript

view:`ViewInkWordScript`*(strokeBundle)*

Is sent when an ink word is recognized and sent to a view. The system searches for this method in the current view and its protos.

strokeBundle Stroke data for the ink word.

Returns `true` if your method handles the incoming ink word and `nil` if not.

If you do not handle the ink word, the edit and paragraph view default handlers are used. Note that views other than edit and paragraph views do not have default handlers.

ViewRawInkScript

view:`ViewRawInkScript`*(strokeBundle)*

Is sent when sketch ink is passed to a view. The system searches for this method in the current view and its protos.

strokeBundle Stroke data for the sketch ink.

Returns `true` if your method handles the incoming sketch ink and `nil` if not.

If you do not handle the sketch ink, the edit and paragraph view default handlers are used. Note that views other than edit and paragraph views do not have default handlers.

Recognition System Reference

This chapter describes in detail the constants, data structures, objects, methods, and global functions you can use to work with the recognition system.

Recognition System Data Structures

This section describes constants and data structures that you can use when working with the recognition system, including system-wide settings, view flags that control recognition behavior, system-supplied dictionaries, stroke bundles, word units, gesture units, shape units, point arrays, `recConfig` frames, `rcBaseInfo` frames, and `rcGridInfo` frames.

System-Wide Settings

You can use the following slots in the system's user configuration data to

- specify the use of a particular recognizer.
- enable and disable the system's ability to modify its handwriting model.
- enable and disable the automatic addition of words to the user dictionary and the auto-add dictionary.

Note that the values of most of these slots are set by the user in various preferences slips. Others are set by a `recToggle` view associated with the view performing recognition. Generally, you should not change the values of these slots.

To access slots in the system's user configuration data, use the `GetUserConfig` and `SetUserConfig` functions, as described in *Newton Programmer's Guide* Chapter 19, "Built-in Applications and System Data." After setting the values of recognition-related slots in the system's user configuration data, you must call the `ReadCursiveOptions` function to cause the recognition system to use the new settings.

Slot descriptions

`letterSetSelection`

Sets the text recognizer currently in use. This value may be either of the constants `kStandardCharSetInfo` (cursive recognizer) or `kUCBlockCharSetInfo` (printed recognizer). Although the recognizers built into Newton platforms through version 2.0 of system software support these values, other recognizers are not guaranteed to support them. You cannot set this value from your view's `recConfig` frame. This value is set by the user; for more information, see "User Preferences for Recognition" (page 9-14) in *Newton Programmer's Guide*.

`learningEnabledOption`

The default value `true` specifies that the system records learning data as the user corrects misrecognized words. Conversely, the value `nil` specifies that correcting misrecognized words does not modify the system-defined handwriting model. Because the printed recognizer does not record learning data, it ignores this value. For more information, see the description of this slot in "protoRecConfig" beginning on page 8-36; see also "User Preferences for Recognition" (page 9-14) in *Newton Programmer's Guide*.

`letterSpaceCursiveOption`

The value of this slot affects the amount of space required to consider sets of strokes as belonging to separate letters or words. This value may be set by the user from the Handwriting Recognition preferences slip, or it may be set programmatically in a `recConfig` frame. For more information, see the description of this slot in "protoRecConfig" beginning on page 8-36; see also "User Preferences for Recognition" (page 9-14) in *Newton Programmer's Guide*.

`timeoutCursiveOption`

This value affects the amount of time a recognizer waits from the completion of a stroke for subsequent strokes that might belong to the same word, shape, or graphic. For more information, see the description of this slot in "protoRecConfig" beginning on page 8-36; see also "User Preferences for Recognition" (page 9-14) in *Newton Programmer's Guide*.

`speedCursiveOption`

The amount of time the cursive recognizer spends recognizing input. Not all recognizers use this value; for more information, see the description of this slot beginning on page 8-39. See also "User Preferences for Recognition" (page 9-14) in *Newton Programmer's Guide*.

`letterInFieldsOption`

The value `true` specifies that, in addition to providing recognition behaviors specified by other settings, recognizers able to do so provide letter-by-letter recognition in `protoLabelInputLine` views. The value `nil` causes some recognizers to return only words appearing in the set of dictionaries available to the recognizer. On 2.0-based Newton systems, the cursive recognizer respects the `letterInFieldsOption` value; on the other hand, the printed recognizer always provides letter-by-letter recognition regardless of the value of this slot. The user can set this slot to `true` by selecting the "Letter-by-letter in fields" checkbox in the Handwriting Settings preferences slip. For more information, see "User Preferences for Recognition" (page 9-14) in *Newton Programmer's Guide*.

`lettersCursiveOption`

The default value `true` specifies that, in addition to providing recognition behaviors specified by other settings, recognizers able to do so provide letter-by-letter recognition in `clEditView` views. The value `nil` causes some recognizers to return only words appearing in the set of dictionaries available to the recognizer. On 2.0-based systems, the cursive recognizer respects the value of the `lettersCursiveOption` slot; on the other hand, the printed recognizer always provides letter-by-letter recognition regardless of the value of this slot. The user can set this slot to `true` by selecting the "Letter-by-letter in notes" checkbox in the Handwriting Settings preferences slip. For more information, see "User Preferences for Recognition" (page 9-14) in *Newton Programmer's Guide*.

`doAutoAdd` The default value `true` specifies that new words are added to the user dictionary and the auto-add dictionary automatically. The value `nil` specifies that words are not added to these dictionaries automatically. For more information, see "Disabling the Auto-Add Mechanism" (page 10-35) in *Newton Programmer's Guide*.

`doTextRecognition`
The value `true` enables word recognition. The system sets the value of this slot to `true` when the user selects the Text item from the `protoRecToggle` view. For more information, see "User Preferences for Recognition" (page 9-14) in *Newton Programmer's Guide* and "Using RecConfig Frames to Enable Recognizers" (page 10-10) in *Newton Programmer's Guide*.

`doShapeRecognition`
The value `true` enables shape recognition. The system sets the value of this slot to `true` when the user selects the Shapes item from the `protoRecToggle` view. For more information, see "User Preferences for Recognition" (page 9-14) in *Newton Programmer's Guide* and "Using RecConfig Frames to Enable Recognizers" (page 10-10) in *Newton Programmer's Guide*.

`doInkWordRecognition`
The value `true` causes the recognizer to convert strokes to ink text rather than sketch ink. Ink text may also be returned to a view when the text recognizer is enabled but cannot recognize the input successfully or when text and shape recognition is disabled. The system sets the value of this slot to `true` when the user selects the Ink Text item from the `protoRecToggle` view. For more information, see "User Preferences for Recognition" (page 9-14) in *Newton Programmer's Guide* and "Using RecConfig Frames to Enable Recognizers" (page 10-10) in *Newton Programmer's Guide*.

View Flags for Recognition

This section describes flags that enable the recognizers and dictionaries used by views for recognition. The system also provides flags that specify aspects of the view's appearance and drawing behavior; for information about these additional view flags, see Chapter 2, "Views Reference."

Note that the specific set of dictionaries enabled by a particular flag can vary according to the user's current location as specified in the built in Time Zones application. The set of dictionaries used by a particular view is specified by a combination of the default settings, the locale specified in user preferences, and the set of view flags specified for the view. For more information about locales, see "How Locale Affects Recognition" (page 20-2) in *Newton Programmer's Guide*.

Multiple view flags may be combined to provide a view with a particular set of attributes; however, every option may not be available in every kind of view. For example, a view of the `clView` class can accept clicks (taps) but can't recognize words unless you supply code that provides this behavior.

The flags described here may be specified in the `viewFlags` slot of the view performing recognition or in the `inputMask` slot of the view's `recConfig` frame. For information on using the `viewFlags` slot, see "Enabling Recognizers" (page 9-8) in *Newton Programmer's Guide*. For information on using the `inputMask` slot, see the following sections in *Newton Programmer's Guide*: "Creating a recConfig Frame"(page 10-9) and "Creating Single-Letter Input Views"(page 10-15).

Note

Although the Newton Toolkit user interface distinguishes between entry flags and view flags, this chapter refers to all such flags as view flags. For more information, see "Flag-Naming Conventions" (page 9-19) in *Newton Programmer's Guide*. ◆

Table 8-1 summarizes the view flags that enable text recognition using enumerated dictionaries (including custom dictionaries).

Table 8-1 View flags for text recognition using enumerated dictionaries

Constant	Value	Description
`vCharsAllowed`	`1 << 12` or `0x01000`	Enables the default text recognizer and the default dictionary set. The default text recognizer is specified as a system-wide setting by the user from within the Handwriting Recognition preferences slip. The default dictionary set for a particular view is defined according to the view class or system prototype from which it is derived, its set of view flags, and the current locale. Setting this flag enables dictionaries for common words, proper names, and the review dictionary (which contains the user, auto-add, and expand dictionaries). For more information, see "How Locale Affects Recognition" in Chapter 15, "Localizing Newton Applications."
`vLettersAllowed`	`1 << 14` or `0x04000`	Enables letter-by-letter text recognition. Set this flag for views that may need to recognize words not present in the currently available set of dictionaries. Setting this flag enables the default text recognizer; if that recognizer is the cursive recognizer, it is enabled in letter-by-letter mode, which allows it to recognize combinations of letters that are not dictionary items. (Note that the printed recognizer can always recognize words that are not present in dictionaries.) For example, the cursive recognizer can return nonword combinations of characters such as "xyz" when the `vLettersAllowed` flag is set. Take care to use this flag only when necessary, as it can slow the performance of the cursive recognizer and make it less reliable.

Table 8-1 View flags for text recognition using enumerated dictionaries (continued)

Constant	Value	Description
`vAddressField`	`1 << 21` or `0x0200000`	Enables recognizers and dictionaries suitable for the input of address data in the current locale. It is not necessary to set the `vPunctuationAllowed` or `vNumbersAllowed` flags in conjunction with this flag. The set of dictionaries this flag enables is suitable for recognizing numbers, punctuation, abbreviations, common words, and proper nouns. Words found in proper noun dictionaries are in most cases capitalized before they are returned to the view for display; thus, you need not set the `vCapsRequired` flag in conjunction with the `vAddressField` flag. At your discretion, you can set the `vCapsRequired` flag to force the capitalization of recognized words before they are returned to the view.
`vNumbersAllowed`	`1 << 13` or `0x02000`	Enables the recognition of numeric characters, monetary values (for example, $12.25), decimal points, and signs (+ or –). To recognize integer values only, set the `vCustomDictionaries` flag instead of setting the `vNumbersAllowed` flag and place only the `kNumbersDict` constant in the `dictionaries` slot of the view or its `recConfig` frame.
`vNameField`	`1 << 22` or `0x0400000`	Enables text recognition optimized for name data. This flag is usually combined with the `vCapsRequired` flag. This flag does not provide access to or control of the Names application or the Names soup.

Table 8-1 View flags for text recognition using enumerated dictionaries (continued)

Constant	Value	Description
`vCustomDictionaries`	`1 << 24` or `0x01000000`	Enables text recognition using dictionaries specified by values in the view's `dictionaries` slot. This flag is used for views that accept custom data such as company names, plant species, and so on. When this flag is set, the view's template or `recConfig` frame must provide a `dictionaries` slot that contains a single dictionary identifier or an array of dictionary identifiers. These identifiers may refer to custom dictionaries you provide or to built-in dictionaries that the system provides. You need not set the `vCharsAllowed` flag with the `vCustomDictionaries` flag unless the view needs to use the system-supplied dictionaries that the `vCharsAllowed` flag enables.
`vPunctuationAllowed`	`1 << 15` or `0x08000`	Enables recognition of punctuation marks by the cursive recognizer. (The printed recognizer always recognizes punctuation marks in any position in input strings, regardless of the setting of the `vPunctuationAllowed` flag.) This flag enables recognition of the following marks preceding a word: single quotation mark, double quotation mark, left parenthesis, and hyphen. This flag also enables the recognition of the following marks at the end of a word: single quotation mark, double quotation mark, right parenthesis, hyphen, period, comma, exclamation point, question mark, colon, and semicolon.

Table 8-1 View flags for text recognition using enumerated dictionaries (continued)

Constant	Value	Description
vCapsRequired	1 << 23 or 0x0800000	Forces capitalization of the first character of each recognized word before it is returned to the view. Setting this flag reduces the view's ability to accept uncapitalized input. Views that do not set this flag capitalize words according to the size of the first letter in the word and the capitalization requirement specified by the dictionary used to recognize the word, if any.

Table 8-2 describes view flags that enable text recognition using system-supplied lexical dictionaries.

Table 8-2 View flags for text recognition using lexical dictionaries

Constant	Value	Description
vNumbersAllowed	1 << 13 or 0x02000	Enables recognition of numbers, monetary values (for example, $12.25), decimal points, and mathematical signs (+ and –).
vPhoneField	1 << 18 or 0x040000	Enables recognition of phone numbers. Note that the set of lexical dictionaries enabled by this flag varies with the text recognizer currently in use. Most notably, views for which this flag is set can recognize phone numbers with intermixed alphabetic characters (for example, "1-800-NOOTOON") when the printed recognizer is enabled, but not when the cursive recognizer is enabled.
vDateField	1 << 19 or 0x080000	Enables recognition of date formats (such as "March 3-95"), names of months, and names of days.
vTimeField	1 << 20 or 0x0100000	Enables recognition of times.

Note

The lexical dictionaries enabled by a particular flag can vary according to the user's current location as specified in the built in Time Zones application. For more information, see "How Locale Affects Recognition" (page 20-2) in *Newton Programmer's Guide*. ◆

Table 8-3 describes view flags that control the nontextual aspects of recognition system behavior.

Table 8-3 Nontext view flags

Constant	Value	Description
`vNothingAllowed`	`0x00000000` or `0x0000`	The view accepts no handwritten or keyboard input. The NTK view editor does not provide a checkbox to set this flag, as it is equivalent to turning off all of the other flags.
`vAnythingAllowed`	`65535 << 9` or `0x01FFFE00`	Set this flag only for views derived from the `clEditView` class. This flag is actually a mask that turns on all recognizers, theoretically allowing the view to accept any kind of input; however, the recognition that the view actually performs at run time is controlled by a combination of user preferences settings, `recToggle` settings, and `recConfig` settings. You must be certain that the `recToggle` view is visible when you use this flag, because it allows the creation of a state in which nothing is recognized. That is, if recognition is turned off and the `recToggle` view is not displayed, the user cannot enable recognition in the view. Note that you'll obtain faster and more accurate recognition using the correct set of individual flags for the types of data that your view accepts. To control specific recognizers, you must use a combination of the other view flags that the system provides.

Table 8-3 Nontext view flags (continued)

Constant	Value	Description
`vClickable`	`1 << 9` or `0x0200`	The system sends the `ViewClickScript` message to the view once for each pen tap that occurs within the view. The unit passed as the argument to the `ViewClickScript` method is valid only during the recognition process—that is, while the various recognition-related scripts are being called. Do not attempt to save units for later use. You must set the `vClickable` flag for any view that is to accept pen input; no taps or strokes are passed to the view when this flag is not set. Views that handle taps explicitly (such as buttons) or that track the pen themselves can set this flag and use the `ViewClickScript` method to implement their handling of pen input. This method can track the position of the pen by calling the `GetPoint` function. For more information, see "GetPoint" on page 8-79. Electronic ink is turned on or off depending on the `vClickable` flag's interaction with the `ViewClickScript` method and the settings of view flags for views in the `_parent` chain. If `vClickable` is the only view flag set for the view, inking is turned off automatically. However, if `vClickable` is not set for the view, any of its parent views may handle clicks or draw ink. For more information, see "Taps and Overlapping Views" (page 9-24) in *Newton Programmer's Guide.* Ink is turned on in views having a `ViewClickScript` method. To turn off inking, the `ViewClickScript` method can call either of the global functions `InkOff` or `InkOffUnHobbled`. Note that the `TrackHilite` and `TrackButton` methods also disable inking.

Table 8-3 Nontext view flags (continued)

Constant	Value	Description
`vStrokesAllowed`	`1 << 10` or `0x0400`	The view accepts strokes and is sent the `ViewStrokeScript` message at the end of each stroke. Note that when several strokes occur within the amount of time specified by the `timeoutCursiveOptions` value, only the first stroke causes the `ViewStrokeScript` message to be sent. The only time you need to set this flag is when the view has a `ViewStrokeScript` method. You might use this method to do something application-specific with strokes, such as recognizing your own gestures. Don't set this flag if your view does not have a `ViewStrokeScript` method—you'll only waste battery power! You must also set the `vClickable` flag when using this flag; otherwise, the view accepts no input.

Table 8-3 Nontext view flags (continued)

Constant	Value	Description
`vGesturesAllowed`	`1 << 11` or `0x0800`	The view accepts gesture strokes such as scrub, highlight, tap, double tap, caret, and line. Most views that accept input also set this flag so that gestures such as scrub can be used. You must also set the `vClickable` flag when using the `vGesturesAllowed` flag; otherwise, the view accepts no pen input. Setting this flag causes the view to send the `ViewGestureScript` message when it recognizes a gesture that it does not handle automatically. Views based on the `clEditView` and `clParagraphView` classes handle standard gestures automatically. To interpret gestures yourself in a `clView` view, you must set the `vGesturesAllowed` flag and provide a `ViewGestureScript` method. See "ViewGestureScript" (page 8-71) for more information.
`vShapesAllowed`	`1 << 16` or `0x010000`	Enables shape recognition within a view based on the `clEditView` class. You must also set the `vClickable` flag when using this flag; otherwise, the view accepts no input.

Table 8-3 Nontext view flags (continued)

Constant	Value	Description
`vSingleUnit`	`1 << 8` or `0x0100`	Disables the recognition system's use of spatial cues (distance between strokes), forcing it to rely on temporal cues (time between the completion of one stroke and the beginning of another) to determine when the user has completed a group of strokes. Using this flag may result in better recognition of complex stroke groups in which users tend to put large spaces, such as phone numbers. This flag has meaning for text recognizers only. Once input has been recognized and added to the view, subsequent input is recognized as separate words. In effect, setting this flag causes the recognizer to ignore short delays, such as those that occur between writing the individual characters in a word. Longer delays cue the recognizer to group the most recently completed set of strokes as a word. The amount of time considered to be a longer delay is a function of the speed of the processor and the recognition system, as well as the value of the `timeoutCursiveOption` user preference. For additional information on suppressing spaces, see the description of the `vNoSpaces` flag.
`vNoSpaces`	`1 << 1` or `0x0002`	Directs a view based on the `clParagraphView` class to not insert spaces between existing text and new text. This post-processing flag neither restricts the interpretation of the input strokes nor assists the recognition system in choosing between alternative interpretations of the input, as the `vSingleUnit` flag does.
`vWidthIsParentWidth`	`1 << 0` or `0x0001`	The right boundary of the `clParagraphView` view is extended to match that of its parent.

System-Supplied Dictionaries

The system supplies various enumerated and lexical dictionaries for the recognition system's use. The set of dictionaries used by a particular view is specified by a combination of the default settings, the locale specified in user preferences, and the set of view flags specified for the view.

Table 8-4 describes the system-supplied enumerated dictionaries accessible from NewtonScript. Note that the content of the dictionary represented by the `kLocalPropersDictionary` constant may vary according to the user's locale. For information on locales, see Chapter 20, "Localizing Newton Applications," in *Newton Programmer's Guide.*

Table 8-4 System-supplied enumerated dictionaries

Dictionary ID (constant)	Value	Contents
`kUserDictionary`	31	Words added by the user
`kCommonDictionary`	0	Commonly used words
`kCountriesDictionary`	8	Names of countries
`kDaysMonthsDictionary`	34	Names of days and months
`kFirstNamesDictionary`	48	First names
`kLocalCitiesDictionary`	41	Names of cities
`kLocalPropersDictionary`[1]	2	Proper names
`kLocalStatesDictionary`	43	Names of states, provinces, etc.
`kSharedPropersDictionary`	1	Proper names, company names, state or province names, and abbreviations

[1] Locale-specific dictionary

Note

Although these constants currently evaluate to integers, do not rely on the integer values. Use only the appropriate constant names to reference these dictionaries. ◆

Table 8-5 lists constants representing system-supplied lexical dictionaries that define formats for dates, times, phone numbers, postal codes, currency values, and other numeric values. Note that most lexical dictionaries are no longer locale-specific—each dictionary specifies lexical formats for all locales. However, the dictionaries represented by the `kLocalNumberDictionary`, `kMoneyOnlyDictionary`, and `kNumbersOnlyDictionary` constants may vary according to the user's locale. For information on locales, see *Newton Programmer's Guide* Chapter 20, "Localizing Newton Applications."

Table 8-5 System-supplied lexical dictionaries

Dictionary ID (constant)	Value	Contents
`kLocalDateDictionary`	`110`	Date formats
`kLocalNumberDictionary`[1]	`113`	Currency and numeric formats
`kLocalPhoneDictionary`	`112`	Phone number formats
`kLocalTimeDictionary`	`111`	Time formats
`kMoneyOnlyDictionary`[1]	`118`	Currency values and formats
`kNumbersOnlyDictionary`[1]	`117`	Numeric values and formats
`kPostalCodeDictionary`	`116`	Postal code formats

[1] Locale-specific dictionary

Note
Although these constants currently evaluate to integers, do not rely on the integer values. Use only the appropriate constant names to reference these dictionaries. ◆

Recognition Configuration Frames

Recognition configuration frames (`recConfig` frames) provide an alternate interface to the recognition system. They can be used to provide any behavior that view flags provide, to supplement behavior provided by view flags, or to provide specialized recognition behaviors that view flags cannot. The use of a `recConfig` frame is required to support ink text, specify baseline information, perform deferred recognition, and define grids of single-letter input areas within a view.

For descriptions of the slots and methods in `recConfig` frames, see "protoRecConfig" on page 8-36.

System-Supplied recConfig Frames

You can base your view's `recConfig` frame on one of the system-supplied `recConfig` frames described in this section.

The `recConfig` frames supplied by the constants `ROM_rcInkOrText`, `ROM_rcPrefsConfig`, and `ROM_rcRerecognizeConfig` require no modification to produce useful behavior. You must provide appropriate initial values for some slots in the `recConfig` frames supplied by the `ROM_rcDefaultConfig`, `ROM_rcSingleCharacterConfig`, and `ROM_rcTryLettersConfig` constants.

For information regarding the use of the constants described in this section, see the following sections in *Newton Programmer's Guide*: "Creating a recConfig Frame" (page 10-9) and "Changing Recognition Behavior Dynamically"(page 10-17).

`ROM_rcInkOrText`

This general-purpose `recConfig` frame can be used as it is for views that accept text input. It allows the user to turn on text recognition only; when text recognition is disabled, the system returns ink text to the view. This `recConfig` frame is generally used with a `protoRecToggle` view to allow the user to specify whether the view displays ink text or normal text. The `ROM_rcInkOrText` frame provides the following slots.

`allowTextRecognition`

Default value of `true` allows user to enable the text recognizer from an associated `recToggle` view. See the description of the `allowTextRecognition` slot (page 8-37) for more information.

`doInkWordRecognition`

Default value of `true` enables recognition of input as ink text when text recognizer is off. See the description of the `doInkWordRecognition` slot on page 8-38 for more information.

`ROM_rcPrefsConfig`

This frame can be used as is to configure views for performing recognition according to user preference settings. Views that have recognition behavior based on this frame permit the user to enable or disable any recognizer for which the system provides a user interface. The default recognition behavior of views that set the `vAnythingAllowed` mask is based on this frame.

Note

The `ROM_rcPrefsConfig` frame does not specify an input mask, forcing the system to build one using settings specified in user preferences. ◆

The `ROM_rcPrefsConfig` frame provides the following slots:

`allowTextRecognition`

Default value of `true` allows the user to enable the text recognizer from an associated `protoRecToggle` view. See the description of the `allowTextRecognition` slot (page 8-37) for more information.

`allowShapeRecognition`

Default value of `true` allows the user to enable the shape recognizer from an associated `protoRecToggle` view. See the description of the `allowShapeRecognition` slot on page 8-37 for more information.

`ROM_rcDefaultConfig`

The `ROM_rcDefaultConfig` frame can be used as a prototype for a generic `recConfig` frame; it provides a useful set of slots for which you must supply values. This frame provides the following slots:

`punctuationCursiveOption`

A value of `true` specifies that the view recognizes punctuation marks. This frame supplies a default value of `nil`.

`dictionaries`

The list of dictionaries to use for recognition. This slot holds an array of dictionary identifiers, a single dictionary identifier, or the value `nil`. This frame supplies a default value of `nil`. For more information, see the description of the `dictionaries` slot in the section "Using Your RAM-Based Custom Dictionary" (page 10-28) in *Newton Programmer's Guide*.

`rcSingleLetters`

A value of `true` specifies that the view recognizes single letters only, rather than dictionary words. This frame supplies a default value of `nil`.

`rcBaseInfo`

Holds an `rcBaseInfo` frame, which describes the coordinates of an editable view having known baselines. This frame supplies a default value of `nil`. For more information, see "rcBaseInfo" beginning on page 8-25.

`inputMask` A bit field specifying the configuration of the recognition system for this view. This frame supplies a default value of zero

(0x0000). For more information, see "View Flags for Recognition" beginning on page 8-6.

ROM_rcSingleCharacterConfig

This frame can be used as it is to configure recognition in views accepting single-character input. For example, you can use this frame to configure the entry fields in a crossword puzzle or the entry fields in a single-character corrector view similar to the protoCharEdit system prototype. For an example of the use of this prototype, see "Creating Single-Letter Input Views" (page 10-15) in *Newton Programmer's Guide*.

The ROM_rcSingleCharacterConfig frame provides the following slots:

_proto
: The default value of this slot is ROM_rcDefaultConfig. Do not change the value of this slot. For more information regarding slots that this frame acquires through prototype inheritance, see the description of the ROM_rcDefaultConfig constant beginning on page 8-20.

letterSpaceCursiveOption
: Indicates whether the recognition system segments strokes into groups by interpreting spatial and temporal cues. The default value of nil specifies that the system performs no segmentation, which is appropriate for a field in which all strokes are to be interpreted as a single word.

rcSingleLetters
: The default value of true indicates that the text recognizer is to recognize single letters rather than dictionary words.

`inputMask` This view's input mask. The default value of `vCustomDictionaries` indicates that the view uses the dictionaries specified in the view's `dictionaries` slot. For more information, see the description of the `dictionaries` slot in the section "Using Your RAM-Based Custom Dictionary" (page 10-28) in *Newton Programmer's Guide*.

`dictionaries`
The default value of `kSymbolsDictionary` specifies that this view uses the system-supplied symbols dictionary for recognition. The symbols dictionary is used to recognize single alphanumeric characters, punctuation marks, mathematical symbols, diacritical marks, and so on.

`inhibitSymbolsDictionary`
The default value of `true` specifies that the system is not to use the symbols dictionary in addition to the specified dictionaries. (To do so would be redundant: the symbols dictionary is already enabled by this frame's `dictionaries` slot.)

`ROM_rcTryLettersConfig`
This frame can be used as it is to configure a view for recognizing alphanumeric character combinations that do not appear in available dictionaries; it is intended for use by views that implement their own form of deferred recognition. For example, the system uses this `recConfig` frame when the user chooses the Try Letters item from the picker displayed as the result of double-tapping a word previously recognized by the cursive recognizer.

`_proto` The default value of this slot is `ROM_rcDefaultConfig`. For more information regarding slots that this frame acquires through prototype inheritance, see the description of the `ROM_rcDefaultConfig` constant beginning on page 8-20.

`letterSpaceCursiveOption`
Indicates whether the recognition system segments strokes into groups by interpreting spatial and temporal cues. The default value of `nil` specifies that the system performs no segmentation, which is appropriate for a field in which all strokes are to be interpreted as a single word.

`inputMask` The default value of `vLettersAllowed+vNumbersAllowed` configures this view to recognize non-dictionary words and numbers. See the descriptions of these flags (page 8-6) for more information. For information regarding the use of the NewtonScript plus (+) operator to combine view flags, see "Combining View Flags" (page 9-26) in *Newton Programmer's Guide*.

`ROM_rcRerecognizeConfig`
This frame can be used as it is by views that implement their own form of deferred recognition. It builds an input mask from user preference settings and the settings of an associated `recToggle` view.

`allowTextRecognition`
Default value of `true` causes the value of the `doTextRecognition` slot to be used. See the description of the `allowTextRecognition` slot (page 8-37) for more information.

`doTextRecognition`
: The default value `true` enables word recognition in the view that this `recConfig` frame controls. For more information, see the `protoRecConfig` section's description of this slot on page 8-37.

`speedCursiveOption`
: The amount of time the cursive recognizer spends recognizing input. This frame provides a default value of 2. For more information see the `protoRecConfig` section's description of this slot on page 8-39.

`letterSpaceCursiveOption`
: Indicates whether the recognition system uses spatial and temporal cues to segment strokes into groups. The default value of `nil` specifies that the system performs no segmentation, which is appropriate for a field in which all strokes are to be interpreted as a single word.

`ROM_canonicalBaseInfo`
: System-supplied `rcBaseInfo` frame. Clone this frame into your `recConfig` frame's `rcBaseInfo` slot.

`ROM_canonicalCharGrid`
: System-supplied `rcGridInfo` frame. Clone this frame into your `recConfig` frame's `rcGridInfo` slot.

Data Structures Used in recConfig Frames

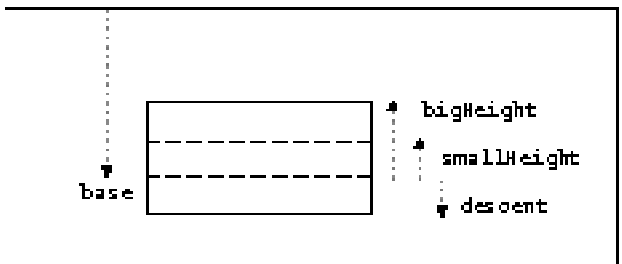
The system-supplied `rcBaseInfo` and `rcGridInfo` frames are used within `recConfig` frames to define baseline information and grids of single-letter input views, respectively.

rcBaseInfo

This frame specifies to the recognizer precisely where characters are written with respect to a well-defined baseline in a view. The `rcBaseInfo` frame is especially valuable in improving the recognition of single letters or letter-size values, for which it is sometimes difficult to derive baseline information from user input alone. For example, without adequate baseline information it is difficult to distinguish between an upper-case letter `P` and a lower-case letter `p`.

Figure 8-1 depicts the editing box that an `rcBaseInfo` frame defines.

Figure 8-1 Single-character editing box specified by `rcBaseInfo` frame



The `rcBaseInfo` frame has the following slots:

Slot descriptions

`base`	The Y coordinate of the view's baseline, expressed in screen coordinates (global coordinates).
`smallHeight`	Positive offset, expressed in pixels, from `base` to the top of a lowercase *x*. Set to `nil` if you aren't sure what value this slot should have.
`bigHeight`	Positive offset, expressed in pixels, from `base` to the top of an uppercase *X*. Set to `nil` if you aren't sure what value this slot should have.

`descent` Positive offset, expressed in pixels, from `base` to the bottom of a lowercase *g*. Set to `nil` if you aren't sure what value this slot should have.

If you aren't sure of appropriate values for the `smallHeight`, `bigHeight`, or `descent` slots, it's better to set them to `nil` than to provide inaccurate values. In general, you shouldn't specify these values unless there is a visible guideline on the screen with which the user can align handwritten input.

Note
If the user can drag the view around on the screen, you'll need to offset the value of the `base` slot when the view is moved. ◆

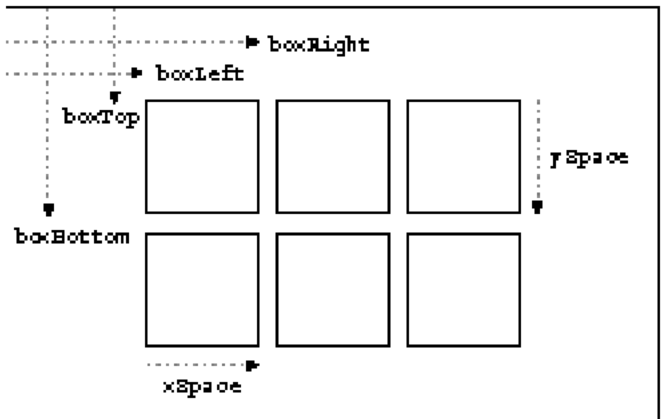
rcGridInfo

You can use the `rcGridInfo` frame in conjunction with an `rcBaseInfo` frame to define to the recognizer the position of a single letter input area within a specified view. The `rcGridInfo` frame can be used to define a single box, a horizontal array of boxes, a vertical array of boxes, or a two-dimensional array of boxes. For example, the system-supplied `protoCharEdit` prototype uses an `rcGridInfo` frame to define the cells of the comb view it provides.

If you provide a grid in which the user is to write characters or words, you need to use an `rcGridInfo` frame to define the grid to the text recognizer. The recognizer requires the information in an `rcGridInfo` frame in order to make character-segmentation decisions.

Figure 8-2 depicts the grid—the two-dimensional array of boxes—that an `rcGridInfo` frame can define.

Figure 8-2 Two-dimensional array of input boxes specified by `rcGridInfo` frame



The `rcGridInfo` frame has the following slots:

Slot descriptions

`boxLeft` The global (screen) coordinate of the left edge of the top-left box.

`boxRight` The global (screen) coordinate of the right edge of the top-left box.

`xSpace` The distance from one `boxLeft` coordinate to the next `boxLeft` coordinate.

`boxTop` The global (screen) coordinate of the topmost edge of the top-left box.

`boxBottom` The global (screen) coordinate of the bottom edge of the top-left box.

`ySpace` The distance from one `boxTop` coordinate to the next `boxTop` coordinate.

The definition of a horizontal array requires the presence of the `boxLeft`, `boxRight`, and `xSpace` slots. The definition of a vertical array requires the presence of the `boxTop`, `boxBottom`, and `ySpace` slots. The definition of a two-dimensional array requires that all six slots be defined.

Note
If the user can drag the view around on the screen, you'll need to offset the values of the `boxLeft`, `boxRight`, `boxTop`, and `boxBottom` slots when the view is moved. ◆

Stroke Bundle Data Structures

This section describes the data structures that you can use to work with stroke bundles.

The Stroke Bundle Frame

The stroke bundle frame describes the point data from an input stroke drawn on the Newton tablet. This frame contains the following slots:

Slot descriptions

`bounds`	The bounding rectangle for the ink strokes in the bundle.
`strokes`	An array with one element for each stroke in the bundle. Each element is a binary object containing tablet resolution data.

Format Specification Values for Stroke Bundle Functions

Several stroke bundle functions use a format specification to determine the resolution of point data. Some functions also use this format specification to determine whether or not to copy duplicate point values. The format specification values are shown in Table 8-6.

Table 8-6 Stroke bundle data format specifications

Value	Description
`0`	Data in screen resolution. Filter out duplicate points.
`1`	Data in screen resolution. Duplicate points are allowed.
`2`	Data in tablet resolution. Filter out duplicate points.
`3`	Data in tablet resolution. Duplicate points are allowed.

Note
Points are stored in a compressed format that is based on screen resolution. ◆

Filtering of duplicate points is irrelevant for several stroke bundle functions. These functions use screen resolution if you supply a filter value of `0` or `1`, and tablet resolution if you supply a filter value of `2` or `3`. For example, the `GetStrokePoint` function (page 8-85) retrieves a specific point from a stroke bundle, and needs to know only the resolution in which to return that point.

Stroke, Word, and Gesture Units

The Newton recognition system uses stroke units to describe information about pen input. You cannot examine a stroke unit directly, but some stroke bundle and recognition functions accept this object type as an argument. The system passes stroke units to the optional `ViewStrokeScript` method of a view that performs recognition.

The Newton recognition system also uses other units. These include word units, which are passed to a view's optional `ViewWordScript` method, and gesture units, which are passed to a view's optional `ViewGestureScript` method.

For more information about stroke, word, and gesture units, as well as the application-defined view methods that use them, see "Customized Processing of Input Strokes" (page 10-40) in *Newton Programmer's Guide*.

Point Arrays

Several of the stroke bundle functions use or return point arrays. This is a single array of coordinate values, with alternating *y* and *x* coordinates.

Note that the first value in each pair is the *y* coordinate value, followed by the *x* coordinate value.

The point array structure is the same structure type that is returned by the `GetPointsArray` function, described on page 8-81.

CorrectInfo Frame

This frame, which contains correction information for recently recognized words, is returned by the `GetCorrectInfo` global function (page 8-56). For descriptions of the slots and methods in this frame, see "protoCorrectInfo" on page 8-53.

WordInfo Frame

This frame contains stroke data, correction information, and learning data for a single written word interpreted by the text recognizer. An array of `wordInfo` frames representing recently recognized words is held by the `info` slot of the `correctInfo` frame. Individual `wordInfo` frames may also be extracted from word units passed to the optional `ViewWordScript` method of the view performing text recognition. For descriptions of the slots and methods in this frame, see "protoWordInfo" on page 8-60.

WordInterp Frame

This frame represents a single interpretation of input strokes returned by the text recognizer. An array of `wordInterp` frames resides in the `wordInfo` frame's `words` slot. For descriptions of the slots in this frame, see "protoWordInterp" on page 8-63.

Recognition System Prototypes

This section describes protos used to configure the recognition system or provide a user interface to it.

protoRecToggle

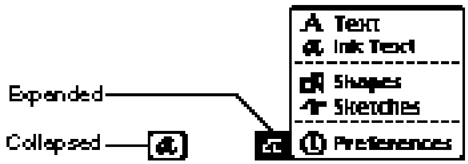
The `protoRecToggle` system prototype provides a picker that controls recognition in an associated view. This prototype is intended for use with views that set the `vAnythingAllowed` mask.

This proto changes the recognition behavior of a view that allows recognition of various kinds of input. For example, the built-in Notepad application provides a `protoRecToggle` view that allows the user to change the recognition behavior of note views. Another common use of this proto is to facilitate changing between text recognition and ink text in an input view that supports both kinds of text.

The `protoRecToggle` view is designed to be added as a child of a status bar view based on the `protoStatus` proto. When used in this way, the `recToggle` view is positioned on the status bar automatically, and the value of its `viewBounds` slot is ignored. For example, the built-in Notepad application positions this view immediately to the right of the `protoInfoButton` view on the status bar.

When collapsed, the `protoRecToggle` view's appearance reflects the current configuration of the recognition system for the view that it controls. Figure 8-3 shows the `protoRecToggle` picker (popup menu) as it appears when collapsed and when expanded.

Figure 8-3 `protoRecToggle` picker collapsed and expanded



You can cause this view to display only those items that are appropriate for your application. For example, applications having only text entry fields typically display only the Text and Ink Text items. On the other hand, applications like the built-in Notepad have views that allow several different types of recognition within the note, and so display additional items in this picker. For more information, see the description of the `_recogPopup` slot, later in this section.

Applications that use a `protoRecToggle` view must provide a `_recogSettings` slot. When your application closes, it can save the contents of this slot and restore it the next time your application opens, thereby restoring the state of the `recToggle` view.

The `protoRecToggle` prototype provides the following slots of interest to developers:

Slot descriptions

`_recogSettings` Required; holds the current setting of the `protoRecToggle` view. When your application closes, it can save the value of this slot for use in restoring the state of the `protoRecToggle` view when the application opens again.

This slot may appear anywhere in the `_parent` chain of the view that the `recToggle` controls. For more information, see "Creating the _recogSettings Slot" (page 10-20) in *Newton Programmer's Guide*.

`defaultItem` Optional; an integer value specifying the element of the `_recogPopup` array to be used as the `recToggle` view's default setting. If this slot is missing or nil, the first element of the `_recogPopup` array specifies the default setting.

`_recogPopup` Optional; an array of symbols specifying the items to be included in the `protoRecToggle` picker. If this slot is missing or `nil`, all items specified in the `_recogPopup` slot of the `recToggle` view's template are included in the picker. The first item in this array is the default setting for the `recToggle` button.

The default `_recogPopup` slot provided by the `protoRecToggle` system prototype contains the array shown in the following code fragment:

```
_recogPopup:[
   'recogText,          // "Text"
   'recogInkText,       // "Ink Text"
   'pickSeparator,      // ------------------
   'recogShapes,        // "Shapes"
   'recogSketches,      // "Sketches"
   'pickSeparator,      // ------------------
   'recToggleSettings,  // "Preferences"
   ],
```

Your `_recogPopup` slot can contain any combination or subset of these symbols, in any order.

The next several paragraphs describe each of the symbols that may appear in the `_recogPopup` array.

`'recogText`

Specifies that the Text item is to appear in the `recToggle` picker. When this item is chosen, it enables text recognition as specified by any view flags, `recConfig` frame, or user preference settings that apply to the view controlled by the `recToggle` view. In views that set the `vAnythingAllowed` mask, this item enables the recognition of words, numbers, dates, times, and letters. For all views controlled by this item, unrecognized words are returned as ink text if the view supports ink text.

`'recogInkText`
: Specifies that the Ink Text item is to appear in the `recToggle` picker. When this item is chosen, written words are returned as unrecognized ink text.

`'recogShapes`
: Specifies that the Shapes item is to appear in the `recToggle` picker. When this item is chosen, it enables shape recognition for the view that the `recToggle` view controls and causes unrecognized shapes to be returned as sketch ink.

`'recogSketches`
: Specifies that the Sketches item is to appear in the `recToggle` picker. When this item is chosen, it disables recognition of text and shapes, causing input to be returned as sketch ink.

`'pickseparator`
: Specifies that an unselectable dotted line is to appear in the `recToggle` picker at the position corresponding to this array element.

`'recToggleSettings`
: Specifies that the Preferences item is to appear in the `recToggle` picker. When this item is chosen, it causes the system to display the Handwriting Recognition

preferences slip. Simply displaying the slip does not change any preferences.

Application-Defined recToggle Methods

The current system supports only one application-defined `recToggle` view method, the `RecogSettingsChanged` method, which is described here.

RecogSettingsChanged

view`:RecogSettingsChanged(`*viewFlags*`)`

This application-defined method provides a means of taking application-specific action in response to changes in the setting of an associated `recToggle` view. This message is sent when the `recToggle` picker changes if this method is defined in the `recToggle` view or anywhere in its `_parent` chain.

Edit views that set the `vAnythingAllowed` mask set use the new recognition settings automatically when this message is sent. Other kinds of views may need to take appropriate action themselves. This message is sent to `self` (which usually evaluates to the `recToggle` view), relying on parent inheritance for appropriate dispatch. Therefore, your implementation of this method must confine its actions to appropriate local changes only.

Typically, your `RecogSettingsChanged` method must add the value of the *viewFlags* parameter to any other appropriate nonrecognition view flags and place the resulting value in the `viewFlags` slot of any view that must respond to the change in the `recToggle` view's state. The new settings are used automatically because when the `recToggle` picker changes, the system calls the `PurgeAreaCache` function before sending the `RecogSettingsChanged` message.

viewFlags The current set of view flags to be used by the associated view for recognition. This value is passed to your `RecogSettingsChanged` method by the system. Note that this value does not include view flags unrelated to recognition, although the proper operation of the view may require them.

The system sets the value of the *viewFlags* parameter as follows:

- If the Text or Ink Text item was chosen in the `recToggle` picker, then the value of the *viewFlags* parameter is set to `vCharsAllowed` plus additional text recognition flags as appropriate.
- If the Shapes item was chosen in the `recToggle` picker, then the value of the *viewFlags* parameter is set to `vShapesAllowed`.
- If the Sketches item was chosen in the `recToggle` picker, then the value of the *viewFlags* parameter is set to `0`.

protoRecConfig

This prototype may be used to configure the recognition system when a particular configuration is not available through the use of view flags. It is also used to support features such as ink text and specialized behavior such as limiting the set of characters recognized by a view.

Your view's `recConfig` frame may be based on this proto or on one of the system-supplied `recConfig` frames (all of which are also based on this proto) described in "System-Supplied recConfig Frames" (page 8-18).

The value of the following slot affects the input mask that the view constructs:

Input mask slots

`inputMask` Required. The bit field that controls the view's recognition behavior. The recognition portion of the view's `viewFlags` slot should be set to the same value as the `inputMask` slot in the `recConfig` frame. There is one exception to this rule: to enable ink text, you can put the system-supplied `recConfig` frame `ROM_rcInkOrText` in your view's `recConfig` slot, leaving everything else the same.

The values of the following slots specify the choices that an associated `protoRecToggle` view provides to the user:

recToggle configuration slots

`allowTextRecognition`

A value of `true` specifies that text recognition is enabled when the value of the `doTextRecognition` slot in the system's user configuration data is `true`. (The system sets the `doTextRecognition` user configuration slot to `true` when the user chooses the Text item from the associated `recToggle` picker.) You might set this value to implement deferred recognition in a view that disables text recognition. User preferences not related to recognizer settings are not affected by the value of this slot.

`allowShapeRecognition`

A value of `true` specifies that shape recognition is enabled when the value of the `doShapeRecognition` slot in the system's user configuration data is `true`. (The system sets the `doShapeRecognition` user configuration slot to `true` when the user chooses the Shapes item from the associated `recToggle` picker.) User preferences not related to recognizer settings are not affected by the value of this slot.

The values of the following slots (or their inherited values) enable the use of a particular recognizer in views that set the `vAnythingAllowed` mask. Note that these slots are used rarely; normally, the bits in the `viewFlags` slot control the view's recognition behavior. The values of these slots can be used to override values inherited from system-wide settings or an associated `recToggle` view. These slots enable specified recognizers unconditionally—as opposed to the `allow`*Xxx*`Recognition` slots, which enable a specified recognizer only when the appropriate slot in user configuration data holds the value `true`.)

Recognizer configuration slots

`doTextRecognition`

The value `true` enables word recognition in the view that this `recConfig` frame controls. This slot is usually used only with views that set the `vAnythingAllowed` mask. When the user turns on text recognition from the `protoRecToggle` view associated with the view this `recConfig` frame controls, the system sets the value of the `doTextRecognition` slot in the system's user configuration data to `true`. This `recConfig` slot can be used to override values inherited from a `protoRecToggle` view or user configuration settings.

`doShapeRecognition`

The value `true` enables shape recognition in the view that this `recConfig` frame controls. This slot is usually used only with views that set the `vAnythingAllowed` mask. When the user turns on shape recognition from the `protoRecToggle` view associated with the view this `recConfig` frame controls, the system sets the value of the `doShapeRecognition` slot in the system's user configuration data to `true`. This `recConfig` slot can be used to override values inherited from a `protoRecToggle` view or user configuration settings.

`doInkWordRecognition`

The value `true` causes the recognizer to convert strokes to ink text rather than sketch ink. If the value of this slot is `nil` or the slot is absent, the view turns unrecognized ink into sketch ink. When the user turns on text or ink text recognition from the `protoRecToggle` view associated with the view this `recConfig` frame controls, the system sets the value of the `doInkWordRecognition` slot in the system's user configuration data to `true`. This `recConfig` slot can be used to override values inherited from a `protoRecToggle` view or user configuration settings. Note that the system may also return ink text to the view when the text recognizer cannot recognize the input successfully or when text and shape recognition are both disabled.

Do not attempt to include `letterSetSelection` or `learningEnabledOption` slots in your `recConfig` frame, for the following reasons:

- The text recognizer (printed or cursive) made available to all views is determined by the value of the `letterSetSelection` slot in the system's user configuration data. Individual views cannot override this system-wide setting.
- The system's ability to save learning data is enabled by the value of the `learningEnabledOption` slot in the system's user configuration data. Individual views cannot override this system-wide setting.

The following slots modify the behavior of the text recognizer:

Text recognizer configuration slots

`speedCursiveOption`
: This value affects the amount of time the cursive recognizer spends analyzing input. This value does not affect the printed recognizer. The user's preference (set by a slider in the Fine Tuning preferences slip) is used as the default value of this slot. This value ranges from 0 to 9, with 0 representing the slowest and most accurate recognition, and 9 representing the fastest and least accurate recognition.

Note
These slots are not guaranteed to affect all recognizers available in future versions of the system. ◆

`timeoutCursiveOption`
: This value affects the amount of time the recognizer waits from the completion of a stroke for subsequent strokes that might belong to the same character, word or shape. The value of this slot is a delay expressed in ticks (60ths of a second). The “Transform my handwriting” slider in the Fine Tuning user preferences slip sets values for this slot ranging from 15 ticks (.25 second) to

60 ticks (1 second). Your view can use larger or smaller values, although it is not recommended.

`letterSpaceCursiveOption`
The value of this slot affects the amount of horizontal space required to consider sets of strokes as belonging to separate letters or words. The user's preference (set by a slider in the Handwriting Recognition preferences slip), is used as the default value of this slot. This value ranges from 0 to 9, with 0 representing widely spaced words or characters, and 9 representing closely spaced words or characters. If the value of this slot is `nil`, the recognizer performs no segmentation.

The following slots affect the view's use of dictionaries for recognition.

Dictionary configuration slots

`dictionaries` Specifies custom dictionaries to be used by the view. This slot may contain a single dictionary identifier or an array of dictionary identifiers. When this slot is present, the view's `dictionaries` slot is ignored. Although not always necessary, it is still a good idea to set the `vCustomDictionaries` bit in the `recConfig` frame's `inputMask` slot when the `recConfig` frame provides a `dictionaries` slot.

`rcSingleLetters`
Set the value of this slot to `true` for a view that is to recognize only single letters. For example, this feature would be useful in a corrector view, in a crossword puzzle view, or when letters in a previously recognized word are overwritten. Note that you still need to provide a dictionary—in this case, one having entries that are single letters.

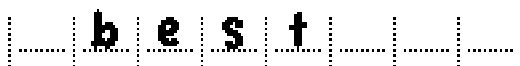
`inhibitSymbolsDictionary`
Set the value of this slot to `true` when the symbols dictionary is not to be included in the set of dictionaries

used by the view for text recognition. The symbols dictionary contains single letters, punctuation marks, and miscellaneous characters, and is normally enabled. It is used by the recognition system when the user overwrites single characters in a misrecognized word.

protoCharEdit

The `protoCharEdit` system prototype provides a comb-style entry view in which the user can edit text. The recognition system uses this proto as a means of allowing the user to correct single letters in a misrecognized word. The `protoCharEdit` system prototype is shown in Figure 8-4.

Figure 8-4 Typical `protoCharEdit` comb view and text to correct



In a `protoCharEdit` view, each character position that can be edited has a dotted line beneath it to indicate that it can be changed. The user can edit a character by overwriting it, causing the recognized value of the new character to be displayed in that position. When the user taps a cell in the comb view, it displays a picker containing alternate interpretations of the strokes which produced the character occupying that cell.

The user can delete one or more characters with the scrub gesture. Alternatively, the user can delete an individual character by tapping it and selecting the Delete item from the alternate interpretations picker that the comb view displays.

The user can insert a space for a new character with the caret gesture. Alternatively, the user can insert a space by tapping the position that the space is to occupy in the comb view and selecting the Insert item from the alternate interpretations picker that the comb view displays.

In addition to these gestures, the user can tap any blank space to display a list of punctuation marks that may be inserted at that position.

The comb view provided by the `protoCharEdit` view may be formatted or unformatted. In unformatted comb views, the word in the comb is of variable length. The user can delete any character, or insert new spaces anywhere. When a character is deleted, the surrounding characters move to close up the space formerly occupied by the deleted character. Although unformatted comb views usually accept any characters as input, it is possible to restrict input to a specified set of characters.

Words displayed in formatted comb views are restricted to a fixed length, and inserting additional characters is not allowed. Scrubbing characters in a formatted comb view clears them rather than deletes them; that is, the scrubbed character is replaced by a space. The set of characters recognized in each position may be restricted to a specified set. For example, a `protoCharEdit` view that holds a phone number is likely to restrict to numeric values the set of characters it returns.

The `protoCharEdit` prototype provides the following slots of interest to application developers. These slots are normally defined in your view template, used during initialization, and not changed subsequently:

Slot descriptions

`top` The screen coordinates of the top edge of the comb view; required when no `viewBounds` value is provided. If you provide the value of the `top` slot, you must also provide values for the `maxChars` and `left` slots.

`left` The screen coordinates of the left edge of the comb view; required when no `viewBounds` value is provided. If you provide the value of the `left` slot, you must also provide values for the `maxChars` and `top` slots.

`viewBounds` A standard `viewBounds` frame that specifies the dimensions of the comb view; required when the `top` and `left` values are not provided. If you provide the value of the `viewBounds` slot, the system provides the value of the `maxChars` slot for you.

`maxChars` — The number of character positions to display in the comb view. The default value is `8`. If you specify the values of the `top` and `left` slots, then you'll also need to specify the value of the `maxChars` slot. If instead you specify the value of the `viewBounds` slot, the value of the `maxChars` slot is calculated for you, based on the width of the view. In formatted comb views, the value of `maxChars` cannot be greater than the maximum number of characters allowed by the template.

`frameCells` — Optional. The value `true` specifies that the comb view displays gray divider lines between cells. The default value is `nil`.

`cellWidth` — The width of each cell in the comb view, expressed in pixels. This value must be an even number. The default value is `24`. If you specify the values of the `top` and `left` slots, then the width of the view is calculated as the value `(cellWidth*maxChars)+1`, and is set for you.

`cellGap` — The number of pixels of blank space between cells in the comb view. This value must be an even number. The default value is `6`. This value is used for drawing the cells and for determining the cells covered by a scrub gesture.

`viewLineSpacing` — The distance in pixels from the top of the `viewBounds` to the dotted line on which the user enters written input. The default value is `30`.

`cellHeight` — The total height of the cell, expressed in pixels. The default value is `50`. If you specify the values of the `top` and `left` slots, then the height of the view as expressed by its `viewBounds` value is set to the value of the `cellHeight` slot. If you specify the value of the `viewBounds` slot explicitly, the value of the `cellHeight` slot is set to the height expressed by the value of the `viewBounds` slot.

`recConfig` — The recognition configuration frame that specifies the recognition behavior of the comb view. The same recognition setting is used for all cells in the comb view. The default `recConfig` frame supplied as the value of this slot allows all standard characters to be recognized. To improve the speed and accuracy of a numbers-only comb view, you could change the `recConfig` frame in this slot appropriately. (For example, you might supply a custom dictionary containing only those digits that represent valid values.) If you change the value of this slot, you must ensure that the `ViewSetupDoneScript` method of the view is invoked afterward.

`template` — Optional frame used to customize the appearance and behavior of the comb view. For more information, see the section "Template Used by ProtoCharEdit Views" beginning on page 8-45.

`text` — The string to be displayed in the comb view. Initially, this slot contains the string to be displayed; after the `ViewSetupFormScript` method executes, this string may contain leading and trailing spaces.

`wordLeft` — The index of the leftmost character in the comb view that is not a space.

`wordRight` — The index of the cell to the right of the rightmost character in the comb view that is not a space.

`dispLeft` — In the `text` slot, the index of the character occupying the leftmost position in the comb view. The `dispLeft` slot normally has the value `0`, but after scrolling it may have values greater than zero.

`dispIndent` — The offset from the leftmost edge of the comb view to the leftmost edge of the first character position displayed, expressed in pixels.

Template Used by ProtoCharEdit Views

System-supplied templates for restricting input in `protoCharEdit` views to numbers, dates, phone numbers or times are described in "System-Supplied protoCharEdit Templates" beginning on page 8-46.

The optional template residing in your `protoCharEdit` view's template slot is a frame that may contain the following slots:

Slot descriptions

`filters` — Required when a `format` slot is provided. An array of one or more strings specifying characters that may be entered in cells of the `protoCharEdit` view. If your template does not provide a `format` slot, this array holds a single element that filters input for all cells in the `protoCharEdit` view. If you provide a `format` slot, this array can contain multiple elements. The `format` slot specifies indexes into this array that associate cells in the `protoCharEdit` view with elements of this array.

`format` — Optional. A string having one character for each position in the `protoCharEdit` view. Each ordinal position in this string specifies an index into the `filters` array to define permissible input in the corresponding ordinal position in the `protoCharEdit` view; any position holding an underscore specifies that the corresponding position in the `protoCharEdit` view cannot be edited.

The presence of the `format` slot specifies that the `protoCharEdit` view is a formatted comb view: it permits only a fixed number of characters; cells cannot be inserted or deleted; and scrubbing clears a cell in the comb rather than deleting it. If the `format` slot is missing or if its value is `nil`, the comb field is an unformatted comb view, like the corrector in the built-in Notepad application.

If the template has a `format` slot, then it must also provide a `filters` slot.

`text` — Optional. This string is used by the `SetupString` and `CleanupString` methods.

`SetupString` — Optional method you supply that provides a string value for the template's `text` slot. For more information, see this method's description in "Application-Defined protoCharEdit Template Methods" beginning on page 8-52.

`CleanupString` — Optional method you provide that processes the string obtained from the `text` slot before it is displayed in the comb view. For more information, see this method's description in "Application-Defined protoCharEdit Template Methods" beginning on page 8-52.

System-Supplied protoCharEdit Templates

This section describes system-supplied templates that can be used to filter input in `protoCharEdit` views. Place the appropriate template in your `protoCharEdit` view's `template` slot to restrict input to phone numbers, dates, times, or numeric values in general.

Note that the specific templates provided for filtering dates, times, or phone numbers may change according to the user's locale.

Phone Number Template

The template for phone numbers is stored in `GetLocale().phoneFilter`. This template lets the user enter phone numbers (excluding area code) in a format acceptable for the current locale. The area code must be entered in a separate input view.

Date Template

The template for dates is stored in `GetLocale().dateFilter`. This template lets you enter a date in *mm/dd/yy* format, with the specific order of these elements determined by the current locale bundle.

m	A digit representing the month.
d	A digit representing the day of the month.
y	A digit representing the year.

Time Template

The template for times is stored in `GetLocale().timeFilter`. This template lets you enter a time in *HH*:*MM* `[AM|PM]` format. For locales that use 24-hour time, the format is simply *HH*:*MM*.

H	A digit representing the hour.
M	A digit representing the minute.

Number Template

A general-purpose numeric template is defined in `ROM_numberFilter`. This template allows the user to enter a variable length integer containing only digits.

protoCharEdit Functions and Methods

The system provides the `protoCharEdit` functions and methods described here. Additionally, you can provide the optional methods described in "Application-Defined protoCharEdit Template Methods" on page 8-52, as well as the protoCharEdit template methods described in "Application-Defined protoCharEdit Template Methods" on page 8-52.

GetWordForDisplay

charEditView`:GetWordForDisplay()`

Returns a cleaned-up version of the string currently displayed by the comb view.

This is the best method to invoke to obtain a readable version of the string for external display—if the `protoCharEdit` view's template defines a `CleanupString` method, this function uses it to further modify the string returned by the `CurrentWord` method.

CurrentWord

charEditView`:CurrentWord()`

Returns the word currently displayed in the comb view, with leading and trailing spaces removed. Because unformatted comb views may add leading and trailing spaces to display strings, the string returned by this method may not be precisely the same as that residing in the `text` slot of *charEditView*.

Always use this method or the `GetWordForDisplay` function to retrieve the text from the comb view. The difference between these routines is that the `GetWordForDisplay` function calls the associated template's optional `CleanupString` method if it is provided.

DeleteText

charEditView`:DeleteText(`*left*`, `*right*`)`

Deletes the specified text from the comb view.

left	The index of the leftmost character to be deleted. This value may be obtained from the `protoCharEdit` view's `wordLeft` slot.
right	The index of the cell to the right of the rightmost character to be deleted. This value may be obtained from the `protoCharEdit` view's `wordRight` slot.

Normally, text is deleted from a `protoCharEdit` view when the user scrubs the text or chooses an item from the picker displayed when a character is tapped. To clear the entire view programmatically, you can use the `DeleteText` method as shown in the following code fragment.

view:`DeleteText(`*view*`.wordLeft,` *view*`.wordRight);`

Scroll

charEditView:`Scroll(`*direction*`)`

Scrolls the comb view left or right as specified. This function returns the value `true` when scrolling occurs.

direction	Integer indicating the direction to scroll. When this value is greater than zero, characters to the right of those currently displayed in the comb view are shown as necessary. When this value is less than zero, the display scrolls back to the beginning of the word (not to the next chunk to the left), as necessary.

UseTextAndTemplate

charEditView:`UseTextAndTemplate()`

Causes the comb view to use the current values of the `text` and `template` slots. Before using this method, you must set the `protoCharEdit` view's `text` and `template` slots to their new values.

You can use this method to change the text or template used by the comb view. It is not necessary to invoke this method when first opening the comb view, as its `ViewSetupFormScript` method provides equivalent initialization.

To change an already open comb view's text without changing its template, invoke the `SetNewWord` method, followed by the `UseNewWord` method. This approach provides better performance than the `UseTextAndTemplate` method does.

SetNewWord

charEditView:`SetNewWord(`*str*`, nil)`

Sets the string displayed in the comb view. This method is intended to be called after the `protoCharEdit` view's `ViewSetupFormScript` method has executed. After calling this method, you must call the `UseNewWord` method to make the comb view display the new string.

Because the `SetNewWord` method performs no reformatting, the string passed as its argument must be of the appropriate length and format. For example, you cannot clear a formatted comb view properly by passing `nil` as the value of the *str* parameter to this method.

For more information on clearing text from comb views, see the description of the `DeleteText` method. To change both the text and the template used by the comb view, call the `UseTextAndTemplate` method instead of the `SetNewWord` method.

str The new text to be displayed. This string must not contain leading or trailing spaces. If this string is to be displayed in a formatted comb view, it must be of the appropriate length and format—this method performs no reformatting.

`nil` For system use only; always set this value to `nil`.

UseNewWord

charEditView:`UseNewWord()`

Initializes the internal parameters of the `protoCharEdit` view as specified by the current values of its `text` and `template` slots. You must invoke this method after using the `SetNewWord` method to make the `protoCharEdit` view use new values for the `text` or `template` slots.

FixedWord

charEditView:`FixedWord()`

Returns `true` when the comb view's `template` slot holds a template that has a non-`nil` `format` slot. When this function returns `true`, characters are cleared rather than deleted; when it returns `false`, leading and trailing spaces are added to the displayed word, as necessary.

FixedWordLength

charEditView`:FixedWordLength()`

Returns the number of characters in the template's `format` slot. If this slot does not specify a format (specifically, when the `FixedWord` method returns `nil`) the `FixedWordLength` method returns `nil`.

MapAmbiguousCharacters

`MapAmbiguousCharacters(`*str*`)`

Replaces character codes for easily-misread glyphs (zero vs. letter *O*, numeric value 1 vs. letter *l*) in the *str* string with character codes that map to more readable glyphs.

IMPORTANT

This operation modifies the *str* argument directly. The modified *str* object is intended for display use only. The rest of the system is not notified of the modifications to this object. Do not rely on the remapped character codes in any way. ▲

str The string to modify; after this function returns, this parameter holds the modified string.

UnmapAmbiguousCharacters

`UnmapAmbiguousCharacters(`*str*`)`

Restores the *str* string modified by the `MapAmbiguousCharacters` function to its original, unmodified state.

str The string to unmap; after this function returns, this parameter holds the restored string.

Application-Defined protoCharEdit View Methods

This section describes the optional `DisplayExternal` and `SaveUndoState` methods that take application-specific action when the user edits text in the comb view or undoes edits to comb view text.

DisplayExternal

charEditView`:DisplayExternal(`*doIt*`)`

This message is sent when the text in the comb view is edited, either by overwriting a cell or by picking an alternate value from a cell's picker. Applications that maintain an externally-displayed view of the comb view's contents can use this method to respond to changes in the comb view.

doIt When this value is `nil`, you should not need to redraw.

SaveUndoState

charEditView`:SaveUndoState(`*appState*`)`

Called by the system to save the state of the comb view for undo operations. You can override this method to provide application-specific undo information. Your override must call the inherited `SaveUndoState` method, passing a frame holding your undo information as its argument; for example,

```
myCharEditView.SaveUndoState := func (appState)
begin
   local savedState := {myInfo : aValue, ...}
   inherited:SaveUndoState(savedState);
end
```

appState Frame containing your application's saved undo information.

Application-Defined protoCharEdit Template Methods

Your template can provide optional `SetupString` and `CleanupString` methods to manipulate the string the comb view displays.

SetupString

charEditTemplate:`SetupString(`*str*`)`

Optional method you provide which preprocesses the string in the comb view's `text` slot. This method formats the string passed as its argument as required for display in the comb view and then sets the value of the comb view's `text` slot to the newly formatted string.

If you provide this method, it is invoked when the `protoCharEdit` view opens and when the `SetTextAndTemplate` method is called.

str — The string on which this method operates. The system obtains this value from the `text` slot before this method is invoked.

CleanupString

charEditTemplate:`CleanupString(`*str*`)`

Optional method you provide which postprocesses the string in the comb view's `text` slot. For example, you can use this method to strip extraneous spaces or leading zeros from the string before it is displayed in the comb view.

If you provide this method, it is called by the `GetWordForDisplay` or `UseTextAndTemplate` methods when the text in the comb view changes.

str — The string on which this method operates. The system obtains this value from the `text` slot before this method is invoked.

protoCorrectInfo

The system holds correction information for recently-recognized words in a frame that you obtain by calling the `GetCorrectInfo` global function (page 8-56). You can send messages to this frame to retrieve and manipulate correction information for individual words. The methods described in this section use the *correctInfo* metasymbol to represent this frame, which is based on the `protoCorrectInfo` system prototype.

The *correctInfo* frame contains the following slots of interest to developers:

Slot descriptions

`info` — An array of `wordInfo` frames based on the `protoWordInfo` system prototype. Each frame stores the correction information for a recognized word. For more information, see "WordInfo Frame" beginning on page 8-30.

`max` — The maximum number of items for which the system stores `correctInfo` frames. Currently, the maximum value for this slot is `10`; it may change in the future.

As words are recognized, the system creates `wordInfo` frames and saves them in the `info` slot of the *correctInfo* frame. The current version of the system saves up to ten `wordInfo` frames at a time (as specified by the *correctInfo* frame's `max` slot). When the system must add an eleventh frame to this array, it extracts learning data from the oldest frame as necessary before discarding the oldest `wordInfo` frame.

The *correctInfo* frame provides the following methods of interest to developers. You do not need to call any of these methods yourself for views based on the `clParagraphView` class—such views provide all of these behaviors automatically. These methods are provided for the implementation of text support in views not descended from the `clParagraphView` class.

CorrectInfo Functions and Methods

These functions allow you to obtain and manipulate correction information for recognized words.

Offset

correctInfo`:Offset(`*view*`, `*start*`, `*oldSize*`, `*newSize*`);`

Repositions `wordInfo` frames in the *correctInfo*.`info` array. Usually, you do not need to call this method when *view* is based on the `clParagraphView` class; these kinds of views update the system's `correctInfo` frame for you automatically as the user edits text in the paragraph view.

When replacing, inserting, or deleting a range of text in a view not derived from the `clParagraphView` class, you may need to add, remove, or delete `wordInfo` frames from the `correctInfo` frame yourself. This method repositions existing `wordInfo` frames in the *correctInfo* frame to make room for your changes to the *correctInfo* frame.

view	The view in which text is being replaced. For purposes of moving *correctInfo* from one view to another—for example, during a drag and drop operation on some text—you can set this value to `0` to offset all the `wordInfo` frames in the `correctInfo` frame regardless of source view.
start	Offset into the text of the beginning of the range of text to replace.
oldSize	Size of the text to replace.
newSize	Size of the text to insert.

When you are inserting text in *view*, the value of *oldSize* is less than that of *newSize*. In this situation, the `Offset` method moves `wordInfo` frames associated with words to the right of the insertion point to the right within the `correctInfo.info` array—that is, it makes room in the array for the subsequent insertion of `wordInfo` frames associated with the inserted text.

When you are deleting text from *view*, the value of *newSize* is less than that of *oldSize*. In this situation, the `Offset` method moves `wordInfo` frames associated with words to the right of the deletion to the left within the `correctInfo.info` array—that is, it shifts existing array elements to close up empty space created by the removal of `wordInfo` frames associated with the deleted text.

When the range being replaced overlaps existing `wordInfo` elements, those elements are deleted.

When you delete a single space that lies between two words, this method merges the corresponding `wordInfo` frames.

When the value of the *view* parameter is `0`, all `wordInfo` frames in the `correctInfo` frame are offset, regardless of their source view.

GetCorrectInfo

`GetCorrectInfo()`

Returns the system-maintained correction information frame.

GetViewID

`GetViewID(`*view*`)`

Returns the specified view's unique identifier. This value is used to identify source and destination views when copying correction information between views.

view The view for which this function returns the identifier.

GetCorrectionWordInfo

`GetCorrectionWordInfo(`*wordUnit*`)`

Returns a `wordInfo` frame extracted from the word unit passed as its argument. You can use this function to inspect or alter the `wordInfo` frame from within your `ViewWordScript` method before a word is actually added to the paragraph view that provides the `ViewWordScript` method. This function creates a new `wordInfo` frame and caches it in the *wordUnit*, so that the same `wordInfo` frame can be used later to add `wordInfo` to the paragraph.

wordUnit The word unit passed to the `ViewWordScript` method. This object is valid only while the various recognition-related `View`*Xxx*`Script` methods are being called. Do not attempt to save units for later use.

RemoveView

correctInfo`:RemoveView(`*view*`);`

Deletes from *correctInfo* all frames having the same `viewID` value as *view*. This method is useful when an entire view is being deleted, and you want to delete all *correctInfo* information corresponding to that view.

view The view from which this method extracts a `viewID` value. This value is the same one returned by the `GetViewID` function.

Find

correctInfo`:Find(`*view*`, `*offset*`);`

Returns the `wordInfo` frame at the specified offset in the specified view.

view The view from which to extract a `wordInfo` frame. This view must contain a `text` slot holding the word to which the value of the *offset* parameter refers.

offset The number of characters from the beginning of the *view*`.text` slot to the first character of the word for which this method extracts a `wordInfo` frame.

FindNew

correctInfo`:FindNew(`*view*`, `*offset*`, `*length*`);`

Returns the `wordInfo` frame at the specified offset in the specified view. If this method does not find a `wordInfo` frame at the specified location, it creates a new `wordInfo` frame for the word, adds it to the *correctInfo* frame, and returns the new `wordInfo` frame.

view The view that performs text recognition, from which this method extracts a `wordInfo` frame. This view must contain a `text` slot holding the word referred to by the value of the *offset* parameter.

offset The number of characters from the beginning of the *view*`.text` slot to the first character of the word for which this method extracts a `wordInfo` frame.

length The number of characters in the word.

AddUnit

correctInfo`:AddUnit(`*view*`,` *start*`,` *stop*`,` *unit*`);`

Extracts the `wordInfo` frame from the specified unit and adds it to the *correctInfo* frame. This method does not move existing elements. If you are inserting or replacing text, you need to invoke the `Offset` method to adjust *correctInfo* before using the `AddUnit` method to add `wordInfo` frames to it. The `AddUnit` method can be called from the `ViewWordScript` method of *view* to write the word unit's `wordInfo` frame into the *correctInfo* frame.

view	The view into which text is being inserted. This view must contain a `text` slot holding the word referred to by the value of the *offset* parameter.
start	The number of characters from the beginning of the *view*`.text` slot to the first character of the word being inserted.
stop	The number of characters from the beginning of the *view*`.text` slot to the end of the word being inserted. This value is equal to the sum of *start* plus the length of the word.
unit	The word unit from which the `wordInfo` frame is extracted. This object is valid only during the recognition process—that is, while the various recognition-related scripts are being called. Do not attempt to save units for later use.

AddWord

correctInfo`:AddWord(`*wordInfo*`);`

Adds the specified *wordInfo* frame to the *correctInfo*`.info` array. This method doesn't do anything to the *wordInfo* or *correctInfo* frames—it just adds the *wordInfo* frame to the *correctInfo*`.info` array.

wordInfo	The `wordInfo` frame to add.

Clear

correctInfo`:Clear(`*view*`, `*offset*`, `*length*`);`

Deletes all `wordInfo` frames that overlap the specified range. If *view* is `nil`, this method removes any `wordInfo` frames that overlap the range, regardless of their originating view.

view	The view from which text is being deleted. This view contains a `text` slot that holds the word referred to by the value of the *offset* parameter.
offset	The number of characters from the beginning of the *view*`.text` slot to the first character of the word being deleted.
length	The number of characters in the word being deleted.

Extract

correctInfo`:Extract(`*view*`, `*start*`, `*stop*`);`

Creates a new correction information frame, copies all `wordInfo` frames overlapping the specified range in *view* into it, and returns the resulting frame. This method does not remove `wordInfo` frames from the *correctInfo* frame—normally, you use the `Offset` and `Clear` methods to do so when necessary.

The `Extract` method clones the entries that are copied, in case the receiver wants to offset them. This method is useful for supporting undo or drag and drop operations. For example, you could use it to copy correction information when dragging a text selection to a new view.

view	The view from which to extract text.
start	Offset to the beginning of the range of text to copy, expressed in characters from the beginning of the string in the *view*`.text` slot.
stop	Offset to the end of the range of text to copy, expressed in characters from the beginning of the string in the *view*`.text` slot.

Insert

destCorrectInfo`:Insert(`*srcCorrectInfo*`, `*destView*`);`

Inserts copies of all `wordInfo` frames in *srcCorrectInfo* into the *destCorrectInfo* frame.

destCorrectInfo	The *correctInfo* frame that is to hold the copied entries.
srcCorrectInfo	The *correctInfo* frame that holds the `wordInfo` frames to copy.
destView	The view into which text is being inserted.

In typical usage of this method, you would take the following steps:

1. Call the `Offset` method of the *destCorrectInfo* frame to create space to hold the new `wordInfo` frames.
2. Call the `Offset` method of the *srcCorrectInfo* frame to specify the range of text for which this method copies wordInfo frames.
3. Call the `Insert` method of the *destCorrectInfo* frame to copy the specified set of `wordInfo` frames

protoWordInfo

The `protoWordInfo` frame holds the correction information for a recognized word. This frame contains slots specifying the `clParagraphView` view that contains the word, the position of the word in its view, the alternate interpretations of the original input strokes that produced the word, and a reference to the recognizer that recognized the word. Optionally, this frame can also contain strokes, ink, and learning data.

Each `protoWordInfo` frame contains the following slots of interest to developers:

Slot descriptions

`ID`	For system use only. An integer used by `protoWordInfo` methods to identify the `clParagraphView` view in which the recognized word

	appears; this is the same value returned by the `GetViewID` function. Do not rely on this value for any operations not performed by `protoWordInfo` methods.
`start`	A zero-based index into the `text` slot of the `clParagraphView` view. This value specifies the position of the word's first character. You can determine the number of characters in the word by subtracting the value of the `stop` slot from the value of the `start` slot.
`stop`	A zero-based index into the `text` slot of the `clParagraphView` view. This value specifies the position of the space after the word's last character. You can determine the number of characters in the word by subtracting the value of the `stop` slot from the value of the `start` slot.
`flags`	Not documented; for system use only.
`unitID`	Not documented; for system use only.
`words`	The list of words returned by the recognition system as an array of `protoWordInterp` frames. The `protoWordInterp` system prototype is described in "protoWordInterp" beginning on page 8-63.
`strokes`	The stroke bundle associated with the word. This value is `nil` when no stroke data is present. For more information, see "The Stroke Bundle Frame" on page 8-28.
`ink`	The compressed ink representing the written word. This value is `nil` when no ink is present for the word. This slot is used rarely and this information is provided for debugging use only; commercial applications must not rely on this value.
`unitData`	Not documented; for system use only.

A typical `protoWordInfo` frame looks like the following code example.

```
[{id: 267, // ID of view that owns this data
   Start: 0, // first char’s offset into clParagraphView view
   Stop: 5, // last char’s offset into clParagraphView view
   flags: forSystemUseOnly, // do not use this slot
   unitID: forSystemUseOnly, // do not use this slot
// list of words & associated data returned by recognition system
   words: [ {word: "Lunch", score: 130, label: -1, index: 0},
            {word: "lunch", score: 0, label: -1, index: -2},
            {word: "Lunar", score: 290, label: -1, index: 1},
            {word: "Sundv", score: 300, label: -1, index: 2}],
// the original input’s stroke data
   strokes: {class: strokeBundle,
             bounds: {left: 176, top: 289, right: 338, bottom: 336},
             strokes: [<stroke, length 2040>]},
   unitData: forSystemUseOnly}, // do not use this slot
```

Note the negative values in the second interpretation of the word *lunch*. The `-1` value is the default value of the `label` slot and the `-2` value in the `index` slot indicates that the word was synthesized by the system; in other words, it’s an alternate capitalization, or something similar. Use these values for debugging purposes only; commercial applications must not rely on them.

WordInfo Methods

You can use the following methods to manipulate correction information encoded as `wordInfo` frames.

SetWords

wordInfo`:SetWords(`*words*`)`

Sets the list of words held in a `wordInfo` frame. For each element in the *words* array, this method clones the `protoWordInterp` frame and sets its `word` slot to the value of that array element.

words An array of strings.

GetWords

wordInfo`:GetWords()`

Returns an array of strings, one for each `wordInterp` frame stored in the *wordInfo*`.words` array.

AutoAdd

wordInfo`:AutoAdd()`

Adds the first item in the *wordInfo* frame's word list to the auto-add dictionary and the user dictionary. If the *wordInfo* frame has a non-`nil` `_noAutoAdd` slot, this method does nothing.

AutoRemove

wordInfo`:AutoRemove()`

Removes the first word in the *wordInfo* frame's word list from the user dictionary if that word was previously added by the `AutoAdd` method.

protoWordInterp

The `words` slot in the `protoCorrectInfo` frame stores an array of `protoWordInterp` frames returned by the recognition system. Each `protoWordInterp` frame contains data associated with a possible interpretation of the original stroke data. For an example of a typical `protoWordInterp` frame, see the `words` slot in the `protoWordInfo` code listing on page 8-62.

Each `protoWordInterp` frame containing the following slots:

Slot descriptions

`word` The text string to which the values of the other slots in this frame apply.

`score` An integer indicating the accuracy level of the match between this word and the original ink. A low score indicates a good match; conversely, a higher score indicates a poorer match.

`label` For system use only. The default value is `-1`. Use this value for debugging purposes only; commercial applications must not rely on it.

`index` An integer indicating the position of this word in the original list of matches returned by the recognition system. The word having the lowest `index` value is displayed at the top of the text-correction picker. This value is initialized to `-1`. A value of `-2` indicates that the word was synthesized by the system; in other words, it's an alternate capitalization or something similar. Use these values for debugging purposes only; commercial applications must not rely on them.

Recognition Functions

This section describes functions that you can use to configure the recognition system, control the display of electronic ink, access information in objects such as units and stroke bundles, manipulate various dictionaries, and implement your own form of deferred recognition.

Recognition Configuration Functions

These functions allow you to configure the recognition system dynamically.

ReadCursiveOptions

`ReadCursiveOptions()`

Reconfigures the recognition system dynamically, using the current values of user preferences for handwriting recognition. You must call this function to cause the recognition system to use the current settings after changing values in the system's user configuration data. You can also call this function after changing a view's view flags, entry flags, or `recConfig` frame; however, it's not absolutely necessary to do so: calling the `PurgeAreaCache` function is sufficient when user preferences have not changed.

PurgeAreaCache

`PurgeAreaCache()`

Recalculates recognition behavior for all views. Call this function when you have changed your view's `recConfig` frame, view flags, entry flags, or `dictionaries` slot. This function does not affect stroke recognition that began before it was called.

PrepRecConfig

`PrepRecConfig(`*recConfigFrame*`)`

Returns a RAM frame that references the specified template from its `_proto` slot and references recognition-related user configuration data from its `_parent` slot.

recConfigFrame The view's recognition configuration frame.

You can use this function to create a `recConfig` frame that can be modified at run time by placing the following code fragment in your view's `ViewSetupFormScript` method:

```
// prebuild editable recConfig frame
recConfig := PrepRecConfig(recConfig);
```

BuildRecConfig

`BuildRecConfig(`*view*`)`

Returns a `recConfig` frame that is configured exactly like the one used for recognition in the specified view. The frame that this function returns is intended for debugging use only—any changes you make to it are not applied to the view.

view	The view from which this function builds a `recConfig` frame.

Application-Defined Recognition Methods

These messages are sent to your view during pen input. Your view can supply the following optional methods to take action in response to these messages.

ViewClickScript

view`:ViewClickScript(`*unit*`)`

This message is sent when the user places the pen on the screen within the bounds of a view that has the `vClickable` flag set. This message is sent before the view system does any processing of the pen input.

The system does not necessarily send this message for every single pen tap. When recognizers that group strokes are enabled, this message is sent only once for each group.

unit	The unit (word unit, stroke unit, gesture unit, or shape unit) passed to the `ViewClickScript` method. This object is valid only while the various recognition-related `View`*Xxx*`Script` methods are being called. Do not attempt to save units for later use.

If the `ViewClickScript` method returns `true`, the pen interaction is considered to be complete: the system performs no further processing of the pen input and no other stroke-related messages are sent to the view (for example, `ViewStrokeScript`, `ViewGestureScript`, and so on).

If the ViewClickScript method returns the 'skip symbol, the view does not pass the ViewClickScript message up the _parent chain but sends all other messages that it normally would. You can return this symbol when you want to prevent clicks from falling through to other views while still passing strokes or gestures along. For information on how views handle pen input in general (rather than for recognition purposes) see "Handling Pen Input" (page 3-10) in *Newton Programmer's Guide*.

If the ViewClickScript method returns nil, the system continues to process the pen input. The message is passed up the parent view chain, until it is handled by a ViewClickScript method or ignored. If the ViewClickScript message is not handled and there are other recognition flags set, then additional system messages may be sent to the view. For example, if the vStrokesAllowed flag is set, then the ViewStrokeScript message may be sent; this may be followed by the ViewGestureScript message, if the vGesturesAllowed flag is set; and this may be followed by the ViewWordScript message, if word recognition is enabled.

You can determine the coordinates of the pen-down location by calling the GetPoint function from within your ViewClickScript method. To prevent the display of electronic ink in the view while tracking the pen, you can call the view methods TrackHilite or TrackButton or the global functions InkOff or InkOffUnHobbled.

Note that calling the Ticks function from within your ViewClickScript method provides the time when the ViewClickScript method was invoked instead of the time when the stroke began. To obtain accurate times for the beginning and the end of a stroke, your ViewClickScript method can call the GetUnitStartTime and GetUnitEndTime functions, respectively.

Attempting to use the GetUnitEndTime function before input is complete can produce unpredictable behavior—most often, a bus error. Your ViewClickScript method can use the StrokeDone function to determine whether the user has finished making the stroke. The following example uses the result returned by the StrokeDone function to condition its call to the Drag method. The Drag method tracks the pen on the screen automatically and drags the view to where the pen is lifted.

```
ViewClickScript := func(unit)
   begin
      startTicks := GetUnitStartTime(unit);
      while not StrokeDone(unit) do
         begin
            // drag the view when it’s tapped
            :Drag(unit, nil);
            // important to sleep in tight loops see note
            Sleep(1);
      end
      // do what you need to do with times here
      endTicks := GetUnitEndTime(unit);
   end
```

Note

Tight loops use power more heavily than normal operation does. To reduce power consumption significantly without sacrificing responsiveness, your loop can call the `Sleep` function with a value of 1 to 10 ticks as its argument. ◆

Here is another example; this code fragment captures all of the points in the stroke. Note that the `ViewStrokeScript` method is not suitable for such use, because it is not called until after a short delay; see "ViewStrokeScript" (page 8-69) for more information.

```
ViewClickScript: func(unit)
   begin
   // track the click until the stroke is finished
   loop
      begin
         while not StrokeDone(unit) do
            // sleep a little to save battery
            Sleep(10);
      end;
```

```
// use GetPointsArray(unit) to get points
// and save them somewhere
local points := GetPointsArray(unit);

// yes, we handled the click, so erase the ink
// and stop recognition for this stroke
true;
end,
```

ViewStrokeScript

view:`ViewStrokeScript(`*unit*`)`

This message is sent when the pen is first lifted after contacting the screen within the boundaries of the specified view (assuming the view has the `vStrokesAllowed` flag set). You can do any processing you want as a result of this event. The view system does no default processing as a result of this event.

The `ViewStrokeScript` message is sent to the view only the first time the pen is lifted during a stroke. If the pen is lifted more than once during a single stroke, only one `ViewStrokeScript` message is sent for that stroke.

The system does not necessarily send this message for every single pen tap. The system treats multiple pen-down/pen-up events that are close together in time as a single stroke (for writing letters and words); as a result, this message may not be sent for every stroke when the delay between strokes is not sufficient for the system to consider the strokes to be separate events. The amount of time required to consider strokes separate is a function of the speed of the processor and the recognition system, as well as the value of the `timeoutCursiveOption` user preference.

unit The stroke unit passed to the `ViewStrokeScript` method. This object is valid only while the various recognition-related `View`*Xxx*`Script` methods are being called. Do not attempt to save units for later use.

If the `ViewStrokeScript` method returns `true`, the pen interaction is considered complete: the system performs no further processing of the pen input, and no other stroke-related messages are sent to the view (for example, `ViewGestureScript`, `ViewWordScript`, and so on).

If the `ViewStrokeScript` method returns `nil`, the system continues to process the pen input. The message is passed up the parent view chain, until it is handled by a `ViewStrokeScript` method or discarded. If the stroke is not handled and other recognition flags are set for the view, then additional system messages may be sent to the view. For example, if the `vGesturesAllowed` flag is set, then the `ViewGestureScript` message may be sent. This message may be followed by the `ViewWordScript` message, if word recognition is enabled.

Note that this message is preceded by a `ViewClickScript` message if the view has defined such a method. To capture all of the points in a stroke, you need to use the `ViewClickScript` method, rather than the `ViewStrokeScript` method, because the `ViewClickScript` message is sent immediately when the pen is placed on the screen, whereas the `ViewStrokeScript` message is not sent until the stroke is complete. For a code example, see "ViewClickScript" beginning on page 8-66.

You can determine the coordinates of the stroke using the function `GetPointsArray`, as shown in the following code example. For more examples of the use of stroke units, see the description of the `ViewClickScript` method beginning on page 8-66.

```
ViewStrokeScript: func(unit)
   begin
      local bounds, points;
      bounds := StrokeBounds(unit);
      print("Bounds of stroke are "); print(bounds);
      points := GetPointsArray(unit);
      print("Points are "); print(points);
      true;
   end
```

ViewGestureScript

view:`ViewGestureScript(`*unit*, *gesture*`)`

When the `vGesturesAllowed` flag is set for a view, this message is sent when the user writes a recognizable gesture inside the view that the view does not handle automatically. Views based on the `clEditView` and `clParagraphView` classes handle standard gestures automatically; other kinds of views do not. Standard gestures include scrub, highlight, tap, double tap, caret, and line. To interpret gestures yourself in a `clView` view, you must set its `vGesturesAllowed` flag and provide a `ViewGestureScript` method.

Gestures are returned only for strokes that are temporally isolated from other strokes—that is, the system does not recognize strokes within a word as gestures. Similarly, strokes that immediately precede or follow other strokes are not recognized as gestures, either.

unit The gesture unit passed to the `ViewGestureScript` method. This object is valid only while the various recognition-related `View`*Xxx*`Script` methods are being called. Do not attempt to save units for later use.

gesture An integer code that identifies the gesture that was recognized. The following gestures are supported:

Gesture	Constant	Integer value
Tap	`aeTap`	49
Double tap	`aeDoubleTap`	50
Scrub	`aeScrub`	13
Highlight	`aeHilite`	47
Caret	`aeCaret`	15
Line	`aeLine`	16

This message is sent after the view system recognizes the gesture, and only if the gesture is one not normally handled by the view. For example, views of the `clParagraphView` class handle all gestures except a tap, so for this kind of view, the `ViewGestureScript` message will not usually be sent (except for pen taps). However, if you set the `vReadOnly` flag in the `viewFlags` slot, the `ViewGestureScript` message will be sent for all gestures except the highlight gesture.

Note

You can work around the limitation that this message is sometimes not sent. For example, you may want a view to receive this message regardless of what kind of view it is or what kinds of input it handles. To do this, create a child view of the `clView` class that is transparent and the same size as the input view. If you set the appropriate input flags for the `clView` view, it will receive the input-related messages first. For any particular message, the `clView` view can take some action and return `true` to prevent the message from being passed to the parent, or it can return `nil` to pass the message on to the parent. ◆

If the `ViewGestureScript` method returns `true`, the recognition system performs no further processing of this pen input and sends no additional recognition-related messages (for example, `ViewWordScript`) to the view .

If the `ViewGestureScript` method returns `nil`, the system continues to process the pen input. The message is passed up the parent view chain, until it is handled by a `ViewGestureScript` method or discarded. If the stroke is not handled and other recognition flags are set for the view, then additional system messages may be sent to the view. For example, if the `vGesturesAllowed` flag is set, the `ViewGestureScript` message may be sent. This message may be followed by the `ViewWordScript` message, when word recognition is enabled for the view.

Note that the `ViewGestureScript` message is preceded by a `ViewStrokeScript` message if the view has defined that method. The `ViewStrokeScript` message may be preceded by a `ViewClickScript` message if the view has defined such a method.

Here is an example of the use of the `ViewGestureScript` method. For more examples of the use of stroke units, see the description of the `ViewClickScript` method beginning on page 8-66.

```
ViewGestureScript: func(unit, gestureKind)
   begin
   if gestureKind = aeLine then  // If it was a line
      begin
      // Make a new data item in our app
      end;
   end
```

ViewWordScript

view`:ViewWordScript(`*stroke*`)`

This message is sent to the view performing text recognition when a word is recognized.

unit	The word unit passed to the `ViewWordScript` method. This object is valid only while the various recognition-related `View`*Xxx*`Script` methods are being called. Do not attempt to save units for later use.

You can get the word that was recognized by calling the function `GetWordArray`. The first string in the array that the `GetWordArray` function returns is the interpretation in which the recognizer has the highest confidence.

The `ViewWordScript` message is sent after the system recognizes the word, and only if the view is not one that normally supports word recognition. For example, views of the `clParagraphView` and `clEditView` class support word recognition, so they do not normally receive this message.

Note
You can work around the limitation that this message is sometimes not sent. For example, you may want a view to receive this message regardless of what kind of view it is, or what kinds of input it handles. To do this, create a child view of the `clView` class that is transparent and the same size as the input view. If you set the appropriate input flags for the `clView` view, it receives the input-related messages first. For any particular message, the `clView` view can take some action and return `true` to prevent the message from being passed to its parent, or it can return `nil` to pass the message on to its parent. ◆

If the `ViewWordScript` method returns `true`, the event is considered to be handled. If the `ViewWordScript` method returns `nil`, the message is passed up the parent view chain until it is handled by a `ViewWordScript` method or discarded. If no method handles the event, the unrecognized strokes are grouped into words and passed to the `ViewInkWordScript` or `ViewRawInkScript` methods.

Here is an example of the use of this method. For more examples of the use of stroke units, see the description of the `ViewClickScript` method beginning on page 8-66.

```
ViewWordScript: func(unit)
   begin
   local matchedWords, recognizedWord;
   matchedWords := GetWordArray(unit);
   recognizedWord := matchedWords[0];
   print("The recognized word was " & recognizedWord);
   true;
   end
```

Note
The system searches for this method only in the current view and its protos. The parent chain is not searched for the `ViewWordScript` method. ◆

ViewCorrectionPopupScript

textView:ViewCorrectionPopupScript(*pickForm*)

The ViewCorrectionPopupScript method provides a means of modifying or replacing the picker displayed when the user double-taps a previously recognized word. This method is not invoked if it has not been defined or if a keyboard view is open when the user double-taps the recognized word.

Your ViewCorrectionPopupScript method must return nil to cause the system to display the picker. For example, your method could insert or remove items in the list of alternate interpretations of the recognized word and return nil to display the modified picker.

Your ViewCorrectionPopupScript method can return true to suppress the display of the picker. For example, your method could provide an alternative user interface to correction information and return true to suppress the display of the picker that the system provides.

pickForm A frame containing information about the recognized word on which the user double tapped. The system builds this frame and passes it to your ViewCorrectionPopupScript method.

The *pickForm* frame contains the following slots:

bounds	The viewBounds of the refCon.form view. (see below)
wordInfo	The wordInfo frame for the originally recognized word. For more information, see "protoWordInfo" beginning on page 8-60.
pickItems	Alternate interpretations of the recognized word.
refCon	A frame describing the view that contains the word that was double-tapped.

The refCon frame contains the following slots:

form	The view containing the word that was double-tapped.

`wordOffset`
: Offset of the beginning of the recognized word, expressed as the number of 2-byte characters from the beginning of the `form` view's `text` slot. This value is similar to the offsets used for other text-related functions and methods.

`wordLength`
: The number of characters in the word.

The following code fragments illustrate typical `pickForm` and `refCon` frames.

```
pickForm := {
        bounds: nil,    // bounding box for word
        wordInfo: nil,  // wordInfo frame for word
        pickItems: nil, // array of words from wordInfo
        refCon: nil,    // frame describing view
         };

refCon := // frame describing view containing word
         {
        form:   nil,       // view containing word
        wordOffset: nil,   // offset of word within form
        wordLength: nil,   // length of word
};
```

Inker Functions

These functions allow you to control the display of electronic ink.

InkOff

`InkOff(`*unit*`)`

Turns off the display of electronic ink for the current stroke, which is referenced by the specified *unit*. This function's return value is unspecified.

This function is usually called from within a view's `ViewClickScript` method. You cannot call this function successfully from within a `ViewStrokeScript` method because the `ViewStrokeScript` message is not sent until the stroke is completed.

Note

This function reduces the tablet's sampling rate to conserve battery power and provide better performance in scrolling views. As a result, stroke information obtained after calling this function is inconsistent with that normally returned by the tablet; however, the reduced sampling rate is suitable for tracking the pen in most situations. If you need high-resolution point data, use the `InkOffUnHobbled` function to disable the inker. ◆

unit — The unit (word unit, stroke unit, gesture unit, or shape unit) passed to the `ViewClickScript` method when the user touches the pen to the screen. This object is valid only during the recognition process—that is, while the various recognition-related scripts are being called. Do not attempt to save units for later use.

InkOffUnHobbled

`InkOffUnHobbled(`*unit*`)`

Turns off the display of electronic ink for the stroke contained in the specified unit. This function does not reduce the tablet hardware's sampling rate. It is intended for use in situations requiring the suppression of inking while tracking the pen with a high degree of precision. This function's return value is unspecified.

This function is usually called from within a view's `ViewClickScript` method. You cannot call this function successfully from within a `ViewStrokeScript` method because the `ViewStrokeScript` message is not sent until the stroke is completed.

unit — The unit (word unit, stroke unit, gesture unit, or shape unit) passed to the `ViewClickScript` method when the user touches the pen to the screen. This object is valid only during the recognition process—that is, while the various recognition-related scripts are being called. Do not attempt to save units for later use.

SetInkerPenSize

`SetInkerPenSize(`*size*`)`

size The width of the pen, in pixels.

Sets the thickness of the electronic ink drawn on the screen. The pen size can range from 1 to 4 pixels wide; the system default is 2.

This function returns `nil` if the pen size was set successfully; otherwise, it returns an error code.

Note

This function only changes the width of ink as it is drawn by the system. To ensure that ink is properly displayed and updated under all circumstances, you must use the `SetUserConfig` function to set an appropriate value for the `userPenSize` slot in the system's user configuration data. After doing so, you need to pass this value as the argument to the `SetInkerPenSize` global function. ◆

The following code example sets the size of the pen to four pixels:

```
SetUserConfig('userPenSize, 4);
SetInkerPenSize(4);
```

Stroke Unit Functions

These functions operate on the objects passed to the `ViewClickScript`, `ViewStrokeScript`, `ViewGestureScript`, and `ViewWordScript` methods. These objects include stroke units, word units, gesture units, and shape units.

GetPoint

`GetPoint(`*selector, unit*`)`

Returns the specified point coordinates from the stroke contained in the specified unit. All points returned are in global (screen) coordinates.

If the stroke is in progress when this function is called, the coordinate of the last point read by the system (`finalX`, `finalY`, or `finalXY`) may not be the last point in the stroke. You can call the `StrokeDone` function to determine whether the stroke is complete.

selector Specifies which point coordinate is returned. This parameter may hold any of the following predefined values:

`firstX`	The X coordinate of the first point in the stroke.
`firstY`	The Y coordinate of the first point in the stroke.
`firstXY`	The X and Y coordinates of the first point in the stroke.
`finalX`	The X coordinate of the last read point in the stroke.
`finalY`	The Y coordinate of the last read point in the stroke.
`finalXY`	The X and Y coordinates of the last read point in the stroke

unit The unit passed to the `ViewClickScript` or `ViewStrokeScript` methods. This object is valid only during the recognition process—that is, while the various recognition-related scripts are being called. Do not attempt to save units for later use.

GetUnitStartTime

`GetUnitStartTime(`*unit*`)`

Returns the time, expressed in ticks, when the strokes comprising the object (word unit, shape unit, stroke unit, or gesture unit) encapsulated by the specified unit began.

unit	The unit passed to the `ViewWordScript`, `ViewStrokeScript`, and `ViewGestureScript` methods. This object is valid only while the various recognition-related `View`*Xxx*`Script` methods are being called. Do not attempt to save units for later use.

GetUnitEndTime

`GetUnitEndTime(`*unit*`)`

Returns the time, expressed in ticks, when the strokes comprising the object (word unit, shape unit, stroke unit, or gesture unit) encapsulated by the specified unit ended.

unit	The unit passed to the `ViewWordScript`, `ViewStrokeScript`, and `ViewGestureScript` methods. This object is valid only while the various recognition-related `View`*Xxx*`Script` methods are being called. Do not attempt to save units for later use.

StrokeDone

`StrokeDone(`*unit*`)`

Returns `true` if the stroke contained in the specified unit has been completed by the user (the pen has been lifted from the screen). Returns `nil` if the stroke is not yet completed.

unit	The stroke unit passed to the `ViewClickScript` method. This object is valid only while the various recognition-related `View`*Xxx*`Script` methods are being called. Do not attempt to save units for later use.

The following code fragment uses the `StrokeDone` function to determine whether the user has finished the current input stroke:

```
ViewClickScript : func(unit)
   begin
      while not StrokeDone(unit) do
      // do something here
      // important to sleep in tight loops - see note
      Sleep(1);
   end
```

Note

Tight loops use power more heavily than normal operation. To reduce power consumption significantly without sacrificing responsiveness, your loop can call the `Sleep` function with a value of 1 to 10 ticks as its argument. ◆

StrokeBounds

`StrokeBounds (`*unit*`)`

Returns a `viewBounds` frame describing the boundaries of the unit in its view. A `viewBounds` frame has this structure:

`{left:` *n1*`, top:` *n2*`, right:` *n3*`, bottom:` *n4*`}`

unit — The unit passed to the `ViewWordScript`, `ViewStrokeScript`, and `ViewGestureScript` methods. This object is valid only while the various recognition-related `View`*Xxx*`Script` methods are being called. Do not attempt to save units for later use.

GetPointsArray

`GetPointsArray(`*unit*`)`

Returns an array of points extracted from the specified unit. If the unit encapsulates multiple strokes, this function returns points from the first stroke.

The array that this function returns consists of coordinate pairs describing the points. The first element contains the Y coordinate of the first point, the second element contains the X coordinate, and so on. (Note that this is the reverse of the usual way that coordinate pairs are written.) Coordinates are global; that is, they are relative to the upper-left corner (0, 0) of the screen.

unit The unit passed to the `ViewWordScript`, `ViewStrokeScript`, and `ViewGestureScript` methods. This object is valid only while the various recognition-related `View`*Xxx*`Script` methods are being called. Do not attempt to save units for later use.

GetWordArray

`GetWordArray(`*unit*`)`

Returns an array of strings that are the recognition choices for the unit passed as its argument. The first element in the array is the word with the highest probability of matching the stroke that the user wrote (the highest score). The subsequent elements are alternate choices in descending order of matching confidence. Note that the “words” returned aren't necessarily alphabetical. They can be numbers, phone numbers, times, or any other kind of recognized characters.

unit The word unit passed to the `ViewWordScript` method. This object is valid only while the various recognition-related `View`*Xxx*`Script` methods are being called. Do not attempt to save units for later use.

GetScoreArray

`GetScoreArray(`*unit*`)`

Returns an array of numbers that are the recognition confidence scores for each of the words returned by `GetWordArray`. There is one score for each word. A score can range from 1 to 1000, with a lower number representing a higher recognition confidence.

unit The word unit passed to the `ViewWordScript` method. This object is valid only while the various recognition-related `View`*Xxx*`Script` methods are being called. Do not attempt to save units for later use.

Stroke Bundle Functions and Methods

This section describes the functions and methods you can use to work with stroke bundles.

ExpandInk

`ExpandInk`*(poly, format)*

Decompresses the ink in a polygon view and returns it as a stroke bundle.

poly	A `clPolygonView`, which is stored as a child of a `clEditView` and has an `ink` slot. You can test this by calling the `PolyContainsInk` function, which is described in "PolyContainsInk" on page 7-35.
format	The data resolution and filtering value. Use one of the values shown in Table 8-6 on page 8-29.

The stroke bundle returned by `ExpandInk` uses the same coordinate system and has the same bounds as the polygon view. Every point within the returned stroke bundle falls within those bounds.

If you expand ink at tablet resolution, the returned stroke bundle contains points that are at the highest resolution that can be derived from the compressed ink. If you expand ink at screen resolution, the points in the stroke bundle are spaced at a resolution approximately equal to screen resolution. The former expansion is suitable for recognition; the latter for display.

ExpandUnit

`ExpandUnit`*(unit)*

Creates a stroke bundle from information in *unit* and returns the stroke bundle, which uses global coordinate values.

unit An object that describes pen input information, as discussed in “Stroke, Word, and Gesture Units” on page 8-29. This is the object passed to one of the following application-defined view methods: the `ViewStrokeScript` method (stroke units), the `ViewWordScript` method (word units), or the `ViewGestureScript` method (gesture units).

Note that if you want a reference to the stroke bundle that is cached in a word unit, you should use the `GetCorrectionWordInfo` function, which returns a frame that contains the stroke bundle in a slot named `strokes`. For more information, see “GetCorrectionWordInfo” on page 8-56.

CompressStrokes

`CompressStrokes`*(strokeBundle)*

Compresses the *strokeBundle* and returns a polygon view.

strokeBundle A stroke bundle frame, as described in “The Stroke Bundle Frame” on page 8-28.

CountPoints

`CountPoints`*(stroke)*

Returns the number of points in *stroke* as an integer value.

stroke A binary object representing an ink stroke.

CountStrokes

`CountStrokes`*(strokeBundle)*

Returns the number of strokes in the stroke bundle as an integer value.

strokeBundle A stroke bundle frame, as described in “The Stroke Bundle Frame” on page 8-28.

GetStroke

`GetStroke`*(strokeBundle, index)*

Returns the stroke binary object specified by *index* from the strokes array in the *strokeBundle* frame.

strokeBundle	A stroke bundle frame, as described in "The Stroke Bundle Frame" on page 8-28.
index	An integer specifying a stroke in the stroke bundle array.

GetStrokeBounds

`GetStrokeBounds`*(stroke)*

Calculates the bounding rectangle for the specified *stroke*, and returns it as a frame.

stroke	A binary object representing an ink stroke.

GetStrokePoint

`GetStrokePoint`*(stroke, index, point, format)*

Copies the data from a specified point in a stroke into a new point.

stroke	A binary object representing an ink stroke.
index	An integer specifying the point in the stroke to copy.
point	A frame containing slots named `x` and `y`.
format	The data resolution and filtering value. Use one of the values shown in Table 8-6 on page 8-29. Note that the duplication filter is ignored by this function.

The `GetStrokePoint` function copies the data for the point in *stroke* specified by *index*. The data is copied into the *point* frame, using the resolution specified by *format*.

GetStrokePointsArray

`GetStrokePointsArray`*(stroke, format)*

Copies the data for all the points in *stroke* into an array. The points are filtered and scaled according to the value of the *format* parameter.

stroke	A binary object representing an ink stroke.
format	The data resolution and filtering value. Use one of the values shown in Table 8-6 on page 8-29.

The `GetStrokePointsArray` function returns a point array, as described in "Point Arrays" on page 8-30.

InkConvert

`InkConvert`*(ink, outputFormat)*

Converts the input *ink* to the specified format and returns the converted ink as a binary object. If *ink* is not a valid ink object, this function returns `nil`.

ink	A binary object that contains the ink to be converted.
outputFormat	A symbol that defines the conversion type. Use one of the following values:

`'ink`	The *ink* is converted to *1.x*-compatible ink.
`'ink2`	The *ink* is converted to *2.x* sketching ink.
`'inkword`	The *ink* is converted to *2.x* ink text.

MakeStrokeBundle

`MakeStrokeBundle`*(strokes, format)*

Creates a stroke bundle from an array of points.

strokes	An array of point arrays. The structure of each point array is described in "Point Arrays" on page 8-30. Each point array represents the coordinate data for a single stroke in the stroke bundle.

format The data resolution and filtering value for the point values in the *strokes* array. Use one of the values shown in Table 8-6 on page 8-29.

The `MakeStrokeBundle` function uses the coordinate data in *strokes* to create a stroke bundle and it returns that bundle. The input data is assumed to be in the resolution specified by *format*.

You can use the `MakeStrokeBundle` function to synthesize ink text for recognition; however, the quality of recognition is uncertain for such data. The recognizer generally requires high-quality, tablet-resolution data in order to produce accurate results.

MergeInk

`MergeInk`*(poly1, poly2)*

Decompresses the ink text in two polygons, recompresses them as a union of the original two polygons, and returns the resulting polygon. If a memory error occurs, `MergeInk` returns `nil`.

poly1, poly2 A `clPolygonView`, which is stored as a child of a `clEditView` and has an `ink` slot. Test this by using the method `PolyContainsInk`, which is described in "PolyContainsInk" on page 7-35.

The `MergeInk` function assumes that both of its arguments are polygon views containing ink text and that these views are horizontally adjacent with no intervening space.

PointsArrayToStroke

`PointsArrayToStroke`*(pointsArray, format)*

Creates a stroke from a point array.

pointsArray A point array, as described in the section "Point Arrays" on page 8-30.

format The data resolution and filtering value for the point values in the *strokes* array. Use one of the values shown in Table 8-6 on page 8-29.

The `PointsArrayToStroke` function creates a stroke from the coordinate data in *pointsArray* and returns the stroke object. The resolution of the input points is specified by *format*.

Note that the `PointsArrayToStroke` function is the inverse of the `GetStrokePointsArray` function, which is described in "GetStrokePointsArray" on page 8-86.

SplitInkAt

`SplitInkAt`*(poly, x, slop)*

Examines a polygon containing ink for a word break, splits the polygon at that word break, and returns an array of two polygons, each of which contains an ink word. The first array element is a polygon containing the first word, and the second element is a polygon containing the second word. If `SplitInkAt` cannot find a reasonable break, it returns `nil`.

Note
The `SplitInkAt` function never finds a word break in the middle of a stroke. ◆

poly A `clPolygonView`, which is stored as a child of a `clEditView` and has an `ink` slot. Test this by using the method `PolyContainsInk`, which is described in "PolyContainsInk" on page 7-35.

x An integer specifying the horizontal position near which this function looks for a word break.

slop An integer specifying how far in either direction (from *x*) to search for a word break. The recommended value for *slop* is somewhere between `xHeight` and `xHeight/2` for the word.

StrokeBundleToInkWord

`StrokeBundleToInkWord`*(strokeBundle)*

Converts the stroke bundle to an ink word. You can pass the resulting ink word object to the `HandleInsertItems` function, which is described in "HandleInsertItems" on page 7-54.

strokeBundle A stroke bundle frame, as described in the section "The Stroke Bundle Frame" on page 8-28.

Deferred Recognition Functions

These functions allow you to implement your own form of deferred text recognition.

RecognizePara

`RecognizePara(`*para*`, `*start*`, `*end*`, `*hilite*`, `*config*`)`

Recognizes ink in the paragraph view from the *start* index to the *end* index, replacing the ink with the recognized text. All ink within the range is converted. All text within the range is left as it is. This function returns an integer that indicates the new *end* value for the range.

para The `clParagraphView` view containing the ink to be recognized.

start Zero-based offset from the beginning of the paragraph to the first ink character to be recognized.

end Zero-based offset from the beginning of the paragraph to the last ink character to be recognized.

hilite The value `true` specifies that the view is to highlight each ink word as it is passed to the recognition system. If this value is `nil`, the words are not highlighted as they are recognized.

config A `recConfig` frame or `nil`. When a `recConfig` frame is passed as this value, *view* uses it to recognize the specified strokes. When this value is `nil`, this method uses the *para* view's default recognition settings.

RecognizePoly

`RecognizePoly(`*poly*`,` *hilite*`,` *config*`)`

Recognizes the ink in the *poly* view and replaces it in its parent view with the text returned by the recognition system.

poly — The `clPolygon` view containing the ink to be recognized.

hilite — A nonzero integer value specifies that the view is to highlight each ink shape as it is passed to the recognition system. If this value is zero, the shapes are not highlighted as they are recognized.

config — A `recConfig` frame or `nil`. When a `recConfig` frame is passed as this value, *view* uses it to recognize the specified strokes. When this value is `nil`, this method uses the *poly* view's default recognition settings.

Recognize

`Recognize(`*strokes*`,` *config*`,` *doGroup*`)`

Recognizes the strokes using the specified `recConfig` frame and returns a correction information frame.

strokes — An array of stroke bundles (as returned by the `ExpandUnit` function), or compressed ink (as returned by the `GetInkAt` function), in which each element of the array contains the strokes for a single word.

config — The `recConfig` frame to be used by this function.

doGroup — The value `true` specifies that the strokes are to be regrouped as part of the recognition process. The value `nil` specifies that the grouping specified by the stroke bundle or ink bundle is to be retained.

Dictionary Functions

These functions allow you to look up words in the built-in dictionaries, to manipulate the review dictionary, and to work with your own custom dictionaries.

GetRandomWord

`GetRandomWord(`*minLength*`,` *maxLength*`)`

Returns a string that is a word chosen at random from the common word dictionary. The string returned by this function is at least *minLength* characters long but not more than *maxLength* characters long. This function does not return any word longer than 20 characters.

The `GetRandomWord` function uses random numbers generated by the system to select words from the dictionary. To begin a new sequence of random words, you must first initialize the random number generator using the `SetRandomSeed` function. To repeat a sequence of words, pass to the `SetRandomSeed` function the same argument that was used to generate the original sequence of random words. You need only call `SetRandomSeed` once to begin a new sequence of random words. For more information, see the discussion of the `SetRandomSeed` function in Chapter 26, “Utility Functions,” in *Newton Programmer’s Guide*.

minLength The minimum number of characters in words returned by this function.

maxLength The maximum number of characters in words returned by this function. This function does not return words longer than 20 characters, regardless of the value specified by this argument.

LookupWordInDictionary

`LookupWordInDictionary(`*dictID*`,` *word*`)`

Returns `true` if the specified word or an alternate capitalization form of the word is present in the specified dictionary. This function returns `nil` if the word is not found.

Note

This function does not strip punctuation from *word* before searching for it in the specified dictionary. ◆

dictID The dictionary identifier specifying the dictionary to be searched.

word The string to be found in the specified dictionary. This string must not contain more than 32 characters.

DeleteWordFromDictionary

`DeleteWordFromDictionary(`*dictID*`, `*word*`)`

Removes the specified word from the RAM-based dictionary indicated by *dictID*, returning `true` if the word is removed and `nil` if it is not. A `nil` result usually indicates that the specified word was not found in the specified dictionary, but it also may indicate an error. For example, it is an error to call this function on a static dictionary.

dictID The dictionary identifier specifying the dictionary to be searched.

word The string to be removed from the specified dictionary.

NewDictionary

`NewDictionary(`*dictionaryKind*`)`

Creates a new RAM-based dictionary and returns a dictionary ID for it. The dictionary ID is used in the other custom dictionary functions.

dictionaryKind Specifies how the dictionary is to be used. Currently, only the symbol `'custom` has any meaning as an argument to this function. If you pass `'custom`, the dictionary is used for recognition only in views where it is specified in a `dictionaries` slot in conjunction with the `vCustomDictionaries` view flag. For more information, see "Using Your RAM-Based Custom Dictionary" (page 10-28) in *Newton Programmer's Guide*.

Note

Although the token returned by the `NewDictionary` function currently evaluates to an integer in the NTK Inspector, the type of value returned by this function may change on future Newton devices. Do not rely on the `NewDictionary` function returning an integer. ◆

DisposeDictionary

`DisposeDictionary(`*dictionary*`)`

Deletes the specified RAM-based dictionary. This function's return value is unspecified.

dictionary	The dictionary to be deleted

AddWordToDictionary

`AddWordToDictionary(`*dictionary*, *wordString*`)`

Adds the specified word to the specified RAM-based dictionary, returning `true` if the word was added successfully. If the word could not be added, this function returns `nil`. This function does not strip punctuation before adding *wordString* to *dictionary*.

IMPORTANT

Do not use the `AddWordToDictionary` function to add words to the personal word list or user dictionary. Instead, use the `AddWord` method of the `ReviewDict` object for this purpose. ▲

dictionary	The dictionary to which this function adds the specified string.
wordString	The string to be added to the specified dictionary. This string must not contain more than 32 characters.

GetDictionaryData

`GetDictionaryData(`*dictionary*`)`

Returns a binary object or virtual binary object representing the specified dictionary's word list. To save dictionary data in a soup, place the object that this function returns in a slot in a frame that you add to a soup. For a code example, see "Saving Dictionary Data to a Soup" (page 10-27) in *Newton Programmer's Guide*.

dictionary The dictionary from which this function extracts data.

SetDictionaryData

`SetDictionaryData(`*dictionary*, *binaryObject*`)`

Retrieves dictionary data (words) from the specified binary object and loads them into the specified dictionary. You can use this function to populate a blank dictionary with dictionary items stored in a soup. This function's return value is unspecified.

dictionary The dictionary into which this function loads data.

binaryObject The binary object or virtual binary object from which this function extracts data.

User Dictionary Functions and Methods

This section describes methods available from the review dictionary object in the root view. You can send messages to this object to manipulate the user dictionary and the expand dictionary.

You can use code similar to the following to get a reference to the review dictionary object:

```
local reviewDict := GetRoot().reviewDict;
```

Note
Future versions of the system are not guaranteed to have this slot. You must verify that the returned value is non-`nil` before using it. ◆

Open

reviewDict`:Open()`

Displays the Personal Word List slip. If there are items in the auto-add dictionary, this method displays the Recently Written Words slip along with the Personal Word List slip.

AddWord

reviewDict`:AddWord(`*word*`)`

Adds the specified word to the user dictionary. If the word is added successfully, this method returns the value `true`. You must call the `SaveUserDictionary` method to make changes persistent. If the word is already in the user dictionary or one of the standard system dictionaries, then the word is not added and the return value of this method is unspecified.

This method updates the display in the Personal Word List slip automatically if it is open. An undo action is posted for this operation. For performance reasons, the dictionary is not flushed to the internal store for each word that is added.

word	The string to be added to the user dictionary. This string may be capitalized or contain punctuation.

RemoveWord

reviewDict`:RemoveWord(`*word*`)`

Removes the specified word from the user dictionary. This method returns `true` if the word was removed successfully; otherwise it returns `nil`. If the Personal Word List is open, the display is updated automatically. An undo action is posted for this operation. For performance reasons, the changed dictionary is not written to the system soup; after calling this function, you must call the `SaveUserDictionary` function to make dictionary changes persist.

word The word to be removed from the user dictionary. If this string's case does not match that of the string in the user dictionary exactly the dictionary entry is not removed.

LoadUserDictionary

`LoadUserDictionary()`

Loads the review dictionary into RAM from the system soup.

Most flags that enable text recognition include this dictionary automatically in the set of dictionaries available to the view performing text recognition. Therefore, you usually do not need to call this function yourself—the system calls it whenever the Personal Word List slip is opened or the system is reset.

SaveUserDictionary

`SaveUserDictionary()`

Writes the user dictionary to the system soup, saving any changes that have been made to the dictionary. You must call this function to make review dictionary changes persistent.

AddExpandWord

reviewDict:`AddExpandWord(`*word*, *expandedWord*`)`

Adds a word and its expanded version to the expand dictionary. The word must be recognized before it can be expanded, so you must first invoke the `AddWord` method to add the word to the user dictionary. If the word is not already in the expand dictionary and can be successfully added, the `AddExpandWord` method returns the value `true`; otherwise, its return value is unspecified. If the Personal Word List slip is open, the display is updated automatically. An undo action is posted for this operation. For performance reasons, this method does not write the changed dictionary to the internal store.

word The abbreviated version of *expandedWord* to be added to the expand dictionary.

expandedWord The word to be added to the user dictionary.

GetExpandedWord

reviewDict`:GetExpandedWord(`*word*`)`

Looks for the specified word in the expand dictionary and returns the expansion if the word is found. If the word is not found in the expand dictionary, the return value of this method is unspecified.

word The word to be found in the expand dictionary.

LoadExpandDictionary

`LoadExpandDictionary()`

Loads the expand dictionary from the internal store into RAM.

SaveExpandDictionary

`SaveExpandDictionary()`

Writes the expand dictionary from RAM into the internal store.

Auto-Add Dictionary Functions and Methods

This section describes functions and methods you can use to manipulate the auto-add dictionary. This dictionary is accessible from the root view. You can use code similar to the following example to get a reference to this object:

```
local autoAdd:= GetRoot().autoAdd;
```

Note

Future versions of the system are not guaranteed to have this slot. You must verify that the returned value is non-`nil` before using it. ◆

This section describes messages you can send to this object to manipulate the auto-add dictionary.

Open

autoAdd`:Open()`

Displays the Recently Written Words slip.

AddAutoAdd

`AddAutoAdd(`*word*`)`

Adds the specified word to the auto-add and user dictionaries if the base word (*word* without punctuation) is not present in the set of dictionaries enabled by the `vCharsAllowed` flag. This function returns `true` when the word is added successfully. You can determine the base word by passing *word* to the `LookupWord` global function.

If the auto-add dictionary already contains the maximum number of words allowed (currently 100), this function displays the notify icon, posts a notify action and does not add the new word. For information about the notify icon and notify actions, see "Additional System Services" (page 17-1) in *Newton Programmer's Guide*

Note
When the printed recognizer is enabled, new words are not added to the user dictionary automatically. ◆

RemoveAutoAdd

`RemoveAutoAdd(`*word*`)`

Removes the specified word from the user and auto-add dictionaries.

User Configuration Functions

These functions allow you to manipulate recognition-related user preferences data; for example, your application can use these functions to save and manage recognition settings for multiple users. These functions are supported only on version 2.0 of the Newton OS. These functions manage only recognition-related user preference settings; they have no effect on any other user preferences.

GetUserSettings

`GetUserSettings ()`

Returns a frame containing the current user recognition settings. The frame this function returns is suitable for use as the argument to the `SetUserSettings` function. Do not modify the frame this function returns, or rely on any values you may find in it. The format and vlaues in this frame may change in future versions of the system.

SetDefaultUserSettings

`SetDefaultUserSettings ()`

Sets recognition-related user preference settings to their default values.

SetUserSettings

`SetUserSettings (`*savedSettings*`)`

Sets user preferences for recognition as specified.

savedSettings	Recognition preferences frame returned by the `GetUserSettings` function.

Data Storage and Retrieval Reference

This chapter describes objects, data structures, functions, and methods used for data storage and retrieval on Newton devices. This chapter begins with a description of important data structures, including soup definition frames, index specification frames, query specification frames, soup change notification callback functions, and package reference information frames. Subsequent sections provide descriptions of functions and methods grouped according to the topic with which they are most closely associated, such as stores, virtual binary objects (VBOs), soups or union soups, tags, soup changes, queries, cursors, soup entries, and mock entries.

Data Structures

This section describes data structures related to the Newton data storage system, including soup definitions and specification frames for single-slot indexes, multiple-slot indexes, and tags indexes. This section also describes query specification frames, tags query specification frames, callback functions for change notification, and package reference information frames.

Soup Definition Frame

Soup definition frames are used to create soups on demand and to provide information about soups to the system, to other applications, and to the user. This section describes the slots present in soup definition frames. For a description of how to use soup definitions, see “Registering and Unregistering Soup Definitions” (page 11-33) in *Newton Programmer’s Guide*.

The soup definition frame specifies the soup’s name, its user-visible name, the application to which it belongs, descriptive strings used to present information to the user, and a default set of indexes to be created along with any soup created from this definition.

The soup definition frame contains the following slots:

Slot descriptions

`name` — Required. A string that identifies the soup to the system. This string must be unique among the names of all soups on the store. For more information about naming soups, see “Naming Soups” (page 11-32) in *Newton Programmer’s Guide*.

`userName` — Required. A string that is the user-visible name for this soup; for example, this string is displayed as the soup’s name in the Extras Drawer.

`ownerApp` — Required. The application symbol identifying the application to which this soup belongs. For more information about application symbols, see “Application Symbol” (page 2-11) in *Newton Programmer’s Guide*

`ownerAppName` — Required. The user-visible string identifying the application to which this soup belongs.

`userDescr` — Required. A string that is the user-visible description of this soup. This string provides information about the purpose of the soup and the data it contains; for example, this string might advise the user not to delete the soup accidentally.

`indexes` Required. An array of index specification frames. Each frame in this array describes one index in the default set of indexes with which this soup is created. For detailed descriptions of index spec frames, see "Single-Slot Index Specification Frame" (page 9-5) and "Multiple-Slot Index Specification Frame" (page 9-6).

`initHook` Optional. Any time this soup definition is used to create a member of a union soup, the system executes the `initHook` function specified by this slot. This slot can hold a symbol or a callback function object. If this slot holds a function object, the function is executed directly; otherwise, the symbol it contains is sent to the base view of the application specified by the value of the `ownerApp` slot of the soup definition. That application must define in its base view a slot containing a function to be executed when this message is sent.

The `initHook` function is meant to provide a means of seeding a new soup with initial values. If your `initHook` function uses auto-transmit methods such as `AddXmit`, it must pass `nil` as the second argument to these functions to suppress the transmission of soup change notification messages. Note that the system sends a `'soupCreated` notification after it executes your `initHook` function.

Regardless of whether the function resides in the `initHook` slot or the application's base view, it must be of the following form:

`myFn := func (`*soup*`,` *soupDef*`) begin … end`

soup The soup on which this function operates. This value is always a single soup, not a union soup; thus your `initHook` function should not invoke union soup methods.

soupDef The soup definition from which *soup* was created; also, the soup definition that defines this `initHook` function.

The following code fragment provides an example of a typical `initHook` function:

```
MyInitHookCallback: func( soup, soupDef )
   begin
      soup:AddFlushedXmit( {aSlot: "Entry A"}, nil );
      soup:AddFlushedXmit( {aSlot: "Entry B"}, nil );
   end;
```

For related information, see the description of the NewtApp framework's `newtSoup` object, which provides methods for creating soups and filling them with entries.

A typical soup definition looks like the following code fragment:

```
local aSlotIndexSpec := {structure: 'slot, path: 'aSlot,
                        type: 'string};
local bSlotIndexSpec := {structure: 'slot, path: 'bSlot,
                        type: 'int};

local mySoupDef :=
{  // string that identifies this soup to the system
   name: "myApp:mySig",
   // string that is user visible name
   userName: "My Application soup",
   // application symbol
   ownerApp: '|myApp:mySig|,
   // user-visible name of app that owns this soup
   ownerAppName: "My Application",
   // user-visible string describing soup
   userDescr: "This soup is used by My Application.",
```

```
    // array of indexSpec frames - default indexes
    indexes: [aSlotIndexSpec, bSlotIndexSpec, … ]
    // optional function used to initialize the soup
    initHook: symbolOrCallBackFn
}
```

Single-Slot Index Specification Frame

This section describes the slots present in single-slot index specification frames. For general information about index specification frames, see “Indexes” (page 11-8) in *Newton Programmer’s Guide*. For a description of how to use index specification frames, see “Registering and Unregistering Soup Definitions” (page 11-33) and “Adding an Index to an Existing Soup” (page 11-36), in *Newton Programmer’s Guide*.

The index specification frame specifies the kind of index to create, the slot from which to extract index key values, and the type of data found in the index key slot.

The index spec frame contains the following slots:

Slot descriptions

`structure` Required. Specifies whether the soup is indexed on a single slot or on multiple slots. For a single-slot index, this value must be the `'slot` symbol.

`path` Required. A path expression specifying the slot from which index key values are extracted. For a complete explanation of path expressions, see *The NewtonScript Programming Language*.

IMPORTANT
You cannot use a value stored in a virtual binary object as an index key. ▲

`type` Required. A symbol specifying the type of data stored in the index key slot. For integer values, specify `'int`; for string values, specify `'string`; for character values, specify `'char`; for real number values, specify `'real`; and for symbolic values, specify `'symbol`.

`order` Optional. Specifies the sorting order for the index; the only permissible value is either the `'ascending` or `'descending` symbol. If this slot is missing or has the value `nil`, the index keys are assumed to be in ascending order.

`sortID` Optional. The value `1` specifies the use of the alternate sort table in ROM, which provides a case and diacritic sensitive sort order suitable for non-English language strings. If this slot is missing or has the value `nil`, the default sort table is used. For more information, see "Indexes" (page 11-8) in *Newton Programmer's Guide*.

A typical single-slot index spec looks like the following code fragment:

```
{
   // must use this value - index keys are slot values
   structure:'slot,
   // entries indexed on this slot
   path: pathExpr,
   // data type found in the indexed slot
   type: symbol,
   // optional. 'ascending or 'descending
   order: symbol,
   // optional. pass 1 to use alternate sort table
   sortID: nil
}
```

Multiple-Slot Index Specification Frame

This section describes the index description frame for a multiple-slot index. Each multiple-slot index can index soup entries on a total of up to six key values. For more information about multiple-slot indexes, see "Querying on Multiple-Slot Indexes" (page 11-47) in *Newton Programmer's Guide*. For descriptions of how to use index specification frames to create soup indexes, see “Using Soups” (page 11-32) and “Adding an Index to an Existing Soup” (page 11-36), in *Newton Programmer’s Guide*.

The multiple-slot index specification frame specifies the kind of index to create, the slots from which to extract index key values, and the types of data found in those slots.

The multiple-slot index spec frame contains the following slots:

Slot descriptions

`structure` — Required. Specifies whether the soup is indexed on a single slot or on multiple slots. For a multiple-slot index, this value must be the `'multiSlot` symbol.

`path` — Required. An array of path expressions specifying the slots from which index key values are extracted. The first element in the array contains the path to the primary key, the second element contains the path to the secondary key, and so on. Each multiple-slot index allows a total of 6 index paths.

Note

The `path` and `type` arrays must have the same number of elements. ◆

`type` — Required. An array having any of the symbols `'string`, `'char`, `'int`, `'real` or `'symbol` as its elements. Each element of this array specifies the type of the data stored in the slot specified by the corresponding element of the `path` array in this index spec frame.

`order` — Optional. An array of any of the possible values `'ascending` or `'descending`. Each element of this array specifies the sorting order for the key stored in the corresponding element of the `path` array in this index description frame. If the `order` array is missing, all index keys are assumed to be in ascending order.

`sortID` Optional. The value `1` specifies the use of the alternate sort table in ROM, which provides a case- and diacritic-sensitive ranking suitable for non-English languages. If this slot is missing or has the value `nil`, the default sort table is used. For more information, see “Indexes” (page 11-8) in *Newton Programmer’s Guide*.

A typical multiple-slot index spec looks like the following code fragment.

```
{
   // index keys are multiple slot values
   structure: 'multiSlot, // must use this value
   // up to six path expressions specifying indexed slots
   path: [pathExpr1, pathExpr2, … , pathExpr6],
   // data type found in each indexed slot
   type: [sym1, sym2, … sym6]
   // optional. 'ascending or 'descending
   order: [sym1, sym2, … , sym6 ]
   // optional. pass 1 to use alternate sort table
   sortID: nil
}
```

Tags Index Specification Frame

The tags index stores the tag symbols associated with the entries in a soup. This index is defined by the tags index specification frame described here. The tags index specification frame specifies the kind of data to index (in this case, symbols used as tags), the slot from which to extract this data, and the kind of index to create (in this case, a tags index.)

▲ **WARNING**
Each soup has only one tags index; if you add a tags index to a soup or union soup that already has one, it replaces the original tags index. ▲

The tags index spec frame contains the following slots:

Slot descriptions

`structure` Required. Specifies whether the soup is indexed on a single slot or on multiple slots. For a tags index, this value must be the `'slot` symbol.

`path` Required. A path expression specifying the slot from which index key values are extracted. In this case, the index key values are tags, so this expression specifies the slot in which this soup stores its tags.

`type` Required. A symbol specifying the type of the data stored in the index key slot. For a tags index, this value must be the `'tags` symbol.

A typical tags index spec frame looks like the following code fragment:

```
{
   // must use this value - tags are slot values
   structure: 'slot,
   // index values (tags) extracted from this slot
   path: pathExpr,
   // must use this value
   type: 'tags,
}
```

Query Specification Frame

A query specification frame (or query spec) is passed as the argument to the `Query` method of soups or union soups. The query spec describes the criteria that soup entries must meet to be included in the set of entries returned by the cursor that this query generates. To retrieve every entry in a soup, pass `nil` as the argument to the `Query` method instead of passing a query spec frame. For more information regarding queries and their results, see “Queries” (page 11-10) in *Newton Programmer’s Guide*.

A query spec frame includes the following slots; missing slots or missing elements in a slot are presumed to be `nil` values:

Slot descriptions

`indexPath` Required. This value specifies the path to the slot in each entry that holds its index key value. Search results are sorted according to the value of the slot specified by this value.

`beginKey` Optional. Specifies the key value defining the beginning of the range over which the cursor generated by this query iterates. Each end of the range may be inclusive or exclusive of a given key value; that is, you can specify *key* `>= beginKey`, *key* `> beginExclKey`, *key* `<= endKey`, or *key* `< endExclKey`. Either end of the range may be unspecified, in which case the range extends all the way to that end of the index. You can't specify both the inclusive and exclusive forms of the same end of the range.

`beginExclKey` Optional. This value specifies a key value to exclude from the beginning of the range over which the cursor generated by this query iterates. This slot specifies the beginning of a range of key values, just as the `beginKey` slot does, but the value of the `beginExclKey` slot is not included in the range of key values over which the query searches. For more information, see the description of the `beginKey` slot on page 9-10.

`endKey` Optional. Specifies the key value at the end of the range over which the cursor generated by this query iterates. For more information, see the description of the `beginKey` slot on page 9-10.

`endExclKey` Optional. This value specifies a key value to exclude from the end of the range over which the cursor generated by this query iterates. This slot specifies the end of a range of key values, just as the `endKey` slot does, but the value of the `endExclKey` slot is not included in the range of key values over which the query searches. For more information, see the description of the `beginKey` slot on page 9-10.

`tagSpec` — Optional. Contains a tags query specification frame as described in "Tags Query Specification Frame" (page 9-13)

`words` — Optional. One or more strings to match with word beginnings in any slot in an entry. Single strings can be passed as they are or as the sole element of an array. Multiple strings must be passed as the elements of an array. This query does not match any strings in the middle of a word. Because each element in the array is a string, each "word" in a `words` query can actually contain multiple words and punctuation. A `words` query is not case sensitive. If you specify multiple array elements, each string in the `words` array must appear somewhere in the entry for it to be included in the query result.

`entireWords` — Optional. The value `true` specifies that the query is to match the entire string in the `words` slot instead of matching strings beginning with the string in the `words` slot.

`text` — Optional. A string for which the query searches. This search is not confined to word boundaries; that is, the search string is found if it appears anywhere in any string in any slot in an entry.

`indexValidTest` Optional. A developer-supplied function that tests key values passed to it and returns a non-`nil` value if the corresponding entry is to be included in the query result. The `indexValidTest` slot contains a function of the form

`indexValidTest:= func (`*args*`) begin … end;`

args This value is a single index key for queries on single-slot indexes. This value is an array of keys for queries on multiple-slot indexes.

The system invokes the `indexValidTest` method before the `validTest` method.

Note that in the following situations the input to the `IndexValidTest` function may not exactly match the entry's actual index key:

Keys of type `'string` are truncated after 39 Unicode characters (80 bytes, 2 of which are used internally).

Ink data is stripped from `'string` keys.

Subkeys in multiple-slot indexes may be truncated or missing when the total key size is greater than 80 bytes.

For more information, see "Limitations of Index Keys" (page 11-52) in *Newton Programmer's Guide*.

`validTest` Optional. A developer-supplied function accepting a soup entry as its argument. The function must return any non-`nil` value for an entry that is to be included in the set of entries returned by the cursor, and return `nil` for an entry that is not to be included in the set of entries returned by the cursor. The use of an `indexValidTest` is preferable to the use of a `validTest`, for performance reasons.

The `validTest` slot contains a function of the form

`validTest:= func (`*entry*`) begin … end;`

entry The entry to be tested.

`secOrder` When the soup being queried has an index that provides internationalized sorting order, the value `True` specifies that cursor find operations such as `GoToKey` are sensitive to case and diacritical values in strings. For more information, see "Indexes" (page 11-8) in *Newton Programmer's Guide*.

The following code fragments provide examples of typical query spec frames:

```
myQSpec := nil // return all entries

// return entries having a name slot
myQSpec := { indexPath : 'name}

// return entries with both "Bob" and "Apple" in any slot
myQSpec := {words : ["Bob","Apple"]}

// return (10 ≥ entry.myIntegerSlot < 190) entries
myQSpec := {indexPath : 'myIntegerSlot}
            beginKey : 10,
            endExclKey : 190}

// return entries having even values in 'myIntegerSlot
myQSpec := {indexPath : 'myIntegerSlot}
            indexValidTest: func (key) (key MOD 2) = 0}
```

Tags Query Specification Frame

The **tags query specification** frame or **tags query spec** described here specifies the use of tags by the `Query` method of soups or union soups. This frame is placed in the `tagSpec` slot of the query spec frame presented as the argument to the `Query` method. In addition to specifying the tags on which to test, the `tagSpec` frame specifies how the query is to use the specified tags; for example, whether the query includes or excludes entries having the specified tags.

The `tagSpec` slot contains a frame holding one or more of the following slots, each of which contains a symbol, an array of symbols, or the value `nil`. The order in which symbols appear in the array is unimportant.

Slot descriptions

`equal` The entry's tags must equal the set of tags specified by this slot. Entries with additional tags or missing tags are not matched. Note that `equal: []` returns all nontagged entries.

`all` The entry's tags must include all tags specified by this slot. Entries having additional tags are included in the query result as well.

`none` Entries having none of the tags specified in this slot (including entries that have no tags) are included in the query result.

`any` Entries having one or more of the tags specified in this slot are included in the query result.

The following code fragment illustrates the use of a simple tags query spec. For additional examples of the use of tags query specs, see "Querying for Tags" (page 11-42) in *Newton Programmer's Guide*.

```
// match ("my text") AND ('tree OR 'flower)
soup:Query({text:"my text",
            tagSpec: {any:['tree, 'flower]}});
```

Callback Functions for Soup Change Notification

This section describes the callback function that your application can register with the soup change notification mechanism. For more information about soup change notification, see "Introduction to Data Storage Objects" (page 11-2) and “Using Soup Change Notification” (page 11-63), in *Newton Programmer’s Guide*.

Your callback function is passed as the value of the *callBackFn* parameter to the `RegSoupChange` global function. The `RegSoupChange` global function registers this callback to be executed in response to changes in a specified soup. Note that your callback function must not call the `RegSoupChange` or `UnRegSoupChange` functions.The value your callback function returns is ignored by the current system.

Your callback function must be of the form

`func(`*soupNameString*`,`*appSymbol*`,`*changeTypeSymbol*`,`*changeData*`);`

IMPORTANT

This callback function must not call the `RegSoupChange` or `UnRegSoupChange` functions. ▲

soupNameString A string that is the name of the soup that changed.

appSymbol A unique symbol identifying the application that caused the change. If this information is not available, the system passes the `'_unknown` symbol to your callback function.

changeTypeSymbol A symbol indicating the kind of change that occurred; for possible values, see Table 9-1, immediately following.

changeData The data that changed. The data passed as this argument varies according to the value of the *changeType* parameter; for more information, see Table 9-1, immediately following.

Table 9-1 Change messages and associated change data

changeTypeSymbol	When sent	changeData
`'entryAdded`	Entry added to soup or union soup.	The new entry added to the soup.

Table 9-1 Change messages and associated change data (continued)

changeTypeSymbol	When sent	changeData
`'entryRemoved`	Entry deleted from soup or union soup.	A frame having the soup the entry came from in its `oldSoup` slot and the (former) entry that was removed in its `entry` slot. For example: `{oldSoup:`*theSoup*`,entry:`*theEntry*`};`
`'entryChanged`	Any change to entry data.	The changed soup entry.
`'entryMoved`	Entry moved from one soup to another.	A frame having the soup the entry came from in its `oldSoup` slot and the entry that moved in its `entry` slot. For example: `{oldSoup:`*theSoup*`,entry:`*theEntry*`}`
`'entryReplaced`	Entry replaced with another.	A frame holding the entry that was replaced in its `oldEntry` slot and the replacement entry in its `entry` slot. For example: `{oldEntry:`*oldOne*`,entry:`*newOne*`}`
`'soupInfoChanged`	Any change to soup information frame.	The soup that changed.
`'soupEnters*`	Soup becomes available to union soup; for example, because card inserted.	The soup that became available to the union soup.

Table 9-1 Change messages and associated change data (continued)

changeTypeSymbol	When sent	changeData
`'soupLeaves*`	Soup becomes unavailable to union soup; for example, because card removed.	The soup that is no longer available; don't use this soup, as it is invalid when this message is sent.
`'soupCreated`	New soup created.	The soup that was created.
`'soupDeleted`	Existing soup deleted.	The store from which the soup was removed.
`'soupTagsChanged`	Several tags changed. Send only when tags are added by the `AddTags` method; otherwise, it's unnecessary.	The soup that changed.
`'soupIndexAdded`	New soup index or tags index added.	A frame having the new version of the soup in its `soup` slot and the new index path in its `index` slot; for example, `{soup:`*reIndexedSoup*`,index:`*newIndexPath*`}`
`'soupIndexRemoved`	Existing soup index or tags index removed.	A frame having the new version of the soup in its `soup` slot and the removed index path in its `index` slot; for example, `{soup:`*reIndexedSoup*`,index:`*indexPath*`}`

Table 9-1 Change messages and associated change data (continued)

changeTypeSymbol	When sent	changeData
'whatThe	Multiple changes to soup, or 1.x application made change.	Value is unspecified. Used when it's impractical to report all of the individual changes to a soup; also used by 1.x applications that still use the obsolete BroadCastSoupChange function.

* This message may not be sent for soups that are not in use. For example, if no cursor object references the soup, this message may not be sent.

Package Reference Information Frame

The GetPkgRefInfo function provides information about a specified package by returning an information frame containing the following slots of interest to NewtonScript developers. Do not rely on the values of any slots in this frame that are not documented here; they are for system use only and subject to change without notice.

Slot descriptions

size	An integer specifying the package's uncompressed size, expressed in bytes.
store	The store on which the package resides.
title	The string that is the name of the package.
version	The integer that is the version number of the package.
timeStamp	The date and time the package was installed, expressed as an integer returned by the Time global function.
creationDate	An integer specifying the date the package was created.
copyProtection	Non-nil value specifies that the package is copy protected.
dispatchOnly	Non-nil value specifies that this package is a dispatch-only package.
copyright	Copyright information string.

`compressed`	Non-`nil` value specifies that the package is compressed.
`cmprsdSz`	Integer specifying the compressed size of package, expressed in bytes.
`numParts`	Integer specifying the number of parts in the package.
`parts`	Array of parts comprising this package. If the package is not active, references to these parts (or objects in them) may be invalid. Do not access parts of inactive packages.
`partTypes`	Array of part type symbols; each element in this array specifies the part type of the corresponding element in the `parts` array.

Data Storage Functions and Methods

This section describes all Newton data storage functions and methods. Methods are listed under the object that defines them or the object on which they operate. Global functions are listed under the object on which they operate.

Package Functions and Methods

A **package** is an object that encapsulates code, scripts, and resources as a Newton application; for more information, see “Packages” (page 11-7) in *Newton Programmer’s Guide*.

The following functions and methods allow you to work with packages.

GetPackageNames

`GetPackageNames(`*store*`)`

Returns an array having elements that are the names of all packages present on the specified store, including inactive (frozen) packages.

store	The store this function tests.

GetPackages

`GetPackages()`

Returns an array of packages currently active in the Newton system. Each array element is a frame containing the following slots:

Slot descriptions

`id`	An integer that identifies this package to the system.
`size`	An integer that is the uncompressed size of the package, expressed in bytes.
`title`	A string that is the name of the package.
`store`	The store on which the package resides.
`version`	An integer that is the package's version number.
`timeStamp`	The time the package was installed, expressed as an integer returned by the `Time` global function.
`copyProtection`	Non-`nil` value specifies that this package is not to be replicated by the Newton system.

IMPORTANT

Because the current package installation process is not reentrant, you cannot call the `GetPackages` function from your part's `InstallScript` function or `RemoveScript` function. (The system calls these functions in the process of installing or removing a package.) ▲

GetPkgRef

`GetPkgRef(`*name*`,` *store*`)`

Returns a `pkgRef` reference to the specified package on the specified store.

name	The string that is the name of the package.
store	The store on which the package resides.

GetPkgRefInfo

`GetPkgRefInfo(`*pkgRef*`)`

Returns a frame containing information about the specified package. For a complete description of this frame, see "Package Reference Information Frame" (page 9-18).

pkgRef — Package reference specifying the package for which this function returns information.

SuckPackageFromBinary

store`:SuckPackageFromBinary(`*binary*`, `*paramFrame*`)`

Creates a package from the specified binary object's data and installs the new package on the specified store.

binary — The binary object supplying this package's data.

paramFrame — The value `nil` or a frame containing information used to build the package. When this value is non-`nil` it is a frame that may contain the following slots and values:

`callbackFrequency`
: The number of bytes to read before executing the callback function again; set to `0` when no callback function is supplied.

`callback`
: Optional callback routine; set to `nil` if no callback is supplied. This callback is commonly used for the implementation of a progress indicator. The callback function must be a function object of the form

```
func(callbackInfo)
begin
   //do something w/callbackInfo
end
```

callbackInfo
A frame containing the following slots:

`packageSize`
Number of bytes in the package.

`numberOfParts`
Number of parts in the package.

`packageName`
Name of the package.

`currentPartNumber`
Index of the part currently being read.

`amountRead`
Number of bytes of the package total read so far.

SuckPackageFromEndpoint

store`:SuckPackageFromEndpoint(`*endPoint*`,` *paramFrame*`)`

Creates a package using data read from the specified endpoint and installs the new package on the specified store.

endPoint The endpoint supplying this package's data.

paramFrame The value `nil` or a frame containing information used to build the package. This frame may contain the following slots and values:

`callbackFrequency`
The number of bytes to read before executing the callback function again; set to `0` when no callback function is supplied.

`callback` Optional callback routine; set to `nil` if no callback is supplied. This callback is commonly used for the implementation of a progress indicator. The callback function must be a function object of the form

```
func(callbackInfo)
begin
   //do something w/callbackInfo
end
```

callbackInfo
A frame containing the following slots:

`packageSize`
Number of bytes in the package.

`numberOfParts`
Number of parts in the package.

`packageName`
Name of the package.

`currentPartNumber`
Index of the part currently being read.

`amountRead`
Number of bytes of the package total read so far.

IsPackage

`IsPackage(`*obj*`)`

Returns a non-`nil` value if the object to be tested is a package reference; otherwise, returns `nil`.

obj The object to be tested.

IsPackageActive

`IsPackageActive(`*pkgRef*`)`

Returns a non-`nil` value when the specified package is active; otherwise, returns `nil`.

pkgRef	A `pkgRef` reference, as returned by the `GetPkgRef` global function.

IsValid

`IsValid(`*obj*`)`

Returns the value `true` if its argument references an object in valid memory. Returns `nil` for invalid objects such as references to objects residing on a card that is no longer available. This function always returns the value `true` for immediate objects. (For a complete list of NewtonScript immediate objects, see *The NewtonScript Programming Language*.) Note that soup and store objects supply their own `IsValid` methods; do not use the global function `IsValid` to test these kinds of objects.

obj	The object to be tested.

▲ **WARNING**
This function tests only whether the object passed as its argument resides in valid memory; it does not follow references that the object may contain. Thus, its use does not cause the display of the "Newton needs the card" slip. However, if the object to be tested is a frame containing a slot that references an object on a storage card that has been removed, the frame itself may test valid even though it contains an invalid reference. In this situation, you would need to use the `IsValid` function to test each slot in the frame recursively to find the slot containing the invalid reference. ▲

ObjectPkgRef

`ObjectPkgRef(`*obj*`)`

Returns a package reference to the package containing the specified object. This function returns `nil` if the object does not reside in a package or the object is a NewtonScript immediate. (For a complete list of NewtonScript immediate objects, see *The NewtonScript Programming Language*.)

obj The NewtonScript object to be tested.

MarkPackageBusy

`MarkPackageBusy(`*pkgRef*`, `*appName*`, `*reason*`)`

Marks the specified package as busy. Any attempt to perform an operation that deactivates a busy package (such as moving or removing it) causes the display of a warning that allows the user to cancel the operation before the package is deactivated. However, if the user proceeds with the operation that deactivates the busy package, your application must handle resultant error conditions gracefully. This function's return value is unspecified.

You should mark a package as busy if its deactivation will cause serious problems; for example, a store part that provides critical data may be marked busy while it is in use. To avoid inconveniencing the user, you must call the `MarkPackageNotBusy` function as soon as possible after calling the `MarkPackageBusy` function.

pkgRef The package on which this function operates. This value is a package reference returned by the `GetPkgRef` global function.

appName String describing the entity requiring the package. Usually this value is the string returned by the `GetAppName` function.

reason Warning string displayed to the user.

MarkPackageNotBusy

`MarkPackageNotBusy(`*pkgRef*`)`

Reverses the effects of the `MarkPackageBusy` function. This function's return value is unspecified. To avoid inconveniencing the user, you must call the `MarkPackageNotBusy` function as soon as possible after calling the `MarkPackageBusy` function.

pkgRef The package on which this function operates. This value is a package reference returned by the `GetPkgRef` global function.

SafeMovePackage

`SafeMovePackage(`*pkgRef*`,` *destStore*`)`

Moves the specified package to the specified store. If the package is busy, this function warns the user to cancel the operation before deactivating the package. (Moving a package requires that it be deactivated, moved, then reactivated.) This function's return value is unspecified.

pkgRef The package on which this function operates. This value is a package reference returned by the `GetPkgRef` global function.

destStore The store to which the specified package is moved.

▲ **WARNING**

Do not call this function from your application part's `InstallScript` function or `RemoveScript` function. ▲

SafeRemovePackage

`SafeRemovePackage(`*pkgRef*`)`

Removes the specified package. If the package is busy, this function warns the user to cancel the operation before deactivating the package. (Removing a package requires that it be deactivated first.) This function's return value is unspecified.

▲ **WARNING**
Do not call this function from your application part's `InstallScript` function or `RemoveScript` function. ▲

pkgRef — The package on which this function operates. This value is a package reference returned by the `GetPkgRef` global function.

SafeFreezePackage

`SafeFreezePackage(`*pkgRef*`)`

Deactivates the specified package until it is activated by the `ThawPackage` function. The `SafeFreezePackage` function's return value is unspecified.

▲ **WARNING**
Do not call this function from your application part's `InstallScript` function or `RemoveScript` function. ▲

pkgRef — The package on which this function operates. This value is a package reference returned by the `GetPkgRef` global function.

ThawPackage

`ThawPackage(`*pkgRef*`)`

Reverses the effects of the `SafeFreezePackage` function. The `ThawPackage` function's return value is unspecified.

pkgRef — The package on which this function operates. This value is a package reference returned by the `GetPkgRef` global function. The package this value represents must have been deactivated previously by the `SafeFreezePackage` function.

Store Functions and Methods

A **store** is a logical data repository on a physical storage device. For more information, see "Introduction to Data Storage Objects" (page 11-2) and "Stores" (page 11-6) in *Newton Programmer's Guide*.

You can use the functions and methods described in this section to

- get information about currently available stores
- get and set the information frame that describes the store and its contents
- create soups
- write soups and packages to a store
- get lists of soups present on a store
- execute multiple operations as a single transaction with respect to a store

AtomicAction

store`:AtomicAction(`*myAction*`)`

Executes the *myAction* function as a transaction, meaning that if its operations do not all succeed, the changes to *store* caused by *myAction* are undone and the store is returned to the state it was in before the *myAction* function executed.

In order to provide this service, the system caches the changes made by the *myAction* function before making them permanent. Therefore, you must avoid doing large amounts of work from within the *myAction* function or the `AtomicAction` method will fail due to insufficient cache space.

Changing a small number of logically related entries falls within this method's intended use, while changing every entry in a soup does not. For example, you might change the Names soup entries for the company name of all the members of a company as an atomic action—that way, if an error occurs, you are ensured that the entries are not left in an inconsistent state (where some members of the company have the old name and some have the new name).

myAction Application-defined function object accepting no arguments.

BusyAction

store`:BusyAction(`*appSymbol*`,` *appName*`,` *myAction*`)`

Calls the *myAction* function object with the store marked busy until the *myAction* function returns. Unlocking the card switch on a store marked busy causes the "Newton needs the card…" slip to be displayed. The `BusyAction` method returns the result of the *action* function.

appSymbol Unique symbol identifying the application that posted the busy action.

appName String displayed in the "Newton needs the card…" slip as the user-visible name of the application that posted the busy action.

myAction Application-defined function object accepting no arguments. The system calls this function with the store marked busy until the function returns.

CheckWriteProtect

store`:CheckWriteProtect()`

Throws an exception if the store is locked or in ROM. The return value of this method is unspecified.

This method throws `|evt.ex.fr.store|(-48020)` when the store is in ROM. If the store is not in ROM, but is write protected, this method throws `|evt.ex.fr.store|(-10605)`. Contrast with the `IsReadOnly` store method, which returns a non-`nil` value when a specified store can't be written.

You can use this function to test whether the store can be written before executing lengthy operations. For an operation that completes quickly, you may prefer to attempt the operation and catch exceptions as they occur. The following code fragment provides an example of the use of this function:

```
//exit if we can't write
myStore:CheckWriteProtect();
// perform potentially lengthy operation
local mySpecialFn := func ()
   begin
      myUSoup := GetUnionSoupAlways(ROM_CardfileSoupName);
      myCurs := myUsoup:Query({beginKey: "Apple", endKey:"Apple");
      while e := myCurs:Entry() do
         // do something to every "Apple" entry here
   end;
myStore:BusyAction('|myApp:mySig|,
                     GetAppName('|myApp:mySig|),
                     mySpecialFn);
```

CreateSoupXmit

store:`CreateSoupXmit(`*soupName*`,` *indexArray*`,` *changeSym*`)`

Creates a soup called *soupName* on the specified store, returns a reference to the newly created soup object, and transmits a soup change notification. Any existing union soups with the same name are updated to include the newly created soup, as are any existing cursors. The soup this method creates does not have a soup information frame.

soupName — A case-insensitive string up to 39 characters long that specifies the name with which the soup is to be created. This name must be unique among all soups on the store.

indexArray — An array of index specification frames or `nil`. For more information, see “Indexes” (page 11-8) in *Newton Programmer’s Guide*, For detailed descriptions of various kinds of index spec frames, see “Single-Slot Index Specification Frame” (page 9-5), “Multiple-Slot Index Specification Frame” (page 9-6), and “Tags Index Specification Frame” (page 9-8) in *Newton Programmer’s Reference*.

changeSym — A unique symbol identifying the application that created the soup; usually this value is the application symbol or some variation on it. Pass `nil` as the value of this parameter to avoid transmitting a soup change notification.

Erase

store:`Erase()`

Erases all contents of the specified store. This method's return value is unspecified.

GetAllInfo

store:`GetAllInfo()`

Returns the store's information frame. This special-purpose method is intended for use by backup/restore applications only; most applications need not use it. Unless an application stores data in this frame, it may not exist on every store. Applications can use the `GetInfo` store method to get their own slot from the store's information frame. For more information, see the description of the `GetInfo` method.

GetInfo

store:`GetInfo(`*slotSymbol*`)`

Returns the contents of the specified slot in the store's information frame. Unless an application stores data in it, the information frame may not exist on every store. This function returns `nil` if the store information frame does not exist. Applications can create a slot in the information frame to store card data, such as the last time the application encountered a particular card. For more information, see the description of the *store*:`SetInfo` method.

slotSymbol — The slot to be returned. This value must be a symbol. Applications should create only a single slot in the store information frame and should store minimal amounts of data in it.

GetDefaultStore

`GetDefaultStore()`

Returns a reference to the store on which new items are created by default. The default store is specified by the user.

GetSignature

store`:GetSignature()`

Returns an integer that is the store's signature. The store signature is a pseudo-random integer assigned by the system when the store object is created.

GetName

store`:GetName()`

Returns the name of the specified store as a string value.

GetSoup

store`:GetSoup(`*soupNameString*`)`

Returns the specified soup object from the specified store. If the soup doesn't exist, this method returns the value `nil`. You can use this method to retrieve a union soup's members one at a time but you cannot use this method to retrieve a union soup object; use the `GetUnionSoupAlways` method for this purpose.

soupNameString The name of the soup to retrieve, as specified by the `name` slot of the soup definition frame used to create the soup.

The following code fragment uses the `GetSoup` method to retrieve the `"mySoup:mySig"` soup from the internal store:

```
local mySoup := GetStores()[0]:GetSoup("mySoup:mySig");
// make sure result is a valid soup
if mySoup:IsValid() then
   // do something
```

GetSoupNames

store:`GetSoupNames()`

Returns an array of strings that are the names of the soups on the specified store.

GetStores

`GetStores()`

Returns an array containing references to all existing stores. Do not modify this array. The elements of this array are store objects to which you can send the messages described in the rest of this section. The element occupying the first position in the array this function returns (`GetStores()[0]`) is always the internal store; however, the meaning of the positions occupied by other stores in this array cannot be relied upon.

HasSoup

store:`HasSoup(`*soupName*`)`

Returns a non-`nil` value if the store specified by *store* contains the soup having the name specified by *soupName*; otherwise, returns `nil`.

soupName	A string that is the name of the soup for which this method tests.

IsReadOnly

store:`IsReadOnly()`

Returns a non-`nil` value if the specified store cannot be written (it could be on a card that is write protected), and returns `nil` if the store can be written.

IsValid

store:`IsValid()`

Returns `true` if the store can be used. A store becomes invalid when it is removed, such as when the storage card on which it resides is removed.

SetDefaultStore

`SetDefaultStore(`*newDefaultStore*`)// platform file function`

Sets the default store as specified and returns a reference to the new default store. Applications should respect the user's default store preferences rather than change them. Do not change any user preferences without obtaining confirmation from the user.

IMPORTANT

This function is not defined in all ROM versions and may be supplied by the NTK Platform file. Call it using this syntax:

`call kSetDefaultStoreFunc with (`*newDefaultStore*`);`

▲

newDefaultStore A reference to the store to be set as default.

SetInfo

store`:SetInfo(`*slotSymbol*`,` *value*`)`

Sets the value of the specified slot in the store's information frame. If the slot does not exist, this function creates it and sets it to the specified value. This method's return value is unspecified.

Applications can create a slot in the information frame to store card data, such as the last time the application encountered a particular card. Because the store information frame is shared by all applications, it is strongly recommended that your application follow the same guidelines for creating its slot in the store information frame as for creating a slot in another application's soup.

IMPORTANT

Values passed to this function must be wrapped in calls to the `EnsureInternal` function to avoid unnecessary appearances of the "Newton need the card…" slip. ▲

slotSymbol The slot to be set (or created if necessary). This value must be a symbol. Applications should create only a

	single slot in the store information frame and should store minimal amounts of data in it. To help avoid name-space collisions with other slots in the store information frame, the name of this slot must be suffixed with your developer signature.
value	The value to be stored in the specified slot.

SetName

store`:SetName(`*storeNameString*`)`

Sets the name of the specified store to the *storeNameString* value and returns the new name of the store. This special-purpose method is intended for use only by backup/restore applications. This method's return value is unspecified.

storeNameString	String that is the store's new name.

TotalSize

store`:TotalSize()`

Returns the total size in bytes of the physical medium on which the specified store resides.

UsedSize

store`:UsedSize()`

Returns the number of bytes used in the store.

Soup Functions and Methods

A **soup** is an opaque object that provides a persistent, dynamic repository for data. Unless removed intentionally, soups remain resident on the Newton device even when the application that owns them is removed. A **union soup** object represents multiple same-named soups as a single entity, regardless of their locations on various physical stores.

The only NewtonScript object you can save in a soup is a frame; however, any slot in a frame can hold any NewtonScript data type and multiple data types can reside in a single frame. The system does not impose any limitations on the number of frames or the kinds of data that may reside in a soup. Frames added to soups must be self-contained; that is, they should not hold references to other data structures.

For more information, see the following sections in *Newton Programmer's Guide*: "Introduction to Data Storage Objects" (page 11-2), "Soups" (page 11-7), and "Entries" (page 11-17).

The functions and methods described in this section allow you to

- obtain a list of soups present on a specified store
- create soups and union soups
- make copies of soups
- write soups and packages to a specified store.
- get information about currently available soups and union soups
- get and set the information frame that describes a soup

RegUnionSoup

`RegUnionSoup(`*appSymbol*`, `*soupDef*`);`

Registers the specified soup definition for use by union soup methods that create soups automatically. This method returns the union soup named by the *soupDef* soup definition or creates a new union soup from that definition, as necessary.

appSymbol	Unique symbol identifying the application to which this soup belongs.
soupDef	A soup definition frame, as specified in "Soup Definition Frame" beginning on page 9-2.

UnRegUnionSoup

`UnRegUnionSoup(`*name*`,` *appSymbol*`);`

Unregisters the specified soup definition with the system. The return value of this method is unspecified; do not rely on this value.

name The name of the soup to unregister.

appSymbol Unique symbol identifying the application to which this soup belongs.

GetUnionSoupAlways

`GetUnionSoupAlways(`*soupNameString*`)`

Returns the union soup object named by the value of the *soupNameString* parameter. This function never returns `nil`; if necessary, it creates a new union soup from the registered soup definition that has *soupNameString* in its `name` slot. For more information, see "Using Newton Data Storage Objects" (page 11-25) in *Newton Programmer's Guide*.

soupNameString The name of the union soup to be retrieved, as specified by the `name` slot in its soup definition frame.

Query

soupOrUSoup`:Query(`*querySpec*`)`

Returns a cursor that iterates over the set of *soupOrUSoup* entries satisfying the *querySpec* query specification.

soupOrUSoup A valid reference to a soup object as returned by the `GetSoup` store method or a union soup object as returned by the `RegUnionSoup` or `GetUnionSoupAlways` global functions.

querySpec A query specification frame, as described in "Query Specification Frame" beginning on page 9-9.

AddToDefaultStoreXmit

uSoup:AddToDefaultStoreXmit(*frame*, *changeSym*)

Adds the specified frame to the specified union soup and transmits a soup change notification message. If necessary, this method creates the member soup to which the frame is added. This method returns the new entry it creates when the frame is added successfully and throws an exception if the frame cannot be added. The frame is added to the appropriate member of the specified union soup according to the user's default store preferences. (The user can specify either the internal store or a store on a storage card as the default store.)

IMPORTANT

The AddToDefaultStoreXmit method modifies the *frame* argument destructively. For more information, see "Adding Entries to Soups" (page 11-35) in *Newton Programmer's Guide*. ▲

frame — The frame to be made into an entry in the specified union soup. This frame must be not be read-only.

changeSym — A unique symbol specifying the application that added the entry; usually this value is the application symbol or some variation on it. Pass nil as the value of this parameter to avoid transmitting a soup change notification.

AddToStoreXmit

uSoup:AddToStoreXmit(*frame*, *store*, *changeSym*)

Adds the specified frame to the member of the specified union soup on the specified store and transmits a soup change notification message. If necessary, this method creates the member soup to which the frame is added.

This method returns the new entry it creates when the frame is added successfully and throws an exception if the frame cannot be added.

frame	The frame to be made into an entry in the specified soup.
store	The store containing the union soup member to which this method adds the specified frame as an entry.
changeSym	A unique symbol specifying the application that added the entry; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

IMPORTANT

The `AddToStoreXmit` method modifies the *frame* argument destructively. For more information, see "Adding Entries to Soups" (page 11-35) in *Newton Programmer's Guide*. ▲

AddFlushedXmit

soupOrUSoup`:AddFlushedXmit(`*frameOrEntry*, *changeSym*`)`

Adds the specified frame or entry to the specified soup, returns the newly added entry, and transmits a soup change notification message. The `AddFlushedXmit` method is similar to the `AddXmit` soup method, except that the `AddFlushedXmit` method does not create a cached entry. This method is intended for use in adding entries that won't be accessed again for awhile (accessing the entry creates the cached entry). For example, you could seed a soup with initial values by calling the `AddFlushedXmit` method from within a loop in your soup's optional `initHook` method.

frameOrEntry	The frame or entry to be added to the specified soup as an entry.
changeSym	A unique symbol specifying the application that added the entry; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

AddToStoreFlushedXmit

uSoup:`AddToStoreFlushedXmit(`*frameOrEntry*`, `*store*`, `*changeSym*`)`

Adds the specified frame or entry to the member of the specified union soup on the specified store, returns the newly added entry, and transmits a soup change notification message. The `AddToStoreFlushedXmit` method is similar to the `AddToStoreXmit` soup method; however, the `AddToStoreFlushedXmit` method does not create a cached entry, nor does it `EnsureInternal` the frame presented as its argument.

This method is intended for use in adding entries that won't be accessed for awhile (accessing the entry creates the cached entry). For example, you could seed a soup on a specified store with initial values by calling the `AddToStoreFlushedXmit` method from within a loop in your soup's optional `initHook` method.

IMPORTANT

The `AddToStoreFlushedXmit` method modifies the *frame* argument destructively. For more information, see "Adding Entries to Soups" (page 11-35) in *Newton Programmer's Guide*. ▲

frameOrEntry	The frame or entry to be added to the specified soup as an entry.
store	The store containing the union soup member to which this method adds the specified frame as an entry.
changeSym	A unique symbol specifying the application that added the entry; usually this value is the application symbol or some variation on it. Pass `nil` as this parameter's value to avoid transmitting a soup change notification.

AddXmit

soup:`AddXmit(`*frame*, *changeSym*`)`

Adds the specified frame to the specified soup, returns the new entry created from this frame, and transmits a change notification.

IMPORTANT

The `AddXmit` method modifies the *frame* argument destructively. For more information, see “Adding Entries to Soups” (page 11-35) in *Newton Programmer’s Guide*. ▲

frame The frame to be made into an entry in the specified soup.

changeSym A unique symbol specifying the application that added the entry; usually this value is the application symbol or some variation on it. Pass `nil` as this parameter’s value to avoid transmitting a soup change notification.

GetMember

uSoup`:GetMember(`*store*`)`

Returns the specified union soup member (single soup) from the specified store, creating that soup if it doesn’t already exist.

IsValid

soup`:IsValid()`

Returns `true` if the soup can be used. A soup object becomes invalid when the store on which it resides is removed, such as when a card is removed, or when the soup itself is deleted.

GetSoupList

uSoup`:GetSoupList()`

Returns an array of soups comprising the specified union soup.

GetSoupDef

`GetSoupDef(`*soupOrUSoupName*`)`

Returns the soup definition frame for the specified soup.

soupOrUSoupName The name of the soup or union soup for which this function retrieves the soup definition.

CopyEntriesXmit

soup`:CopyEntriesXmit(`*destSoup*`,` *changeSym*`)`

Copies the entries in the source soup to the destination soup and transmits a change notification. The copied entries preserve the values of the original entries' unique identifiers. This method's return value is unspecified.

destSoup	The soup in which the copied entries are written. This soup must be empty; this function does not check for duplicate entries in this soup. This soup must not be a union soup; if it is, this method throws a "cant copy to union soup" exception `|evt.ex.fr.store|(-48015)`.
changeSym	A unique symbol identifying the application that copied the entries; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

AddIndexXmit

soupOrUsoup`:AddIndexXmit(`*indexSpec*`,` *changeSym*`)`

Adds an index to the specified soup or union soup and transmits a soup change notification. If this message is sent to a union soup, the index is added to all soups in the union. If the specified soup or union soup resides on a read-only store, this method throws a "store is in ROM" exception `|evt.ex.fr.store|( -48020)`. This method's return value is unspecified.

▲ **WARNING**

Each soup has only one tags index; if you add a tags index to a soup or union soup that already has one, it replaces the original tags index.

You cannot query a union soup on an index not present in all its member soups. Sending the `AddIndexXmit` message to a union soup adds the specified index to all soups currently available to the union; however, any soup introduced to the union subsequently has only its original complement of indexes, which may not include the index this method added. Similarly, any member soup created by the system has only the indexes specified by its soup definition, which may not include the index this method added. ▲

indexSpec An index specification frame. For detailed descriptions of various kinds of index spec frames, see "Data Structures" beginning on page 9-1.

changeSym A unique symbol identifying the application that added the index; usually this value is the application symbol or some variation on it. Pass `nil` as the value of this parameter to avoid transmitting a soup change notification.

GetAllInfo

soup`:GetAllInfo()`

Returns the soup's information frame. Unless an application stores data in this frame, it may not exist in every soup. This special-purpose method is intended for use by backup/restore applications only; most applications need not use it. Applications can use the `GetInfo` method to get their own slot from the soup information frame. For more information, see the description of the `GetInfo` method on page 9-44. See also "Soup Compatibility Information" (page 11-21) in *Newton Programmer's Guide*.

GetIndexes

soup`:GetIndexes()`

Returns an array of index specification frames corresponding to the indexes that exist in the soup.

GetInfo

soup`:GetInfo(`*slotSymbol*`)`

Returns the contents of the specified slot in the soup's information frame; this function returns `nil` if the slot does not exist.

soup	The soup having the information frame to be returned; it must be a single soup, not a union soup. This method is undefined for union soups.
slotSymbol	The slot to be returned. This value must be a symbol.

GetName

soupOrUsoup`:GetName()`

Returns a string that is the name of the soup or union soup object to which this message is sent.

GetNextUid

soup`:GetNextUid()`

Returns the unique identifier to be assigned to the next entry added to the soup. This special-purpose method is intended for use by backup/restore applications. Because the methods that add entries to soups or union soups assign these identifiers automatically, most applications do not need to use the `GetNextUid` method.

GetSignature

soup`:GetSignature()`

Returns an integer that is the signature for the soup. The signature is a random integer that identifies the soup uniquely to the system; it is assigned by the system when the soup is created. You can use this value to determine whether a soup was replaced with another having the same name.

GetSize

soupOrUsoup`:GetSize()`

Returns the size of the specified soup, expressed in bytes.

GetStore

soup`:GetStore()`

Returns a reference to the store on which the specified soup resides.

IsSoupEntry

`IsSoupEntry(`*object*`)`

Returns `true` if the data object passed to this function is a soup entry; otherwise, returns `nil`.

object The object to be tested.

MakeKey

soup`:MakeKey(`*string*`,` *indexPath*`)`

Constructs the index key that would be used for one or more specified values.

You can use this method to determine precisely the index key used for a specified string; under the following conditions, a string may not match its key exactly:

- Keys of type `'string` are truncated after 39 Unicode characters.
- Ink data is stripped from `'string` keys.
- Subkeys in multiple-slot indexes may be truncated or missing when the total key size is greater than 80 bytes.

For code examples, see “Limitations of Index Keys” (page 11-52) in *Newton Programmer’s Guide*.

string — The string for which this method constructs an index key. This string need not exist in the soup to which the `MakeKey` message is sent. When *soup* has a multiple-slot index, the value of this parameter can be an array of strings; otherwise, this value must be a single string. Missing elements are presumed to be `nil` values. When the value of this parameter is an array, each of its elements must hold the data type specified by the corresponding element of the *indexPath* array.

indexPath — The index path associated with the key value specified by the value of the *string* parameter. This value must represent a valid index path in the soup to which the `MakeKey` message is sent. When *soup* has a multiple-slot index, the value of the *indexPath* parameter can be an array of index paths corresponding to the elements of the array passed as the value of the *string* parameter; otherwise, the value of the *indexPath* parameter must be a single index path. When making a key for use with a multiple-slot index, the *indexPath* parameter must specify all the slots indexed by a particular multiple-slot index in the same order as used to generate the index. If the value of this parameter is missing any of the paths indexed by a multiple-slot index on the soup, or any of the paths do not appear in the same order as in the index spec used to generate the multiple-slot index, this method throws the “Soup index does not exist”`|evt.ex.fr.store|(-48013)` exception.

RemoveAllEntriesXmit

soup`:RemoveAllEntriesXmit(`*changeSym*`)`

Deletes all entries from the specified soup and transmits a change notification. The soup object to which this message is sent must be a single soup; this method is not implemented for union soups. This method's return value is unspecified.

changeSym A unique symbol identifying the application that removed the entries; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

RemoveFromStoreXmit

soup`:RemoveFromStoreXmit(`*changeSym*`)`

Removes the specified soup from its store, deletes all of its entries, and sends a soup change notification. This method cannot be used on a union soup. This method's return value is unspecified.

changeSym A unique symbol identifying the application that removed the soup; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

RemoveIndexXmit

soupOrUsoup`:RemoveIndexXmit(`*indexPath*`, `*changeSym*`)`

Removes an index from the specified soup or union soup object and transmits a soup change notification. This method's return value is unspecified.

▲ **WARNING**
You cannot query a union soup on an index that is not present in all of its member soups. Sending the `RemoveIndexXmit` message to a union soup removes the specified index from all soups currently in the union. However, any soup introduced subsequently to the union has its original complement of indexes, which may include the one this method removed. Similarly, any member soup created subsequently by the system is created with the indexes specified in its soup definition, which may include the index this method removed from other members. ▲

indexPath The path expression on which the index to remove was generated; that is, the same index path used to create the index.

changeSym A unique symbol identifying the application that removed the index; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

AddWithUniqueIDXmit

soup`:AddWithUniqueIDXmit(`*entry*`,` *changeSym*`)`

Adds the *entry* frame to the specified soup as a soup entry having the unique identifier specified in the *entry* frame, returns the newly added entry, and transmits a soup change notification. This method throws an exception if the specified unique identifier is already used by an entry in the destination soup.

This special-purpose function is intended only for restoration of soup data; most applications should not use it. Normally, applications use the *soup*`:AddXmit` method to add a frame to a specified soup. The *soup*`:AddXmit` method generates a new unique identifier for the entry it adds.

entry — The entry to be added to the specified soup. This value must be a soup entry rather than a normal frame.

changeSym — A unique symbol identifying the application that added the entry; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

SetAllInfoXmit

soup`:SetAllInfoXmit(`*frame*`, `*changeSym*`)`

Writes the specified frame as the soup's information frame and transmits a soup change notification. This method's return value is unspecified.

This special-purpose method is intended for use by backup/restore applications only; most applications need not use it. Instead, applications should use the *soup*`:SetInfoXmit` method to store data in a single slot in the soup information frame. For more information, see the description of the `SetInfoXmit` method (page 9-50).

▲ **WARNING**

The soup information frame holds the soup definition frame used to create the soup. Loss of the soup definition frame can lead to the presence of a null union soup. For more information, see "Null Union Soups" (page 11-23) in *Newton Programmer's Guide*. ▲

frame — The frame to be written as the soup's information frame.

changeSym — A unique symbol identifying the application that changed the soup information frame; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

SetInfoXmit

soup`:SetInfoXmit(`*slotSymbol*`, `*value*`, `*changeSym*`)`

Sets the value of the specified slot in the soup information frame and transmits a soup change notification. If the slot does not exist, this function creates it and sets it to the specified value. This method's return value is unspecified.

slotSymbol — The slot to be set (or created if necessary). This value must be a symbol. Applications should create only a single slot in the soup information frame and should store minimal amounts of data in this slot. To avoid name-space collisions with other slots in the soup information frame, it is strongly recommended that you incorporate your unique developer signature in this name.

For more information, see "Soup Information Frame" (page 11-22) and "Making Changes to Other Applications' Soups" (page 11-37) in *Newton Programmer's Guide*.

value — The value to be stored in the specified slot.

changeSym — A unique symbol identifying the application that changed the soup information frame; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

SetName

soup`:SetName(`*soupNameString*`)`

Sets the name of the soup to the *soupNameString* string. This method's return value is unspecified. If you try to set the name to an invalid value (for example, one already in use) this method throws an exception. Generally, you should avoid changing the names of soups (even your own), because other applications may be using them.

soupNameString — The string that is the soup's new name.

▲ **WARNING**
Do not under any circumstances change the names of the built-in soups. ▲

CreateSoupFromSoupDef

`CreateSoupFromSoupDef(`*soupDef*, *store*, *changeSym*`)`

Creates a single soup on the specified store using the specified soup definition, transmits a `'soupChanged` notification and returns a reference to the new soup. Normally, plain soups like the one returned by this function are created by methods that add entries to union soups.

soupDef	The soup definition used to create the new soup.
store	The store on which to create the new soup.
changeSym	A unique symbol identifying the application that created the new soup; usually this value is the application symbol or some variation on it. Pass `nil` as the value of this parameter to avoid transmitting a soup change notification.

SupplantSoupDef

`SupplantSoupDef(`*soup*, *soupDef*`)// platform file function`

Installs the specified soup definition in the specified single soup. This method's return value is unspecified.

▲ **WARNING**
Changing a soup definition frame is not recommended. Use this function only if you know that what you are attempting to do will not cause errors or undesirable side effects. ▲

IMPORTANT

This function is not defined in all ROM versions and may be supplied by the NTK Platform file. Call it using this syntax:

`call kSupplantSoupDefFunc with (`*soup*`, `*soupDef*`);`

▲

soup The soup on which this method operates. This object must be a soup, not a union soup.

soupDef The soup definition frame this method installs.

The `SupplantSoupDef` function works on single soups only, not on union soups. You can use the union soup method `GetSoupList` to retrieve a list of the member soups currently available to a specified union soup.

You can use the `SupplantSoupDef` function to

- Change the user-visible information for a specified soup. For example, you could use this function to change the string that the Extras Drawer displays as the soup's name.
- Add a soup definition frame to a soup that lacks one. For example, soups created by system software prior to version 2.0 do not have soup definition frames.
- Replace the soup definition frame in a soup that already has one. Note that this may cause inconsistencies with other soups in the union that can lead to unstable behavior.

Note

This function does not change the soup definition currently registered with the system—it changes only the local copy of the definition held by a soup created from that definition. To change a soup definition registered with the system, you must replace it completely. To do so, first call the `UnRegUnionSoup` function to unregister the current soup definition, and then call the `RegUnionSoup` function to register the new soup definition. ◆

Because most of the information in a soup definition frame is used only when the system creates a new soup, the appropriate usage of the `SupplantSoupDef` function is limited. For example, although you can use this method to change the indexes a soup definition specifies for new soups, the actual indexes in existing soups are not updated by this method. Soups created subsequently from this definition may not have the same complement of indexes as other soups in their union, which may cause operations on the union soup to fail. Exercise extreme caution when using this method for any purpose.

The following code fragment provides an example of the proper use of this function. Note that because this function is supplied by the Newton 2.0 platform file, it must be called using the `call k`*FnName*`Func with ()` syntax shown in the example:

```
// unregister old definition
UnRegUnionSoup("mySoup:mySig",'|MyApp:MySig|);
// register new version of soup definition
// assume myNewSoupDef is valid
local uSoup := RegUnionSoup('|MyApp:MySig|, myNewSoupDef);
// update existing soups
foreach member in uSoup:GetSoupList() do
   begin
      call kSupplantSoupDefFunc with (member, myNewSoupDef);
   // perform other housekeeping such as adding or removing indexes
   end;
```

GetIndexesModTime

soup`:GetIndexesModTime()`

Returns the time when the soup indexes were last changed, expressed in the system's internal time format as returned by the `Time` function. Soup index information is set when the soup is created or restored; when indexes are added or removed; and when indexed soup entries are added, deleted, or changed.

GetInfoModTime

soup`:GetInfoModTime()`

Returns the time when the soup info was last changed. Values in the soup information frame are set when the soup is created or restored. These values may also be changed by the `SetInfoXmit` and `SetAllInfoXmit` soup methods.

Soup Change Notification Functions

These functions allow you to register and unregister callback functions that the system executes when a specified soup changes in some way; for example, when soup entries are added or removed, when the soup itself is created or removed, and so on.

RegSoupChange

`RegSoupChange(`*soupName*`,` *callbackID*`,` *callBackFn*`)`

Registers a callback function to be executed whenever the specified soup changes. This function's return value is unspecified.

soupName A string that is the name of the soup that changed.

callbackID A unique symbol identifying the *callBackFn* function to the soup change mechanism. Because this symbol must be unique among the symbols registered with this soup, this value normally includes your application's application symbol or some variation on it.

▲ **WARNING**
The *callBackFn* function must not call the `RegSoupChange` or `UnRegSoupChange` functions. ▲

callBackFn A function executed when the specified soup changes. The current system ignores the value this function returns; however, it is recommended that this function return the value `nil`. This function must not call either of the `RegSoupChange` or `UnRegSoupChange` functions. For a detailed description of this function, see "Callback Functions for Soup Change Notification" beginning on page 9-14.

UnRegSoupChange

`UnRegSoupChange(`*soupName*`,` *callbackID*`)`

Unregisters the specified callback function with the soup change notification service for the specified soup only. This function's return value is unspecified.

soupName — A string that is the name of the soup that changed.

callbackID — A unique symbol identifying the *callBackFn* function to the soup change mechanism. Because this symbol must be unique among the symbols registered with this soup, this value normally includes your application's application symbol or some variation on it.

XmitSoupChange

`XmitSoupChange(`*soupName*`,` *appSymbol*`,` *changeType*`,` *changeData*`)`

Notifies applications registered with the soup change mechanism that the specified soup has changed. Use this function when you don't want to transmit separate notifications for every change to a soup, or to send change notifications on older Newton devices.

soupName — A string that is the name of the soup that changed.

appSymbol — Unique symbol identifying the application that caused the change.

changeType — A symbol indicating the kind of change that occurred; this value must be one of the symbols listed in Table 9-1 (page 9-15).

changeData — The data that changed. The data passed as this argument varies according to the value of the *changeType* parameter; see Table 9-1 (page 9-15) for more information.

Store Part Functions

A **store part** is an object that encapsulates a read-only store. Because you can build store parts into application packages, a store part is sometimes referred to as a **package store.** For more information, see "Parts" (page 12-3) in *Newton Programmer's Guide.*

This section describes functions that can be used to work with store parts.

GetPackageStore

`GetPackageStore(`*name*`)`

Returns the package store having the specified name; otherwise, returns `nil`. As always in NewtonScript, string comparison is not case sensitive. When more than one currently available store has the specified name, this function's behavior is unspecified.

name String that is the name of the package store to retrieve.

GetPackageStores

`GetPackageStores()`

Returns an array of all available package stores.

▲ **WARNING**
Do not modify the array this function returns. ▲

Methods for Manipulating Tags

A **tag** is an optional developer-defined symbol used to mark one or more soup entries. Each soup can contain a maximum of 624 tags. The system treats missing tags as `nil` values.

Tags reside in a developer-specified slot that can be indexed, with the results stored in a special index called the **tags index.** The tags index is used to select soup entries according to their associated symbolic values without reading the entries themselves into memory; for example, one could select the subset of entries tagged `'business` from the `ROM_CardfileSoupName` soup. Note that the system allows only one tags index per soup.

For more information, see “Indexes” (page 11-8) in *Newton Programmer’s Guide*.

The methods described here allow you to add, remove, and modify tags in soups and union soups as well as get information on the currently existing tags in a specified soup. Methods that modify soups can transmit change notifications automatically.

AddTagsXmit

soupOrUsoup`:AddTagsXmit(`*tags*`, `*changeSym*`)`

Adds the specified tags to the soup’s tags index as necessary and transmits a soup change notification. This method requires that the soup already have a tags index. If this message is sent to a union soup, the tags are added to each soup in the union. Note that the soup entries themselves are not changed by this method. This method’s return value is unspecified.

Normally you do not need to add tags to a soup explicitly; when you add an entry that uses new tags, the system adds them to the tags index automatically. You should use the `AddTagsXmit` method only when unused tags must be added to the tags index for some reason. For example, if you wanted to allow the user to file items in a folder category that was not yet used, you could use the `AddTagsXmit` method to add the unused tag to the tags index. Subsequently, you could use the `GetTags` method to retrieve all the currently available tags (including unused tags) for display to the user.

This method throws the “no tags” exception `|evt.ex.fr.store|(-48027)` when the soup has no tags index. When executing this method causes the maximum number of tags for the specified soup to be exceeded, this method throws the “invalid tags count” exception `|evt.ex.fr.store|(-48026)` and does not add any of the new tags.

Note
Most applications do not need to use this method. When an entry with one or more new tags is added to the soup, the new tags are added to the tags index automatically. ◆

tagsToAdd An array of symbols or a single symbol.

changeSym A unique symbol identifying the application that added the tag(s); usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

GetTags

soupOrUSoup`:GetTags()`

Returns an array containing the specified soup's tags. Returns `nil` if the soup does not have a tags index. If the specified soup is a union soup, the array returned by this method contains the tags for all soups in the union.

soupOrUSoup The soup or union soup from which this method retrieves tags.

HasTags

soupOrUSoup`:HasTags()`

Returns `true` if the specified soup has a tags index. If the specified soup is a union soup, this method returns `true` only if each of the union's member soups has a tags index.

soupOrUSoup The soup or union soup to be tested.

ModifyTagXmit

soupOrUsoup`:ModifyTagXmit(`*oldTag*`, `*newTag*`, `*changeSym*`)`

Changes the symbol specified by *oldTag* to that specified by *newTag*, updates the soup entries, and transmits a soup change notification. If this message is sent to a union soup, the specified tag is modified in all soups in the union. This method returns the value `nil` if successful. This method returns `nil` and does nothing if *oldTag* is not one of the tags in the specified soup.

Note
If the only difference between *oldTag* and *newTag* is case, this method does nothing because symbolic values are case insensitive. For example, changing a tag from `'foo` to `'Foo` has no effect. ◆

This method throws the "no tags" `|evt.ex.fr.store|(-48027)` exception when the *soupOrUsoup* soup has no tags index. If the *newTag* tag is already present in the *soupOrUsoup* soup's tags index, this method throws an "invalid tag spec" exception `|evt.ex.fr.store|( -48028`).

soupOrUSoup — The soup or union soup for which this method modifies the specified tag.

oldTag — A symbol specifying an existing soup tag.

newTag — The new symbol for the tag specified by the *oldTag* argument.

changeSym — A unique symbol identifying the application that invoked this method; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

RemoveTagsXmit

soupOrUsoup`:RemoveTagsXmit(`*tagsToRemove*`,` *changeSym*`)`

Removes the specified tags as necessary from the specified soup, updates the soup entries, and transmits a soup change notification. If this message is sent to a union soup, the specified tags are removed from all soups in the union. This method's return value is unspecified.

This method throws the "no tags" `|evt.ex.fr.store|(-48027)` exception when the soup has no tags index.

soupOrUSoup — The soup or union soup from which this method removes the specified tags.

tagsToRemove — An array of symbols or a single symbol.

changeSym — A unique symbol identifying the application that removed the tag(s); usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

Query and Cursor Methods

This section describes the `Query` method of soups and union soups. This method retrieves soup data according to criteria specified by a query specification frame or **query spec** passed as its argument. This method returns a **cursor**, which is an object that iterates over the set of soup entries meeting the criteria defined by the query spec. A **soup entry** is a frame that has been saved in a soup. For more information, see the following sections in *Newton Programmer's Guide*: "Introduction to Data Storage Objects" (page 11-2), "Queries" (page 11-10), and "Cursors" (page 11-16).

In addition to describing the `Query` method of soups and union soups, this section describes methods that manipulate the cursor to obtain individual soup entries.

Clone

cursor`:Clone()`

This method makes a copy of the specified cursor and returns the copy.

Note
Do not use the global functions `Clone` or `DeepClone` to clone cursors. Instead, use the `Clone` method for cursors, as described here. ◆

CountEntries

cursor`:CountEntries()`

Returns the number of entries matching the query specification that generated the *cursor* cursor. If the query spec used to generate the cursor specifies endrange values (includes any of the `beginKey`, `beginExclKey`, `endKey`, or `endExclKey` slots), this method counts only the entries within the range over which the cursor iterates.

cursor Soup cursor returned by the `Query` function.

Note
Use this method only when necessary—counting a large number of entries may be time-consuming and may require relatively large amounts of heap space. ◆

Entry

cursor`:Entry()`

Returns the current soup entry referenced by *cursor*.

If the current entry is deleted from the soup, the entry reverts to a plain frame (rather than a soup entry), and the method *cursor*`:Entry` returns the symbol `'deleted`. Make sure your code can handle gracefully a return value of `'deleted` from the *cursor*`:Entry` method.

If the cursor is advanced past the last entry or moved before the first entry in the set, the current entry pointed to by the cursor has the value `nil`. Make sure your code can also handle gracefully a `nil` value returned from the *cursor*`:Entry` method.

If the current entry is altered in a way that causes it to move to a different index position, the cursor moves with it.

EntryKey

cursor`:EntryKey()`

Returns the current entry key without reading the entry into memory.

Note
The value this method returns may be different from the actual index key value for a particular entry; for more information, see the description of the `indexValidTest` function in “Query Specification Frame” beginning on page 9-9. ◆

GoTo

cursor`:GoTo(`*entry*`)`

If the specified entry is valid, this method moves the cursor to the specified entry and returns `true`. If the specified entry is not valid, the cursor does not move and this method throws an exception.

entry — The entry to which this method moves the cursor. You cannot create an entry procedurally by creating a frame having certain slots and values. The only valid entries are those returned by the various cursor and entry methods.

GoToKey

cursor`:GoToKey(`*key*`)`

Moves the cursor to the first valid entry having the specified key value, or to the next entry in index order if no entry has the specified key value, and returns the entry. If no entries have the specified key value, or the specified key value is invalid, the cursor tests each entry until it runs out of entries, at which point this method returns `nil`.

key — For soups indexed on a single slot, a single index key value; for soups having a multiple-slot index, an array of these values. The data type must be that specified by the soup index used to generate the *cursor* object that received the `GoToKey` message.

MapCursor

`MapCursor(`*cursor*`, `*function*`)`

Applies the specified function to each of the cursor's entries in turn and returns an array of the results. If *function* is `nil`, the returned array consists of the entries themselves.

cursor — The cursor supplying the entries against which this method executes the specified function.

function — The function that is to be mapped to the cursor's entries. This function must accept a single entry as its argument.

If this function returns a `nil` result for an entry, that entry is not added to the return array. `nil` results are discarded.

Move

cursor`:Move(`*n*`)`

Moves the cursor *n* entries forward from its current position and returns that entry. If *n* is negative, the cursor is moved backwards. If the cursor is advanced past the last entry or moved before the first entry in the set of entries it references, this method returns the value `nil`.

n — Number of positions (entries) to move the cursor.

Next

cursor`:Next()`

Moves the cursor to the next entry in the query result and returns the entry. If the cursor is advanced past the last entry in the set of entries it references, this method returns the value `nil`.

Prev

cursor:`Prev()`

Moves the cursor to the previous entry in the set of entries referenced by the cursor and returns the entry. If the cursor is moved before the first entry in the set of entries it references, this method returns the value `nil`.

Reset

cursor:`Reset()`

Resets the cursor to the entry at the beginning of the range over which it iterates.

ResetToEnd

cursor:`ResetToEnd()`

Resets the cursor to the entry at the end of the range over which it iterates.

Status

cursor:`Status()`

Returns a symbol describing the validity of the cursor. Cursors on union soups become invalid when a soup missing an index common to the rest of the union is included in the union. For more information, see "Testing Validity of the Cursor" (page 11-54) in *Newton Programmer's Guide*.

This method returns the following symbols:

`'valid`	No problems with the soups or indexes used by this cursor.
`'missingIndex`	At least one soup referenced by this cursor is missing one or more indexes common to the other soups in the union. The missing index may have been specified in the `indexPath` or `tagsSpec` slot of the query spec used to generate the cursor.

WhichEnd

cursor`:WhichEnd()`

Returns `'begin` or `'end` when the cursor's position is outside the range of valid entries. When the cursor is within the valid range of entries, this function's return value is `nil`.

Entry Functions

An **entry** is a frame added to a soup by any of several soup or union soup methods provided for this purpose. A valid entry can be obtained only as the result of a cursor method or a method that adds a frame to a soup or union soup. You cannot create a valid entry by adding certain slots and values to a frame—the system must create the entry for you from the frame presented to an entry creation method such as the `AddToDefaultStoreXmit` union soup method. For more information, see the following sections in *Newton Programmer's Guide*: "Introduction to Data Storage Objects" (page 11-2) and "Entries" (page 11-17).

This section describes functions used to work with individual soup entries.

EntryChangeXmit

`EntryChangeXmit(`*entry*`,` *changeSym*`)`

Writes a cached *entry* back to its soup and transmits a change notification. Returns an error if *entry* is not a valid soup entry; otherwise, this function's return value is unspecified.

entry	The cached entry this method writes back to its soup.
changeSym	A unique symbol identifying the application that changed the entry; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

EntryUndoChanges

`EntryUndoChanges(`*entry*`)`

Disposes of the cached *entry* frame. Any changes made to the cached entry are lost and the entry reverts to the version stored in the soup. This function's return value is unspecified.

entry — The soup entry. If this entry contains VBO data, this function undoes its changes also.

EntryFlushXmit

`EntryFlushXmit(`*entry*`, `*changeSym*`)`

Writes the entry cache back to the specified soup entry and transmits a change notification. This function's return value is unspecified.

This function is intended for use in changing entries that won't be accessed for awhile (accessing the entry creates the cached entry). Use of this function can result in dramatic savings of time and heap space when writing a large frame or many smaller frames to a soup. For example, you might call this function from within a loop that changes a slot in every entry in a soup.

The `EntryFlushXmit` function is similar to the `EntryChangeXmit` function; however, the `EntryFlushXmit` function clears the entry cache instead of updating it.

entry — The entry from which the cached frame was originally extracted.

changeSym — A unique symbol identifying the application that changed the entry; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

EntryIsResident

`EntryIsResident(`*entry*`)`

Returns `true` if the specified entry is cached; otherwise, returns `nil`. For more information about the entry cache, see "Entries" (page 11-17) in *Newton Programmer's Guide*.

entry	The entry to be tested.

EntryCopyXmit

`EntryCopyXmit(`*entry*`,` *newSoup*`,` *changeSym*`)`

Copies the specified entry into the specified soup, returns the copy of *entry*, and transmits a change notification.

Note
This function copies the cached entry frame—not the original soup entry—into the new soup. ◆

entry	The entry to be copied.
newSoup	The soup into which the specified entry is to be copied.
changeSym	A unique symbol identifying the application that copied the entry; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

EntryMoveXmit

`EntryMoveXmit(`*entry*`,` *newSoup*`,` *changeSym*`)`

Moves the specified entry into the specified soup and transmits a soup change notification message. This function copies the cached entry into the new soup, verifies the integrity of the duplicate entry, and deletes the original soup entry. This function's return value is unspecified.

entry	The soup entry to be moved.
newSoup	The soup into which the specified entry is to be moved.

changeSym A unique symbol identifying the application that moved the entry; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

EntryReplaceXmit

`EntryReplaceXmit(`*original*`,`*replacement*`,`*changeSym*`)`

Replaces the contents of the *original* soup entry with the *replacement* entry and transmits a soup change notification. This function's return value is unspecified.

original The soup entry to be replaced. This value must be a soup entry, not a normal frame.

replacement The soup entry to be added. This value can be an entry or a normal frame. In the latter case, this function makes the frame into a soup entry and adds the new entry to the soup.

changeSym A unique symbol identifying the application that replaced the entry; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

EntryRemoveFromSoupXmit

`EntryRemoveFromSoupXmit(`*entry*`, `*changeSym*`)`

Removes *entry* from its soup and transmits a soup change notification. The entry frame is converted to a plain frame (unmarked as belonging to a soup). The return value of this function is unspecified.

entry The soup entry to be removed and converted to a plain frame.

changeSym A unique symbol identifying the application that removed the entry; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

EntrySize

`EntrySize(`*entry*`)`

Returns the number of bytes that *entry* occupies on the store. Note that entries are compressed when resident on a store, and decompressed automatically when they are read into the NewtonScript heap.

entry The soup entry on which this function operates.

EntrySoup

`EntrySoup(`*entry*`)`

Returns a reference to the soup in which *entry* resides.

entry The soup entry on which this function operates.

EntryStore

`EntryStore(`*entry*`)`

Returns a reference to the store on which *entry* resides.

entry The soup entry on which this function operates.

EntryTextSize

`EntryTextSize(`*entry*`)`

Returns the number of bytes of *entry* that are occupied by text.

entry The soup entry on which this function operates.

FrameDirty

`FrameDirty(`*frameOrEntry*`)`

Returns `true` if the specified frame in memory has been modified since it was retrieved from its soup; otherwise, returns `nil`. Although this function detects changes to nested frames, it does not discern changes to bytes within binary objects. Because strings are implemented as binary objects, this function does not detect changes to individual characters in a string.

frameOrEntry The frame or soup entry to be tested.

The `FrameDirty` function may not detect changes caused by editing string data in `clParagraphView` views because these views manipulate characters within strings as much as possible in lieu of creating new strings. The following code fragment demonstrates this problem in the NTK Inspector:

```
s := GetStores()[0]:CreateSoup("Test:NewtonDTS", []);
e := s:Add({slot: 'value, string: "A test entry",
nested: {slot: 'notherValue}})
#4410B69  {slot: value,
           String: "A test entry",
           nested: {slot: notherValue},
           _uniqueID: 0}
// the unmodified entry tests clean
FrameDirty(e)
#2        NIL
// Modify the string without changing its reference
e.string[0] := $a;
// FrameDirty doesn't detect in-place changes to binaries
FrameDirty(e)
#2        NIL

// writing the cached entry marks it as unchanged
EntryChange(e);
// change the string reference
e.string := "A new string";
// FrameDirty detects this kind of change successfully
FrameDirty(e)
#1A       TRUE
```

```
// FrameDirty also detects nested changes successfully
EntryChange(e);
e.nested.slot := 'newValue;
FrameDirty(e)
#1A       TRUE

// cleanup
s:RemoveFromStore()
```

EntryModTime

EntryModTime(*entry*)

Returns the time when the specified entry was last modified. The time is expressed as an integer that is the number of minutes passed since midnight, January 1, 1904. This function gets this information directly from the soup, which is faster than referencing the entry; the latter approach would require that the entire entry frame be constructed.

entry The soup entry on which this function operates.

EntryChangeWithModTimeXmit

EntryChangeWithModTimeXmit(*entry*, *changeSym*)

Writes a cached *entry* back to its soup using the modification time you specify, and transmits a soup change notification. This function's return value is unspecified. This special-purpose function is intended for use by backup/restore applications only; most applications need not use it.

entry The cached entry this method writes back to its soup.

changeSym A unique symbol identifying the application that changed the entry; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

EntryReplaceWithModTimeXmit

`EntryReplaceWithModTimeXmit(`*original*`,`*replacement*`,`*changeSym*`)`

Replaces the *original* entry with the *replacement* entry, sets the modification time of the *replacement* entry to match that of the *original* entry, and transmits a soup change notification. This function's return value is unspecified.

This special-purpose method is intended for use by backup/restore applications only; most applications need not use it.

original	The soup entry to be replaced. This value must be an entry, not a normal frame.
replacement	The soup entry to be added. This value can be an entry or a normal frame. In the latter case, this function makes the frame into a soup entry and adds the new entry to the soup.
changeSym	A unique symbol identifying the application that replaced the entry; usually this value is the application symbol or some variation on it. Pass `nil` for the value of this parameter to avoid transmitting a soup change notification.

EntryUniqueID

`EntryUniqueID(`*entry*`)`

Returns the value that identifies the specified entry to the system. This function gets this information without reading the entry into the cache.

Entry Alias Functions

An **entry alias** is an object that provides a standard way to save a reference to a soup entry. A soup entry cannot save a reference to an entry that resides in another soup, but entry aliases themselves may be stored in soups.

The functions described here allow you to work with entry aliases.

MakeEntryAlias

`MakeEntryAlias(`*entry*`)`

Returns an entry alias object representing the specified soup entry. This object can be saved in a soup and later used as input to the `ResolveEntryAlias` function to retrieve the soup entry.

entry The soup entry to which this method creates a reference.

ResolveEntryAlias

`ResolveEntryAlias(`*alias*`)`

Returns the soup entry referenced by the specified alias. Returns `nil` if the entry cannot be retrieved—typically because the original store, the original soup, or the original entry is not found.

alias The alias for which this method retrieves the corresponding soup entry.

IsEntryAlias

`IsEntryAlias(`*object*`)`

Returns true if the specified object is an entry alias.

object The object to be tested.

IsSameEntry

`IsSameEntry(`*entryOralias1*`, `*entryOralias2*`)`

This method returns the value `true` only if its arguments evaluate to the same soup entry. Passing two distinct entries with identical content to this function does not cause it to return the value `true`. This method can compare soup entries, entry aliases, or combinations of the two.

entryOralias1 The soup entry or entry alias to be compared to the value of the *entryOralias2* parameter.

entryOralias2 The soup entry or entry alias to be compared to the value of the *entryOralias1* parameter.

VBO Functions and Methods

A **virtual binary object** or **VBO** is a special kind of object used to hold binary data larger than the available space in the NewtonScript heap. For more information, see "Virtual Binary Objects" (page 12-2) in *Newton Programmer's Guide*.

In addition to the functions described in this section, VBOs support all standard object system functions such as `ClassOf`, `SetClass`, `Length`, `SetLength`, `Clone`, `BinaryMunger`, and so on. VBO data is not persistent until the VBO is put in a soup entry and the entry is written to a soup.

IMPORTANT
Store memory for VBO data is not allocated until the VBO is written to a soup. It is strongly recommended that you enclose in a `try` block any code that writes VBO data. For more information, see "Using Virtual Binary Objects" (page 12-8) in *Newton Programmer's Guide*. ▲

NewVBO

store`:NewVBO(`*class*`, ` *size*`)`

Creates on the specified store a virtual binary object of the specified class large enough to store the specified number of bytes. This function returns a reference to the object it creates.

IMPORTANT
Store memory for VBO data is not allocated when the VBO is created—it is allocated when the VBO is written to a soup. For more information, see "Using Virtual Binary Objects" (page 12-8) in *Newton Programmer's Guide*. ▲

class	A symbol specifying the class of the virtual binary object this method creates.
size	The initial size of the VBO, expressed in bytes.

NewCompressedVBO

store:`NewCompressedVBO(`*class*`,` *size*`,` *companderName*`,` *companderArgs*`)`

Creates on the specified store a virtual binary object large enough to store the specified number of bytes. This function returns a reference to the object it creates. Normally, the object returned by this function compresses and decompresses its associated binary data on demand; however, this method creates an object that saves binary data in uncompressed form when `nil` is specified as the value of the *companderName* parameter.

A compander (compressor-expander) is an object that transparently compresses data as it is stored and expands data as it is read. The compander specified by the value of the *companderName* parameter is instantiated using the values specified by the *companderArgs* parameter. Because both companders provided by the current system initialize themselves automatically, you must always pass `nil` as the value of the *companderArgs* parameter.

IMPORTANT
Store memory for VBO data is not allocated when the VBO is created—it is allocated when the VBO is written to a soup. For more information, see “Using Virtual Binary Objects” (page 12-8) in *Newton Programmer’s Guide*. ▲

class — A symbol specifying the class of the binary object that this method creates.

size — The initial size of the VBO, expressed in bytes.

companderName — A string value specifying the implementation of the store compander protocol used when the VBO created by this object is written to or read from a soup entry. If the value of this parameter is `nil`, an uncompressed object is created. The following strings are valid values for this parameter:

`"TLZStoreCompander"`
: Specifies the use of the Lempel-Ziv compressor-expander.

`"TPixelMapCompander"`
: Specifies the use of a compander specialized for pixel map data. (A bitmap is a pixel map having a bit depth of 1.) This compander assumes that the data in the VBO is a pixel map and that the pixel map data is 32-bit aligned; that is, the length of the rows in the pixel map is an even multiple of 4 bytes.

  For a description of the Newton bitmap format, see "MakeBitmap" (page 10-19).

companderArgs Arguments for instantiating the specified compander. In the current implementation, always pass `nil` as the value of this parameter.

IsVBO

`IsVBO(`*vbo*`)`

Returns a non-`nil` value if the object to be tested is a virtual binary object; otherwise, returns `nil`.

vbo The object to be tested.

GetVBOStore

`GetVBOStore(`*vbo*`)`

Returns the store object on which the specified virtual binary object resides. This function returns `nil` if its argument is not a VBO.

vbo The virtual binary object to be tested.

GetVBOStoredSize

`GetVBOStoredSize(`*vbo*`)`

Returns the number of bytes the specified VBO actually uses in the store; for example, if the VBO is compressed, this function returns its compressed size.

vbo The VBO to be tested. Do not use objects other than VBOs as the value of this parameter.

GetVBOCompander

`GetVBOCompander(`*vbo*`)`

Return the name of the compander used for the specified object. If the object is not a VBO, this function returns an unspecified value.

vbo The VBO to be tested.

Mock Entry Functions

A **mock entry** is a NewtonScript object that mimics the behavior of a soup entry. The mock entry is a foundation object you can use to build up a suite of objects that acts like the system-supplied store, soup, cursor, and entry objects. For example, you could create a mock entry object that uses a serial communications link to retrieve a record from a remote database; additional objects could implement methods to provide cursor-like access to these mock entries, just as if they resided in a local soup.

The current implementation of the Newton object system provides only mock entries; you must implement appropriate mock cursors, mock soups, and mock stores as required.

For more information, see "Mock Entries" (page 12-4) in *Newton Programmer's Guide*.

The global functions described here create and manipulate mock entries. They do not work on normal soup entries.

See also Chapter 23, "Utility Functions Reference," for a description of the `NewWeakArray` function.

NewMockEntry

`NewMockEntry(`*handler*`,` *cachedFrame*`)`

Creates a new mock entry having the specified handler and cached frame.

handler The frame implementing this mock entry's methods.

cachedFrame The frame containing the mock entry's data. You can pass `nil` for this value and fill in the entry data later from the `EntryAccess` method of this mock entry's handler frame.

IsMockEntry

`IsMockEntry(`*object*`)`

Returns a non-`nil` value if the specified object is a mock entry; otherwise, returns `nil`. This function returns the value `nil` when the object to be tested is a normal soup entry; in contrast, the `IsSoupEntry` function returns `true` for mock entries and for normal soup entries.

object The object to be tested.

EntrySetCachedObject

`EntrySetCachedObject(`*mockEntry*`,` *newCachedFrame*`)`

Installs the specified cached frame in the specified mock entry. The cached frame is the frame that holds the mock entry's data— the system forwards accesses of the specified mock entry to this frame transparently.

mockEntry The mock entry object for which the *newCachedFrame* frame is to be entry data. If the value of this parameter is not a mock entry (as created by the `NewMockEntry` function), an error is signalled.

newCachedFrame The frame to be installed as the entry data for the specified mock entry.

EntryCachedObject

`EntryCachedObject(`*mockEntry*`)`

Returns the specified mock entry's cached frame.

EntrySetHandler

`EntrySetHandler(`*mockEntry*`, `*newHandler*`)`

Installs the specified frame as the handler for the specified mock entry.

mockEntry	The mock entry in which the *newHandler* frame is installed.
newHandler	The handler frame to install in the *mockEntry* object.

EntryHandler

`EntryHandler(`*mockEntry*`)`

Returns the specified mock entry's handler frame. This special-purpose method is intended for debugging purposes only.

mockEntry	The mock entry object to be tested.

Developer-Defined Entry Handler Methods

You must implement the methods described here in order to use mock entries.

EntryAccess

handler`:EntryAccess(`*mockEntry*`)`

You supply this method, which is called when the frame system needs to access a slot in a mock entry and the mock entry's cached frame is not present. This method must create a frame representing the entry and use the `EntrySetCachedObject` function to assign that frame to the *mockEntry* object.

handler	The handler frame for the specified mock entry.

mockEntry The mock entry being accessed. Do not rely on this value—it is not always passed.

Optional Developer-Defined Entry Handler Methods

Your mock entry handler should also implement the following methods as necessary. These methods are the mock entry counterparts to system-supplied entry functions.

handler`:EntrySoup(`*mockEntry*`)`
handler`:EntryStore(`*mockEntry*`)`
handler`:EntrySize(`*mockEntry*`)`
handler`:EntryTextSize(`*mockEntry*`)`
handler`:EntryUniqueID(`*mockEntry*`)`
handler`:EntryModTime(`*mockEntry*`)`
handler`:EntryChange(`*mockEntry*`)`
handler`:EntryChangeWithModTime(`*mockEntry*`)`
handler`:EntryRemoveFromSoup(`*mockEntry*`)`
handler`:EntryReplace(`*original*`, `*replacement*`)`
handler`:EntryReplaceWithModTime(`*original*`, `*replacement*`)`
handler`:EntryUndoChanges(`*mockEntry*`)`
handler`:EntryCopy(`*mockEntry*`, `*newSoup*`)`
handler`:EntryMove(`*mockEntry*`, `*newSoup*`)`
handler`:EntryValid(`*mockEntry*`)`

Drawing and Graphics Reference

This chapter describes the protos, functions, and methods used by the drawing interface.

Data Structure

The Drawing interface uses the following structure.

Style Frame

The style frame is used to specify characteristics that affect the way the shape is imaged, such as the size of the pen or the fill pattern to be used. These characteristics are specified by the values of slots in a style frame associated with the shape. If the value of the style frame is `nil`, the view system draws the shape using default values for these drawing characteristics.

Slot descriptions

The style frame contains one or more of the slots listed here. If any single slot is not provided, the default value for that slot is used.

`transferMode` The drawing transfer mode for the pen (or for the text, if text is being drawn). Specify one of these standard constants: `modeCopy`, `modeOr`, `modeXor`, `modeBic`, `modeNotCopy`, `modeNotOr`, `modeNotXor`, `modeNotBic`. See "viewTransferMode Constants" (page 2-13) for a description of these constants. The default transfer mode is a split state: bitmaps and text are drawn with a `modeOr` transfer mode, but other items (geometric shapes, pens, and fill patterns) are drawn with a `modeCopy` transfer mode. However, when you actually specify a transfer mode (by placing a non-`nil` value in the `transferMode` slot of the style frame), all drawing uses the specified mode.

`penSize` The size of the pen in pixels. You can specify a single integer to indicate a square pen of the specified size, or you can specify an array giving the pen width and height (for example, `[1, 2]`). This value is not used for drawing text. The minimum and default pen size is 1. However, no frame will be drawn for a shape if `penPattern` is set to `vfNone` (the default `penPattern` is `vfBlack`).

`penPattern` The pen pattern. You can specify the following patterns: `vfNone`, `vfWhite`, `vfLtGray`, `vfGray`, `vfDkGray`, and `vfBlack`. The default value is `vfBlack`.

To use a custom pen pattern, store a binary object of class `'pattern` in this slot. An easy way to create such an object is to clone a binary string containing 16 Unicode hexadecimal digits, set the class of the clone to `'pattern` and store the result in this slot. For more information, see "Custom Fill and Frame Patterns" (page 3-21) in the *Newton Programmer's Guide*.

`fillPattern` The fill pattern. You can specify the same patterns as for the `penPattern` slot. This value is not used for drawing text. The default value is `vfNone`.

To use a custom fill pattern, store a binary object of class `'pattern` in this slot. For more information, see "Custom Fill and Frame Patterns" (page 3-21) in the *Newton Programmer's Guide*.

`font` The font to use for drawing text. The default is the font selected by the user in the Styles palette. See "Fonts for Text and Ink Display" (page 8-3) in *Newton Programmer's Guide* for details on specifying a font.

`justification` The alignment of text in the rectangle specified for it. Specify one of the following symbols: `'left`, `'right`, `'center`. The default value is `'left`.

`clipping` Specifies a clipping region to which all drawing is clipped in addition to the default clipping. The value of this slot can be a primitive shape, a region, or an array of shapes (from which a new clipping region is constructed automatically by the system). For more information see "Controlling Clipping" (page 13-12) in the *Newton Programmer's Guide*.

`transform` Used to offset or scale the shape. The value of this slot is an array that can hold a coordinate pair or a pair of source and destination rectangles. For more information, see "Transforming a Shape" (page 13-13) in the *Newton Programmer's Guide*.

View Classes

The following view classes are used to display objects in views.

Shape View (clPolygonView)

Displays polygons or ink, or accepts graphic or ink input.

Slot descriptions

`viewBounds`	Set to the size of the view and the view location where you want it to appear.
`points`	If the view contains a polygon shape, this slot contains a binary data structure of the type `'polygonShape`, which holds the polygon data.
`ink`	If the view contains ink, this slot contains a binary data structure of the type `'ink`, which holds the ink data.
`viewFlags`	The default setting is `vVisible`. You will most likely want to set additional flags to control the recognition behavior of the view; for example, `vShapesAllowed`.
`viewFormat`	Optional. The default setting is `vfPen(2)`. The `vfPen` setting controls the thickness of polygon lines.

Picture View (clPictureView)

Displays a picture. A picture can be a bitmap, graphic shape, or picture object.

Slot descriptions

`icon`	A bitmap, graphic shape, or picture object to be displayed in the view. A bitmap is selected from a resource file by using the icon slot editor in NTK. A picture object is obtained from a resource file by using the `GetResource` or `GetNamedResource` compile-time functions in NTK.
`viewBounds`	Set to the size of the view and the location where you want it to appear.
`viewFlags`	The default setting is `vVisible`.
`viewFormat`	Optional. The default setting is `nil`.

A picture object is simply a binary object with the class `'picture`.

If the contents of the `icon` slot is a graphic shape, the style frame for drawing the shape in the view contains the single slot `transferMode`. The `transferMode` slot is set to the same value as the `viewTransferMode` slot of the view (if this slot exists), or to the default value `modeCopy` if there is no `viewTransferMode` slot in the view.

Your graphic shape can provide a different set of styles by including a style frame in the shape array. In this case, any `transferMode` slot in the style frame that you specify overrides the `viewTransferMode` setting for the view.

Scaled View (clRemoteView)

Displays a scaled image of another view.

Slot descriptions

`stepChildren`	Specify a single child view in this array. This child view is scaled to fit inside the `clRemoteView`. Typically, you set this slot at run time in the `ViewSetupFormScript` method.
`viewBounds`	Set to the size of the view and the location where it is to appear.
`viewFlags`	The default setting is `nil`.
`viewFormat`	Optional. The default setting is `nil`.

Graphics and Drawing Protos

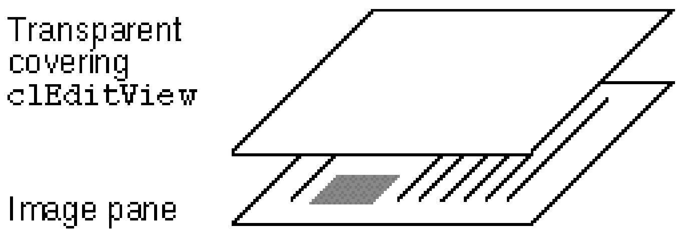
This section describes the protos that work with graphics and drawing. The protos include:

- `protoImageView`
- `protoThumbnail`
- `protoThumbnailFloater`

protoImageView

This proto provides a view in which you can display, magnify, scroll, and annotate images. However, it depends on the use of `protoThumbnail` and `protoThumbnailFloater` to provide controls for magnifying, scrolling, and paging. The structure of the `protoImageView` is shown in Figure 10-1.

Figure 10-1 `protoImageView` Structure



The annotations can be selected and modified when the image is shown at full size. The image and annotations are clipped so that only the portion of their contents that falls within the bounds of their parent view is shown. Annotations scroll along with the image.

In general, in this discussion, a reference to the “image” means both the image and the annotation, while the “image plane” refers only to the image. Also, references to the “pane” refer to the bounding box of the `protoImageView`, under the assumption that the image is larger than can be displayed in the box, so the `protoImageView` is a window, or pane, into the larger image. Finally, scaling frequently refers to both size and position of the pane in the image.

Slot descriptions

You may provide the following slots:

`Image` This slot should contain a NewtonScript shape. It will be rendered by the image plane and can be provided either by proto or parent inheritance. This slot is

	required if the image viewer is not opened with `OpenImage` or `ToggleImage`.
`Annotations`	This slot should either be `nil`, or should contain an array of views appropriate to be added as `viewChildren` to a `clEditView`. This slot can be provided either by proto or parent inheritance. **Note** `Annotations` is referenced during view setup (see `Setup` (page 10-9) for details) and is not maintained afterwards; to retrieve user annotations, call the `GetAnnotations` method. ◆
`scalingInfo`	This slot should either be `nil` or should contain a slot similar to that returned by `GetScalingInfo` (page 10-11). The `scalingInfo` slot can be provided by either proto or parent inheritance.

You can override the following slots:

`viewBounds`	The default is `{top: 88, left: 0, right: 0, bottom: -24}`.
`viewJustify`	The default setting is `vjParentFullH + vjParentFullV`.
`viewFlags`	The default setting is `vfLtGray + vfFillShift`.
`viewFormat`	The default setting is `vfPen(1)`.
`zoomStops`	An array specifying an ordered set of zoom stops, smallest to largest, used by the `ZoomBy` method. If this slot is not provided, it is initialized to the default set. Each item in the set should be either a number or a symbol. If a number, `zoomStops` specifies the fractional size to be displayed, where 1.0 is the size of the original image based on the resolution. If `zoomStops` is a symbol it may be `'fitInWindow`, `'fullSize`, `'fullResolution`, or `'twiceFullResolution`. The minimal default set is `['fullSize, 'twiceFullResolution]`. The symbol `'fullSize` should always be a member of the array.

`dragCorridor` An integer. When dragging the image, clinging to the closest axis when within a specific corridor smooths linear scrolling considerably. The `dragCorridor` slot specifies the distance from the closest axis the user must move the pen to break out of that corridor and scroll diagonally. The default value is 7 (resulting in a 14-pixel corridor along both axes).

`grabbyHand` When appropriate, a picture is painted under the pen while `pendown` is executing to indicate that the image can be dragged. The `grabbyHand` slot contains the appropriate shape to render. It should have `top-left = 0, 0`. The picture is automatically centered under the pen.

Note

This slot can only be generated dynamically and must be generated before `ViewSetupDone` is called. ◆

Do not change the following slot:

`declareSelf` This slot is set by default to `'imagebase`. Do not change it.

The following additional slots and methods are used internally. They are listed here so that you don't inadvertently override them.

System slots:

`viewClass, declareSelf, and ViewSetupDoneScript`

Additional slots:

`myImage, tempImage, tempAnnotes, tempScales, tempOpen, fXOffset, fYOffset, fMaxX, fMaxY, cHorMult, cVertMult, fAnnotateMode, handShape, usefulSizes, currentSize, fullSize, fZoomedTo, quiet, CalculateUsefulSizes, SetupZoomStops, SetupSizes, ZoomByDest, DoUndo.`

The following sections describe the methods of `protoImageView` that you may need to use.

PenDown

myImageView:`PenDown(`*strokeUnit*`)`

Used to drag an image.

Called by the image view's `ViewClickScript` to handle taps (except when in `'edit` mode, see "SetAnnotationMode" (page 10-12). The default script drags the image. You can override the default to handle the click. Keep in mind that it is not possible to override `ViewClickScript` as `protoImageView` is composed of multiple views, any one of which can be handling the tap.

strokeUnit	Unit from the `ViewClickScript` method; contains information describing the interaction of the pen with the display.

ScalingInfoChanged

myImageView:`ScalingInfoChanged(`*slot*`)`

Called whenever a frame returned by `GetScalingInfo` would change due to some programmatic action; for example, a call to `ZoomTo`, `ScrollBy`, and so on.

slot Value varies depending on the event causing the `GetScalingInfo` call:

`'zoom`	The magnification of the image changed.
`'scroll`	The image was scrolled.
`'dragging`	The image is being dragged by the pen.
`'dragDone`	The image is finished being dragged by the pen.

Setup

myImageView:`Setup(`*image*`,` *annotations*`,` *scalingInfo*`)`

Performs appropriate initialization to display the specified image. This method is typically used after the view is opened to let another image be displayed (for example, when switching pages in a fax). (Note that the `ViewSetupDoneScript` method calls `Setup` automatically.)

image — Contains a NewtonScript shape which is rendered by the image plane.

annotations — Is either `nil` or contains an array of views appropriate to add as `viewChildren` to a `clEditView`.

scalingInfo — If specified, *scalingInfo* sets the image to the appropriate magnification and offset.

Note
You can define your own `Setup` method; however, you must then call the inherited method (`inherited:Setup();`) from your own method. ◆

OpenImage

myImageView:`OpenImage(`*image*`,` *annotations*`,` *scalingInfo*`)`

Opens and initializes the view displaying the image, annotations, and whatever scaling it was set to. If *scalingInfo* is `nil`, the image size does not change; however the annotation may change. Otherwise, the image sets the scaling according to the specified scaling information. If the image is already open the imagery, annotations, and scaling (if specified) are set.

image — Contains a NewtonScript shape which is rendered by the image plane.

annotations — Is either `nil` or contains an array of views appropriate to add as `viewChildren` to a `clEditView`.

scalingInfo — If specified, *scalingInfo* sets the image to the appropriate magnification and offset.

ToggleImage

myImageView:`ToggleImage(`*image*`,` *annotations*`,` *scalingInfo*`)`

Opens or closes the view and sets the image, annotations, and scaling information (if specified). If *scalingInfo* is `nil`, the image size does not change. If the image is already open, the image, annotations, and scaling information are set.

image	Contains a NewtonScript shape which is rendered by the image plane.
annotations	Is either `nil` or contains an array of views appropriate to add as `viewChildren` to a `clEditView`.
scalingInfo	If specified, *scalingInfo* sets the image to the appropriate magnification and offset.

GetScalingInfo

myImageView:`GetScalingInfo()`

Returns a frame of scaling information. The returned frame has the following slots:

`offsetX`	The horizontal offset of the pane within the image (positive).
`offsetY`	The vertical offset of the pane within the image (positive).
`zoomedTo`	The symbol or number representing the current zoom.
`extent`	The bounding box of the image at the current scale.
`viewBox`	The (localbox) bounding box of the pane (never changes).

HasAnnotations

myImageView:`HasAnnotations()`

Returns non-`nil` if the displayed image has annotations, `nil` otherwise.

GetAnnotations

myImageView:`GetAnnotations()`

Returns an array of views appropriate to become `clEditView` children. This array represents the current annotations drawn on the `clEditView` annotation layer.

SetAnnotationMode

myImageView:`SetAnnotationMode(`*theMode*`)`

Sets the annotation display behavior and the pen behavior when it is tapped.

theMode Specifies the mode as follows:

`'hide`	Annotations are not visible and a pen tap results in a drag.
`'show`	Annotations are made visible, and a pen tap drags.
`'edit`	Annotations are visible and editable.

Note
Due to system limitations, it is not possible to edit annotations at any magnification other than `'fullSize`. If you attempt to `SetAnnotationMode('edit)` while at any other magnification, an exception is thrown. ◆

GetAnnotationMode

myImageView:`GetAnnotationMode()`

Returns the symbol representing the current annotation mode.

TargetChanged

myImageView:`TargetChanged()`

Called when any annotation is added or edited.

CanScroll

myImageView:`CanScroll()`

Returns a frame indicating the direction (`'left`, `'right`, `'up`, and `'down`) in which scrolling is possible. If scrolling is not possible `nil` is returned.

ScrollT0

myImageView:`ScrollTo(`*x*`,` *y*`)`

Scrolls the scaled image within the clipping window. This method returns a non-`nil` value if the image was moved or `nil` if it was not moved (either it was already there, or doing so would have moved the pane past the edge of the image). `ScrollTo` does not scroll the image away from the edge of the view.

x, y	The offset of the top-left corner of the pane from the top left corner of the image.

Note

Zooming the image changes the size (and content) of the image window, but doesn't change the scrolling behavior. ◆

ScrollBy

myImageView:`ScrollBy(`*x*`,` *y*`)`

Scrolls the image by the specified offset amount, where *deltaX* and *deltaY* indicate how far to move the pane within the image. This method returns a non-`nil` value if the image was moved or `nil` if it was not moved. `ScrollBy` does not scroll the image away from the edge of the view.

x	The horizontal distance in which to scroll the image.
y	The vertical distance in which to scroll the image.

ZoomBy

myImageView:`ZoomBy(`*direction*`)`

Makes an image larger or smaller as specified by the sizes in the `zoomStops` array. If the current zoom is a number between a pair of stops, the image increases to the nearest stop in the direction specified (where a positive number enlarges the image; a negative number shrinks the image).

The following example shows the use of `zoomStops`:

```
['fitInWindow, 0.24, 0.5, 'fullSize, 2, 4,
'fullResolution, 'twiceFullResolution]
```

The current zoom is 0.35, `ZoomBy`(1) increases the image by 0.5 (that is, half size), `ZoomBy`(2) makes the image `'fullSize`, and so on. `ZoomBy` returns non-`nil` if the zooming was changed.

direction A number of discrete steps by which to zoom the image.

ZoomTo

myImageView:`ZoomTo(`*imageSize*`)`

Changes the size of the image.

imageSize An positive number or symbol as described in the `scalingInfo` slot on (page 10-7).

CanZoomBy

myImageView:`CanZoomBy(`*imageSize*`)`

Returns `nil` if `ZoomBy` would change the size of the image. Returns non-`nil` otherwise.

imageSize A number of discrete steps by which to zoom the image.

ZoomToBox

myImageView:`ZoomToBox(`*boundsFrame*`)`

Resizes the image to the size specified with the *boundsFrame* parameter. Note that you don't need to specify the same aspect ratio as the original image; this method allows you to stretch the image in either dimension.

boundsFrame Specifies the size to which you want the image to resize.

protoThumbnail

This proto is designed to be used in conjunction with `protoImageView`. It displays a small copy of the image (a "thumbnail" sketch), with a rectangle representing the location of the pane in the image.

In this discussion, the grey box refers to the rectangle representing the location of the pane in the image. Scaling frequently refers to both the size and position of the grey box in the thumbnail.

Slot descriptions

You may provide the following slots:

`ImageTarget` — This slot should point to a view capable of responding to both the `GetScalingInfo` and the `ScrollTo` methods. If this slot is defined, the thumbnail proto does not need to provide these two methods.

`Image` — If this slot is present when the image is opened, it is expected to contain a graphic shape or bitmap that is used to render the background shape—the thumbnail sketch—in the view. If this slot is present, it must not be `nil`.

You can override the following slots to modify the appearance of the grey box or thumbnail:

`viewBounds` — The default is `{top: 12, left: -50, right: -2, bottom: -23}`.

`viewJustify` — The default setting is `vjParentRightH + vjParentFullV`.

`trackWhileScrolling` — If non-`nil`, this slot causes intermediary calls to `ScrollTo` while the grey box is being dragged around the thumbnail. If `nil`, `ScrollTo` is called only when the pen is lifted.

The following additional slots and methods are used internally. They are listed so that you don't inadvertently override them.

System slots:

`viewClass, viewFlags, viewFormat, ViewClickScript, ViewSetupDoneScript, ViewDrawScript`

Additional slots:
`tempImage, thumbnail, thumbnailBounds, greyBox, greyBounds, theShape, needToUpdate, RelocateGreyBox`

You can invoke the `protoThumbnail` methods described in the following sections.

Setup

myThumbnail:`Setup(`*image)*

Prepares the thumbnail to show a new image created from `protoThumbnail`. The image is scaled and rendered into an internal bitmap image. This is useful for large images, as it reduces memory paging.

image	The image to be scaled.

OpenThumbnail

myThumbnail:`OpenThumbnail(`*image*`)`

Convenience routine to open thumbnails. If *image* is specified and an `image` slot is available, the parameter takes precedence. `OpenThumbnail` internally calls `Setup`.

image	The image to display.

ToggleThumbnail

myThumbnail:`ToggleThumbnail(`*image*`)`

Opens or closes the image. If the image is open, it is closed. If the image is closed, `ToggleThumbnail` calls `OpenThumbnail`.

image	The image to open or close.

Update

myThumbnail:`Update()`

Re-renders the thumbnail view, which can be a fairly slow process, as the grey box is rescaled. This slot is necessary only if the scaling information (the location or magnification) of the source image has changed; generally the standard `Dirty` call should suffice.

GetScalingInfo

myThumbnail:`GetScalingInfo()`

Must return a frame of scaling information like that returned by the `GetScalingInfo` method of `protoImageView`. The easiest way to do this is simply to call the `GetScalingInfo` method of an instance of a `protoImageView`.

PrepareToScroll

myThumbnail:`PrepareToScroll()`

Called immediately before any scrolling is performed to allow you to perform any necessary preparation.

ScrollTo

myThumbnail:`ScrollTo(`*x*`,` *y*`)`

Called to scroll the view if a pen down event causes scrolling (the usual case). Again, the easiest way to scroll is to call the `ScrollTo` method of an instance of a `protoImageView`. This method must be provided if the view can be clicked on.

x, *y*	The position of the pen in the thumbnail scaled to the `size` and `extent` slot in the frame returned by `GetScalingInfo`. For example, if the thumbnail is 10x10 and the extent is 100x100, a pen down at position 3,5 in the thumbnail results in a call to `ScrollTo(30, 50)`.

DoneScrolling

myThumbnail:`DoneScrolling()`

Called following the scroll operation to allow any necessary clean-up to be performed.

protoThumbnailFloater

This proto provides a convenient way to use a thumbnail. It follows the same basic conventions as the `protoThumbnail`, with the added benefit of being based on the `protoFloatNGo` proto so that it adjusts its size to reflect the aspect ratio of the image it contains. It is always as large as possible without getting any larger in either dimension than the original `viewBounds`. Furthermore, it adjusts its bounds so that only the edges farthest away from the parent's closest edge move. In other words, if the floater is dragged to the top-left, the bottom-right corner moves, if it is at the bottom-right corner of the parent, only its top-left corner changes.

▲ **WARNING**
This proto should not be parent full-justified, as this will break the code that adjusts its size. ▲

All of the slots are defined and used identically to the `protoThumbnail`, with the following additions that are used internally:

`maxW`, `maxH`, `ViewSetupFormScript`

Functions and Methods

This section describes drawing functions and methods. It contains the following topics:

- Functions to handle bitmaps
- Functions to handle hit-testing
- Functions to handle creating shapes
- Functions that operate on shapes
- General utility functions

Bitmap Functions

This section describes the bitmap functions and methods.

MakeBitmap

`MakeBitmap(`*widthInPixels*, *heightInPixels*, *optionsFrame*`)`

Returns a blank (white) bitmap shape of the specified size. The origin of the bitmap returned is at (0,0); however, you can subsequently use the `OffsetShape` function to modify the returned bitmap's origin.

widthInPixels Width of the bitmap shape.

heightInPixels Height of the bitmap shape.

optionsFrame An optional frame specifying additional characteristics of the bitmap shape created by this method. It can contain any of the slots specified here. If this frame is not used, the value of the *optionsFrame* parameter must be `nil`.

`rowBytes` Specifies the number of bytes per row of the bitmap; use only for a data source that creates scan lines longer than the default value. An `exMakeBitmapBadArgs` exception is thrown if the value of `rowBytes` is not a multiple of 32 bits or is too narrow for the bitmap's width as specified by the *widthInPixels* parameter. When no other value is specified, this slot has the default value `BAND(`*widthInPixels*` + 31, -32) / 8`.

`resolution`

Specifies high- or low-resolution images. Like a pen size, the value of the `resolution` slot may be an array or a single value. If this value is an array, the elements of the array specify the *x* and *y* dimensions of the pixels comprising the bitmap. If this slot stores a single value, it specifies that the pixels are square, having equal values for their *x* and *y* dimensions. Applications that display or otherwise manipulate bitmap documents (for example, fax pages) need to use this slot to control scaling functionality. This slot's default value is `[72,72]` when no other value is specified.

`store` By specifying a store, the bitmap is created as a VBO (virtual binary object). To applications, VBOs appear to be NewtonScript binaries, but they are actually handled directly by the system, using automatic compression and decompression to allow these objects to be much larger than the available heap space. If you are going to create a bitmap, and you know that it will ultimately wind up in a soup on a particular store, you can increase the system efficiency by using this slot to specify the store on which to create the object.

If this slot is `nil`, the NewtonScript heap is used, and the bitmap will not be a VBO. You need to limit the use of the NewtonScript heap to small bitmaps only.

A throw occurs in the event the NewtonScript heap or store does not have enough space for the bitmap.

`companderName`
When a VBO is written to the store, the system uses a compander, or compression-decompression utility. This slot is a string that represents the name of the compander to use when writing or reading this bitmap from the store.

The default compander is `TPixelMapCompander`, which is efficient for monochrome images. During compression, the data is preprocessed by XORing scan lines. It is then passed on to the Lempel Ziv implementation contained in the ROM.

You can supply your own compander as a protocol. If you don't want to compress the data when written out to the store you would need to supply an appropriate protocol. If you are not writing to the store (default), then there is no compression, no VBO, and the data is written out to the frames heap.

`companderData`
This slot is intended for optional arguments that would be passed to the compander. The default is `nil`.

DrawIntoBitmap

`DrawIntoBitmap(`*shape*, *styleFrame*, *destBitmap*`)`

Draws shapes into a bitmap in the same way that the `DrawShape` method (page 10-35) draws shapes into a view. Drawing is clipped to the boundaries of the destination bitmap.

shape	Any of the shapes returned by the shape-creation functions, or an array of such shapes intermixed with optional style frames. If a style frame is included in the shape array, the style frame applies to all subsequent shapes in the array, until overridden by another style frame.
styleFrame	A style frame as specified in the description of the `DrawShape` method (page 10-35).
destBitmap	The bitmap in which drawing takes place.

To perform offscreen buffering, your application's `ViewDrawScript` method can use the `DrawIntoBitmap` function to create a bitmap and then draw that bitmap into the final onscreen view by sending the `DrawShape` message to the view.

MungeBitmap

`MungeBitmap(`*bitmap*, *operator*, *options*`)`

Performs various bitmap operations such as rotating or flipping the bitmap. These operations are destructive to the bitmap passed as an argument to this function; the bitmap is modified in place and the modified bitmap shape is returned.

bitmap — A bitmap shape on which this function operates. For convenience, the bitmap shape is modified in place and the modified bitmap shape is returned in this slot.

operator — A symbol specifying the bitmap modification to perform; it may have any of the following values:

`'flipHorizontal`
: Flips the image bitwise horizontally (mirror image).

`'flipVertical`
: Flips the image bitwise vertically (mirror image).

`'rotateLeft`
: Rotates the image 90 degrees left.

`'rotateRight`
: Rotates the image 90 degrees right.

`'rotate180`
: Rotates the image 180 degrees; unlike `'flipHorizontal`, the image is not mirrored.

The `'flip`*Direction* operators return a shape having the same dimensions as the source bitmap; no view bounds or other rectangles are changed.

The `'rotate`*Direction* operators, however, change the dimensions of the object; therefore, they change the returned bitmap's bounds rectangle to reflect the new size and shape.

If the source bitmap has been offset, the coordinates of the upper-left corner of the returned object are the same as those of the source bitmap and the coordinates of the bottom-right corner of the returned bitmap are changed. See the *Newton Programmer's Guide*, Figure 13-11 (page 13-19) for an illustration of the relationships between the coordinates of the source bitmap and those of the returned bitmap.

options The options frame contains slots for the support of future munge operations. Only one slot is supported at this time:

`callBack` A callback function provided by you to allow display of the progress of the three rotation operations to the user. The munge operations call this function with an array argument, ranging from 0 to 100 inclusive, representing the completion percentage of the rotation operation.

ViewIntoBitmap

view`:ViewIntoBitmap(` *srcRect*`,` *destRect*`,` *destBitmap*`)`

Writes a portion of the specified view into the specified bitmap. This function always returns `nil`. This function does not provide a scaling capability, although scaling can be accomplished by passing the *destBitmap* bitmap returned by this method to the `DrawIntoBitmap` function as the value of its *shape* parameter. See the *Newton Programmer's Guide*, Figure 13-10 (page 13-19) for a graphical depiction of the relationships between the view to be captured, the source rectangle, the destination bitmap, and the destination rectangle.

srcRect — The portion of the view that is to be captured, specified as a rectangle in the local coordinate system of the source view. A value of `nil` causes this function to use the view bounds of the source view as the dimensions of the source rectangle. The size of the source rectangle is clipped to the intersection of *destRect* and the bounds of the destination bitmap.

Because *srcRect* expects local coordinates, you may need to call *myview*: `localBox()` to get correct coordinates of *srcRect* if `myView` is justified relative to other views.

destRect — Defines the bounds of the portion of the bitmap into which the image is drawn. A value of `nil` causes the view bounds of *srcRect* to be used as the default value of *destRect*. The bounds of *destRect* are clipped to stay within the bounds of the destination bitmap.

destBitmap — The bitmap shape into which the captured view image is written. You can use the `MakeBitmap` function to create this shape.

Hit-Testing Functions

The following functions allow you to determine whether a point or stroke lies within a specified shape.

HitShape

`HitShape(`*shape*, *x*, *y*`)`

Indicates whether the point described by the *x* and *y* coordinate parameters lies within the shape.

x — The *x* coordinate of the point to be tested, in local (view) coordinates.

y — The *y* coordinate of the point to be tested, in local (view) coordinates.

shape — A shape returned by one of the routines that creates shapes (such as `MakeRect`, `MakeOval`, `MakeRegion`, `MakePolygon`, and so on.) You can specify an array of shapes for *shape*, and in this case, each shape in the array is hit-tested with the point. If a hit is found, the function returns immediately and subsequent shapes in the array are not tested.

When a single shape is passed to this function, it returns non-`nil` if the specified point lies within the boundaries of the shape and `nil` if the specified point does not lie within the boundaries of any shape passed to it. For unclosed polygons, the result of this function is undefined. When passed an array of shapes, this function returns an "array path" indicating the shape within which the point lies. The "array path" is an array in which each element represents an index into the nested array of shapes passed to `HitShape`.

PtInPicture

`PtInPicture(`*x*, *y*, *bitmap*`)`

Returns non-`nil` if the point described by the *x* and *y* coordinates lies at a black pixel. If no mask is defined for the specified bitmap, this function tests whether the point lies within the bitmap itself. This function returns `nil` if the point is outside the test area.

`PtInPicture` supports bitmaps loaded from resources using the compile-time function `GetPictAsBits` or those created using `MakeBitmap`.

x — The *x* coordinate of the point to be tested, in local (view) coordinates.

y — The *y* coordinate of the point to be tested, in local (view) coordinates.

bitmap — The bitmap object associated with the mask to be tested.

Returns non-`nil` if the point described by the *x* and *y* coordinates lies within the mask associated with the specified bitmap object. If no mask is defined for the specified bitmap, this function tests whether the point lies within the bitmap itself. This function returns `nil` if the point is outside the test area.

Shape-Creation Functions

These global functions create shape objects which you can draw using the `DrawShape` method.

MakeLine

`MakeLine(`*x1*`, `*y1*`, `*x2*`, `*y2*`)`

Creates and returns the specified line shape.

x1	The *x* coordinate of the first point drawn.
y1	The *y* coordinate of the first point drawn.
x2	The *x* coordinate of the last point drawn.
y2	The *y* coordinate of the last point drawn.

MakeRect

`MakeRect(`*left*`, `*top*`, `*right*`, `*bottom*`)`

Creates and returns the specified rectangle shape.

left	The *x* coordinate of the top-left corner of the rectangle.
top	The *y* coordinate of the top-left corner of the rectangle.
right	The *x*-coordinate of the bottom-right corner of the rectangle's enclosing rectangle.
bottom	The *y*-coordinate of the bottom-right corner of the rectangle's enclosing rectangle.

Note

If a rectangle is drawn with four line calls, the bottom and right sides of the rectangle will lie outside the bottom right line coordinates by the amount of the pen width and height. ◆

MakeRoundRect

`MakeRoundRect(`*left*`, `*top*`, `*right*`, `*bottom*`, `*diameter*`)`

Creates and returns a rounded rectangle shape (a rectangle having rounded corners).

left	The *x* coordinate of the top-left corner of the rectangle.
top	The *y* coordinate of the top-left corner of the rectangle.
right	The *x*-coordinate of the bottom-right corner of the rounded rectangle's enclosing rectangle.
bottom	The *y*-coordinate of the bottom-right corner of the rounded rectangle's enclosing rectangle.
diameter	The curvature of the rectangle's corners, specified as if a circle of the given diameter, in pixels, were placed in each of the rectangle's corners.

Note

If a rectangle is drawn with four line calls, the bottom and right sides of the rectangle will lie outside the bottom right line coordinates by the amount of the pen width and height. ◆

MakeOval

`MakeOval(`*left*`, `*top*`, `*right*`, `*bottom*`)`

Creates and returns an oval shape. The oval is drawn to fit just *inside* the specified rectangle. If you specify a rectangle that is square, this method draws a circle.

left	The *x* coordinate of the top-left corner of the oval's enclosing rectangle.
top	The *y* coordinate of the top-left corner of the oval's enclosing rectangle.
right	The *x*-coordinate of the bottom-right corner of the oval's enclosing rectangle.

bottom The *y*-coordinate of the bottom-right corner of the oval's enclosing rectangle.

MakeWedge

`MakeWedge(`*left*`,` *top*`,` *right*`,` *bottom*, *startAngle*`,` *arcAngle*`)`

Draws an arc as a part of an oval that fits just within the specified rectangle. If you draw the wedge with no fill, you see just the arc line. If you draw the shape with a visible fill pattern, you see a solid wedge shape.

left The *x* coordinate of the top-left corner of the arc's enclosing rectangle.

top The *y* coordinate of the top-left corner of the arc's enclosing rectangle.

right The *x* coordinate of the bottom-right corner of the wedge's enclosing rectangle.

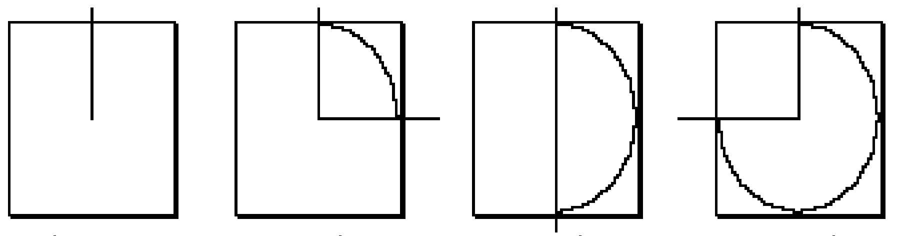
bottom The *y* coordinate of the bottom-right corner of the wedge's enclosing rectangle.

startAngle The angle at which the arc begins, in positive (clockwise) or negative (counterclockwise) degree values.

arcAngle The angle through which the arc extends, in positive (clockwise) or negative (counterclockwise) degree values.

The angles are given in positive or negative degrees; a positive angle goes clockwise, while a negative angle goes counterclockwise. Zero degrees is at 12 o'clock high, 90 (or -270) is at 3 o'clock, 180 (or -180) is at 6 o'clock, and 270 (or -90) is at 9 o'clock. Other angles are measured relative to the enclosing rectangle: a line from the center of the rectangle through its top-right corner is 45 degrees, even if the rectangle isn't square; a line through the bottom-right corner is at 135 degrees, and so on.

Figure 10-2 Angles for arcs and wedges



MakePolygon

`MakePolygon(`*pointArray*`)`

Creates and returns the specified polygon graphic object.

pointArray An array of *x* and *y* coordinate pairs specifying the vertices of the polygon.

MakeShape

`MakeShape(`*object*`)`

Creates and returns a shape based on *object*. `MakeShape` may return a shape that is smaller in size than what you would get if you did the equivalent capture of a view into a bitmap with `ViewIntoBitmap.`

`Object` The following kinds of shapes are created, depending on what kind of object is passed in *object*:

rectangle You can pass in a bounds frame describing a rectangle. A bounds frame has the following slots: `left`, `top`, `right`, `bottom`. A rectangle shape is created and returned.

points You can pass in the value stored in the `points` slot in a view of class `clPolygonView`. This is a binary data structure that has a class of 'polygonShape and contains data describing a polygon shape.

A polygon shape is created and returned.

Note

This option is intended to create a shape from data you retrieve from a clPolygonView. However, you can manually create the points data structure by using the ArrayToPoints routine. ◆

bitmap — You can pass in a bitmap frame object. A bitmap shape is created and returned. You can use the compile-time function GetPictAsBits to create a bitmap from a PICT resource; for more information, see *Newton Toolkit User's Guide*.

picture — You can pass in a picture. A picture shape is created and returned.

view — You can pass in a view. A picture shape is created and returned.

MakeRegion

MakeRegion(*shapeArray*)

Creates and returns a region of arbitrary size, shape, and complexity. You define a region by defining its boundary with other shape-drawing functions. The boundary can be any set of lines and shapes (even including other regions) forming one or more closed loops. A region can be concave or convex, can consist of one connected area or many separate areas.

shapeArray — An array of shapes returned by any of the shape-making functions described in this section.

MakePict

MakePict(*shapeArray*, *styleFrame*)

Creates and returns a picture shape that is made by recording a sequence of drawing operations. This groups several drawn shapes into a single graphical entity which is easier, smaller, and faster to use in subsequent drawing operations than drawing each shape individually every time.

This function works exactly like the `DrawShape` method except that the shapes are not drawn on the screen, but are instead drawn into a picture shape object that is returned.

shapeArray — An array of shapes to draw using the characteristics specified in *styleFrame*. The shapes can be any of those returned by the shape-creation functions, and the shape array can include other style frames intermixed with the shapes. If a style frame is included in the shape array, it applies to all subsequent shapes in the array, until overridden by another style frame.

styleFrame — A frame having one or more of the slots "Style Frame" (page 10-1). If this frame is `nil`, the default values are used. If any single slot is not provided, the default value for that slot is used.

MakeText

`MakeText(`*string*`, `*left*`, `*top*`, `*right*`, `*bottom*`)`

Creates and returns a text shape drawn within the specified rectangle. The font used for the text is specified as the value of a slot in the style frame. `MakeText` can create only one line of text at a time.

string — The text string to be drawn.

left — The *x* coordinate of the top-left corner of the text's enclosing rectangle.

top — The *y* coordinate of the top-left corner of the text's enclosing rectangle.

right — The *x* coordinate of the bottom-right corner of the text's enclosing rectangle.

bottom — The *y* coordinate of the bottom-right corner of the text's enclosing rectangle.

When drawn, the baseline of the text is placed at the bottom of the rectangle you specify as an argument to the `DrawShape` method. The text is clipped horizontally to the nearest letter boundary within the rectangle, but it is not clipped vertically. The text is aligned to the left, right, or center of the rectangle you specify, as controlled by the `justification` slot in the style frame associated with the text shape.

MakeTextLines

`MakeTextLines(`*string*`, `*bounds*`, `*lineheight*`, `*font*`)`

Creates and returns a text shape drawn within the specified rectangle. The text shapes are made and wrapped in relation to the dimensions of the bounds frame specified by the value of the *box* parameter. Words are scanned in until the end of a line is reached, as determined by the width of *box*. The location of the next line is determined by the value of *lineheight*. Text shapes are made until either the string terminates or the limits of the box are reached. In the event that the first word on a line is longer than the width of the box, a partial word is made on the line.

string	The text string to be drawn.
bounds	The dimension of the bounds frame.
lineheight	The location of the next text line to be drawn, in pixels, before the current line.
font	The the font used to draw the text.

TextBox

`TextBox(`*text*, *fontFrame*, *bounds*`)`

Draws text on the screen without using shapes or creating a view for the drawing. You generally use this call during a `ViewDrawScript`.

text	The string to draw.

fontFrame A frame contains both a font specification and, optionally, a justification.

The format is:

```
{
   font: <font-spec>,
   justification: <justification>
}
```

`font-spec` can be a font frame or an integer specification.

`justification` a slot that is either: `['left|'center|'right]`. If the slot is not present or the slot is `nil`, it defaults to `'left`.

bounds The bounds frame in which to draw.

`TextBox` is used for single-font text or single-font text with ink; that is, you can pass it a rich string. Multi-style text must be drawn differently, usually with a `clParagraphView`.

Unlike `MakeText`, which can only be used for a single line of text `TextBox` wraps the text within specified box. Also, unlike `MakeText`, it doesn't consume frames heap space, because no shape object is created.

Note

The bounds are in local coordinates of the view that makes the `TextBox` call. See the text sample code on this subject for an example. ◆

Shape Operation Functions and Methods

These methods and global functions operate on shapes returned from the shape-creation functions described in the previous section.

You also can do hit-testing on shapes using the `HitShape` method (page 10-24).

GetShapeInfo

`GetShapeInfo(`*shape*`)`

Returns a frame containing information about certain kinds of graphics shapes.

shape — Any of the shapes returned by the shape-creation functions, or an array of such shapes intermixed with optional style frames. If a style frame is included in the shape array, it applies to all subsequent shapes in the array, until overridden by another style frame.

For all shapes, the returned frame contains a `bounds` slot. For text shapes, the returned frame additionally contains a `text` slot; modifying this string does not affect the text shape. For bitmaps created using `MakeBitmap`, the frame contains the following slots.

▲ **WARNING**
Do not rely on `GetShapeInfo` or the following slots for shape created by other applications, images stored in the Newton ROM, images created with functions other than `MakeBitmap`, or images with a depth other than 1. ▲

`bits` — The binary object containing bitmap data. The bitmap data can be manipulated at run time (or copied for non-Newton use) using the other slots in the return value of `GetShapeInfo` to interpret the bitmap binary object: `scanOffset`, `bitBounds`, and `rowBytes`.

`bitsBounds` — The size of the bitmap itself, expressed as the boundaries of a rectangle having a (0,0) origin.

`bounds` — The boundaries of the scaled and offset bitmap.

`depth` — An integer expressing the number of bits per pixel. Newton 2.0 OS currently supports only the value 1. See `MakeBitmap` (page 10-19) for storage details.

`resolution` — An integer specifying the resolution of the bitmap, expressed in dots per inch. For example, the built-in fax viewer application uses this slot to store the resolution of the fax image.

`rowBytes` — An integer specifying the number of bytes per horizontal row in the bitmap image.

`scanOffset` — An offset into the `bits` slots that specifies where bitmap image data begins, expressed as the number of bytes from the beginning of the bitmap.

`store` — The store on which a virtual binary object's soup resides. The value of this slot is `nil` for shapes that are in the NewtonScript heap. See "MakeBitmap" (page 10-19) for storage details.

The following code example shows how the first bit of an image created with `MakeBitmap` (with depth equal to 1) can be obtained:

```
bitmapInfo := GetShapeInfo(theBitmap);
firstByte := ExtractByte(bitmapInfo.bits, bitmapInfo.scanOffset);
firstBit := firstByte >> 7; // 1 or 0, representing on or off.
```

Note that `rowBytes` will always be 32-bit aligned. For example, a bitmap with depth equal to 1 with a `bitBounds` slot having a width of 33 pixels, `rowBytes` will be 8 to indicate 8-byte offsets per horizontal line and 31 bits of unused data at the end of every horizontal line.

DrawShape

view`:DrawShape (`*shape*`, `*styleFrame*`)`

Draws the specified shape (or shapes) in the view using the characteristics specified in *styleFrame*.

shape — Any of the shapes returned by the shape-creation functions, or an array of such shapes intermixed with optional style frames. If a style frame is included in the shape array, it applies to all subsequent shapes in the array, until overridden by another style frame.

styleFrame — A frame having one or more of the slots listed in “Style Frame” (page 10-1). If this frame is `nil`, the default values are used. If any single slot is not provided, the default value for that slot is used.

Note that style frame values don’t apply to drawing shapes that are pictures; they are drawn as is. When drawing bitmaps, only the `transferMode` slot is used; the other slots in the style frame don’t apply.

OffsetShape

`OffsetShape (`*shape*`, `*deltaH*`, `*deltaV*`)`

Returns the shape with its bounds offset from the original bounds as specified.

shape — The shape to be offset.

deltaH — The horizontal amount by which to offset the specified shape from its original bounds.

deltaV — The vertical amount by which to offset the specified shape from its original bounds.

You can specify an array of shapes for *shape* in which case each shape in the array will be offset. This function is destructive to the shape you pass it; that is, it modifies and returns that shape.

ScaleShape

`ScaleShape (`*shape*`, `*srcRect*`, `*dstRect*`)`

Enlarges or reduces one or more shapes from the size specified by the rectangle *srcRect* to the size specified by the rectangle *dstRect* and returns the scaled shape(s). This function is destructive to the *shape* argument; that is, it modifies and returns its value.

shape — A shape or array of shapes to be scaled.

srcRect — A view bounds frame defining a rectangle that encloses the shape at its original size. The frame has the slots `left`, `top`, `right`, `bottom`. If this frame is `nil`, the

	shape's original bounds are used as the source rectangle, effectively scaling the shape from its current size to the size of the destination rectangle.
dstRect	A view bounds frame defining a rectangle that encloses the shape at its modified size. The frame has the slots `left`, `top`, `right`, `bottom`.

Note

If the widths and heights of the source and destination rectangles are not proportionate, the returned shape is distorted to fit exactly within the destination rectangle, even if this means that the width and height of the shape are scaled unequally. ◆

ShapeBounds

`ShapeBounds (`*shape*`)`

shape	A shape or array of shapes.

Returns a bounds frame describing the rectangle that encloses the shape. The bounds frame has the following slots: `left`, `top`, `right`, `bottom`. You can specify an array of shapes for *shape*, and in this case, this function returns the rectangle that encloses the entire group of shapes.

InvertRect

view:`InvertRect(`*left*`, `*top*`, `*right*`, `*bottom*`)`

Inverts the specified rectangle in the current view. It is important to send this message to a particular view so that the inversion display can be clipped properly.

left, *top*	Defines the left-top corner of the rectangle, relative to the local view.
right, *bottom*	Defines the right-bottom corner of the rectangle, relative to the local view.

InsetRect

`InsetRect(`*rect*`, `*deltax*`, `*deltay*`)`

Shrinks or expands the rectangle you specify with the *rect* frame: the left and right sides are moved in by the amount you specify in the *deltax* parameter; the top and bottom are moved toward the center by the amount you specify in the *deltay* parameter. If the value you pass in *deltax* or *deltay* is negative, the appropriate pair of sides is moved outward instead of inward. The effect is to alter the size by `2*`*deltax* horizontally and `2*`*deltay* vertically, with the rectangle remaining centered in the same place in the coordinate pair.

aBounds	The bounds of the rectangle to alter.
deltax	The horizontal distance to move the left and right sides in toward or outward from the center of the rectangle.
deltay	The vertical distance to move the top and bottom sides in toward or outward from the center of the rectangle.

IsPtInRect

`IsPtInRect(`*x*`, `*y*`, `*bounds*`)`

Checks to see if the point specified by *x* and *y* is in the bounds of the rectangle. Returns non-`nil` if the point (*x*, *y*) is inside *bounds*. Otherwise, it returns `nil`.

bounds	The bounds of the rectangle to check.
x	The horizontal distance to check.
y	The vertical distance to check.

FitToBox

`FitToBox(`*sourceBox*`, `*boundingBox*`, `*justify*`)`

Makes a box fit into another box while maintaining the source box's original aspect ratio and justifying that resulting box to *boundingBox*'s original aspect ratio, and justifying that resulting box to *boundingBox* according to the justify parameter. The result is a bounds rectangle.

sourceBox — The bounds rectangle you're trying to fit into *boundingBox*.

boundingBox — The area you have to display in, that is, the local box of a view.

justify — An integer encoded the same way as a `viewJustify` slot.

The constants `vjCenterH`, `vjLeftH`, `vjRightH`, `vjCenterV`, `vjTopV`, and `vjBottomV` are supported.

OffsetRect

`OffsetRect(`*rect*`, `*deltaX*`, `*deltaY*`)`

Returns a bounds frame that is *rect* moved to the right by *deltaX* and down by *deltaY*.

rect — The size of the rectangle and location where you want it to appear.

deltaX — How much to offset the horizontal coordinates in the frame.

deltaY — How much to offset the vertical coordinates in the frame.

SectRect

`SectRect(`*rect1*`, `*rect2*`)`

Returns a bounds frame that is the intersection of *rect1* and *rect2*. For example, if you pass *rect1* and *rect2*, you get a result frame similar to `{ left: 15, top: 25, right: 100, bottom: 50 }`. If *rect1* and *rect2* do not intersect, an empty bounds frame is returned, for example:

```
rect1 := SetBounds(0, 0, 50, 50)
      rect2 := SetBounds(100, 100, 150, 150)
      sect := sectRect(rect1, rect2)
      {left: 0, top: 0, right: 0, bottom: 0}
```

rect1 and *rect2* — The rectangles of which to find the intersection.

UnionRect

`UnionRect(`*rect1*`, `*rect2*`)`

Returns a bounds frame that is determined by the smallest rectangle that encloses both *rect1* and *rect2*. If *rect1* is `nil`, a bounds frame with the same coordinates as *rect2* is returned.

rect1	A bounds frame describing the first rectangle.
rect2	A bounds frame describing the second rectangle.

RectsOverlap

`RectsOverlap(`*rect1*`, `*rect2*`)`

Checks to see if there is an overlap between two specified rectangles. Returns non-`nil` if the two rectangles overlap, otherwise it returns `nil.`

rect1	A bounds frame describing the first rectangle.
rect2	A bounds frame describing the second rectangle.

Utility Functions

This section describes additional drawing functions and methods.

DoDrawing

view`:DoDrawing(`*drawMethodSym*`, `*parameters*`)`

Ensures that any drawing done by the *drawMethodSym* method does not overwrite other obscuring views (such as floating views that may be partially obscuring the view in which this method draws). Using the `DoDrawing` method is the preferred way to draw objects other than in a view's `ViewDrawScript` method.

`DoDrawing` sets the clipping according to the setting of the `vClipping` flag for the specified view, invokes the view's *drawMethodSym* method, and restores clipping to what it was before the *drawMethodSym* method was called. `DoDrawing` passes through the return value of the method called.

drawMethodSym Quoted symbol specifying the method that performs drawing operations; for example, to indicate use of the `DrawShape` method, pass the symbol `'DrawShape` as the value of this parameter.

parameters An array of parameters to pass to the *drawMethodSym* method. Set the value of this argument to `nil` if the *drawMethodSym* method accepts no arguments.

Note
If the view's `vClipping` flag is not set, drawing is not clipped to the view's bounds but to the view bounds of the hierarchically closest parent view having its `vClipping` flag set. ◆

CopyBits

view`:CopyBits(`*picture*`, `*x*`, `*y*`, `*mode*`)`

Draws a bitmap in the specified location in the view using the specified transfer mode.

picture A reference to the bitmap object to be drawn. You can use the compile-time function `GetPictAsBits` to create a bitmap from a `PICT` resource; for more information, see the *Newton Toolkit User's Guide*.

x The *x* coordinate of the top-left corner of the bitmap.

y The *y* coordinate of the top-left corner of the bitmap.

mode One of the standard drawing transfer modes: `modeCopy`, `modeOr`, `modeXor`, `modeBic`, `modeNotCopy`, `modeNotOr`, `modeNotXor`, `modeNotBic`. If you pass `nil` for mode, the default, `modeCopy`, is used. These constants are described in "viewTransferMode Constants" (page 2-13).

Note
`CopyBits` uses the bitmap's `bounds` slot to scale the bitmap. So, by changing the bounds of a bitmap (or more likely, a clone of a bitmap), you can perform scaling. ◆

DrawXBitmap

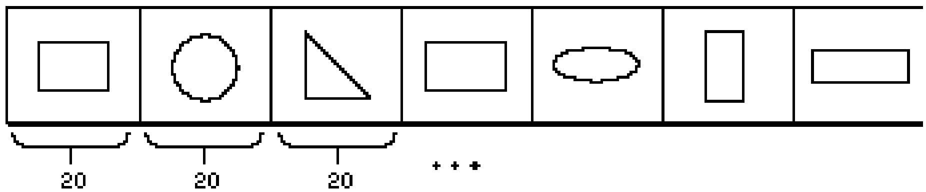
`DrawXBitmap(`*bounds*`,`*picture*`,`*index*`,`*mode*`)`

Draws a bitmap extracted from the specified portion of a larger bitmap composed of a horizontal row of equal width bitmaps.

bounds	The size of the bitmap and the location in which it is to be drawn in the current view.
picture	A reference to the bitmap object to be drawn. You can use the compile-time function `GetPictAsBits` to create a bitmap from a `PICT` resource; for more information, see *Newton Toolkit User's Guide*.
index	The index in the bitmap resource of the particular bitmap that is to be drawn.
mode	One of the standard drawing transfer modes: `modeCopy`, `modeOr`, `modeXor`, `modeBic`, `modeNotCopy`, `modeNotOr`, `modeNotXor`, `modeNotBic`. These constants are described in "viewTransferMode Constants" (page 2-13).

The width of each row in the *picture* bitmap is assumed to be the width as specified in the *bounds* parameter; thus, if you specify a width of 20 pixels and an index of 2, the chunk of *picture* beginning at pixel 40 and extending horizontally through pixel 59 will be extracted and drawn:

Figure 10-3 Row width of picture bitmap



IsPrimShape

`IsPrimShape(`*shape*`)`

Returns non-`nil` if the object passed in is a primitive shape (not an array of shapes). Because a shape can have many different internal structures, this procedure is the only way to reliably verify that an object is a shape. If the object is not a shape, this function returns `nil`.

shape Any shape returned by one of the shape-creation functions.

Note
This object fails on an array of shapes—it returns non-`nil` only if the object is a single shape. ◆

LockScreen

view`:LockScreen(`*lock*`)`

Prevents the screen from updating, or reverses the effect of a previous call to the `LockScreen` method.

lock A Boolean value; when set to `true`, the screen is locked and no updates can occur. To unlock the screen, call the `LockScreen` method again with *lock* set to `nil`.

Normally, all drawing occurs in off-screen memory and the system periodically updates the screen bits from the off-screen memory. This function prevents the copying of bits to update the screen. This allows you to make drawing calls, or erase and redraw things without an accompanying flicker if the screen happens to update during your drawing sequence. When you finish drawing, call `LockScreen(nil)` to unlock the screen.

Here's how you would typically use `LockScreen`:

```
...
:LockScreen(TRUE);
:DoDrawing(myDrawFnSym, nil)
:LockScreen(nil);
...
```

Note

The system automatically calls `LockScreen` before sending a view the `ViewDrawScript` message, and unlocks the screen afterwards. Therefore, you don't have to call `LockScreen` in your `ViewDrawScript` method, but only when you want to draw at some other time. ◆

PointsToArray

`PointsToArray(`*polygonShape*`)`

Converts the points data in a polygon shape between the binary data structure in the shape view and an array. Returns an array defining the polygon.

polygonShape A binary object of the class `'polygonShape` (from the `points` slot of a `clPolygonView`).

The first element in the returned array is an integer identifying the shape type). The second element in the array is the number of points. Beginning with the third array element, the remainder of the array consists of coordinate pairs describing the points. The third element contains the *x* coordinate of the first polygon point and the fourth element contains the *y* coordinate, and so on. Coordinates are relative to the top-left corner (0, 0) of the `clPolygonView`.

Here is an example of an array returned from a rectangle shape:

```
[11, 5, 0, 0, 0, 29, 40, 29, 40, 0, 0, 0]
```

The first element, 11, describes the array as a rectangle; the second element, 5, indicates that there are five points in the shape; and the remaining elements describe the five points—(0, 0), (0, 29), (29, 40), (40, 0), and (0, 0).

ArrayToPoints

`ArrayToPoints(`*pointsArray*`)`

Converts an array of points to a binary object of the class `'polygonShape` (as found in the `points` slot of a `clPolygonView`). The binary object is returned. The shape type (element 1 of the array) can be either 4 for a closed polygon (the ending point is the same as the starting point) and 5 if it is an open polygon (different starting and ending points).

pointsArray An array of points in the same format as returned by `PointsToArray`.

Sound Reference

This chapter describes the data structures, the `protoSoundChannel`, ROM sounds, and sound functions in Newton system software.

Sound Data Structures

Sound has two data structures: the sound frame and the sound result frame. Each structure is described in the following sections.

Sound Frame

The sound frame defines a sound and contains one or more of the slots listed here. If any single slot is not provided, its default value is used.

Slot descriptions

`sndFrameType` Specifies the format of this sound frame. Currently, Newton sound frames always have the symbol `'simpleSound` in this slot; future Newton devices may store other values here. Required.

`samples` — A frame of class `'samples` containing the binary sound data. The sound data must have been sampled at 11 khz or 22 kHz. Required.

`samplingRate` — A floating-point or integer value specifying the rate at which to play back the sample data. The constants `kFloat11kRate` and `kFloat22kRate` can be used to specify standard rates of 11 kHz and 22 kHz, respectively. If this slot is not provided, the default value of 22 kHz is used.

`compressionType` — Currently, the value of this slot is always `kNone`, indicating no compression.

`dataType` — Integer. Size of samples in bits. Optional. If present, it must be 1 (`k8Bit`). If missing, `k8Bit` is assumed.

`start` — Integer. Index of first sample to begin play. Optional. If missing, 0 is assumed.

`count` — Integer. Number of samples to play. If missing, `Length(samples) / (dataType/8)` is assumed.

`loops` — Integer. Number of times to repeat the sound. For example, setting loops to 3 means play the sound a total of four times. Optional. If missing, 0 is assumed.

You can invoke the following method, which provides status information about the sound that was played.

Callback

`Callback(`*state*`,` *result*`)`

Invoked when the sound frame completes.

state — State is one of the following:

0 =`kSoundCompleted`

1 = `kSoundAborted`

2 = `kSoundPaused`

result Result is an error code, if any.

Sound Result Frame

A sound result frame returns status information about a sound operation. It has the following slots:

Slot descriptions

`sound` Reference to the `soundFrame` that was paused, stopped, or completed.

`index` Index of the sample where the sound was paused or stopped. This number will be between `soundFrame.start` and (`soundFrame.start + soundFrame.count`).

Protos

Sound uses one proto: `protoSoundChannel`.

protoSoundChannel

The `protoSoundChannel` object provides methods that implement pause, playback, and callback of sounds. It also provides query methods that return whether the sound is running or paused.

Open

soundChannel`:Open()`

Opens the sound channel. This method throws an `|evt.ex.fr|` exception if an error occurs; otherwise, it returns `nil`.

Close

soundChannel:`Close()`

Closes the sound channel. This method throws `|evt.ex.fr|` exception if an error occurs; otherwise, it returns `nil`.

Note

You must call `Close`; because, the sound channel is not disposed of in garbage collection. ◆

Schedule

soundChannel:`Schedule(`*soundFrameRef*`)`

Queues `soundFrame` for play. This method throws an `|evt.ex.fr|` exception if an error occurs; otherwise, it returns `nil`. As each sound completes, the sound channel sends the `Callback` message to the `soundFrame` (if defined).

soundFrameRef The sound frame to be played. See "Sound Frame" (page 11-1) for a list of slots and a description of the `Callback` function.

Start

soundChannel:`Start(`*async*`)`

Starts the sound channel. The channel begins playing sound frames in the order they were scheduled (see the previous description). This method throws an`|evt.ex.fr|` exception if an error occurs; otherwise it returns `nil`.

async A Boolean value of `true` or `nil`. If *async* is `nil`, the call does not return until the entire play queue is empty (all scheduled sounds have completed).

Stop

soundChannel:`Stop()`

Stops the sound channel. The channel stops all scheduled sound frames, including the currently playing one, if any. Throws an `|evt.ex.fr|` exception if an error occurs. Returns a sound result frame (page 11-3) indicating which sound frame was stopped, or `nil` if no sound was currently playing. All scheduled sound frames complete (via the `callback` function) with state 1 `(kSoundAborted)`.

Pause

soundChannel:`Pause()`

Temporarily halts the current playback process in the specified sound channel. If the sound channel is stopped when this message is sent, the message starts the channel, pausing its operation at the beginning of the sound data. If the sound channel is paused, the message resumes playback of the sound.

IsPaused

soundChannel:`IsPaused()`

Returns `true` if the specified sound channel is paused; otherwise, returns `nil`.

IsActive

soundChannel:`IsActive()`

Returns `true` if the channel is active (playing or paused); otherwise, returns `nil`.

Functions and Methods

The functions and methods described in this section play sounds, generate telephone dialing tones, and allow you to get and set the playback volume.

Dial

`Dial(`*numberString*`,` *where*`)`

Dials the specified telephone number synchronously as a deferred action, using the speaker or modem as specified. To dial asynchronously, use the global `RawDial` (page 11-7) function.

This function always returns non-`nil`.

numberString A string specifying the number to dial. Acceptable values for this string include only the digits `0-9`, the alphanumeric characters `A-Z`, and the special characters `*`(asterisk), `#` (pound), `-` (dash) and `,` (comma). This function maps alphabetic characters to the tones that a standard telephone keypad generates for these characters. The letters Q and Z, which are not present on a standard telephone keypad, are mapped to the digit 1. Note that the letters A - D do not generate the specialized DTMF dialing tones used by some phone systems; these letters are mapped to the tones that they would produce on a standard telephone keypad. The dash (-) character inserts a delay of 50 milliseconds when dialing and the comma (,) character inserts a delay of 500 milliseconds when dialing.

where A symbol, either `'speaker` or `'modem` specifying whether to dial through the speaker or modem, respectively.

For example:

```
GetRoot():Dial("555-1212", 'modem)
```

GetVolume

`GetVolume()`

Returns the current volume setting for sounds. This is an integer from `0-4`.

PlaySoundSync

`PlaySoundSync(`*soundFrameRef*`)`

Plays a sound defined by the specified sound frame. The sound is played synchronously; that is, the Newton stops everything it's doing, plays the sound, and then this function returns. This function always returns `true`.

soundFrameRef The sound frame to be played. See "Sound Frame" (page 11-1) for a list of slots.

Note
If you want to be notified when the sound completes, use the sound channel interface instead. ◆

RawDial

`RawDial(`*numberString*`,` *where*`)`

Dials the specified telephone number asynchronously, using the speaker or modem as specified.

This function always returns `true`.

numberString A string specifying the number to dial. Acceptable values for this string include only the digits `0-9`, the alphanumeric characters `A-Z`, and the special characters `*`(asterisk), # (pound), `-` (dash), and (comma). This function maps alphabetic characters to the tones that a standard telephone keypad generates for these characters. The letters Q and Z, which are not present on a standard telephone keypad, are mapped to the digit 1. Note that the letters A - D do not generate the specialized DTMF dialtones used by some phone systems; these letters are mapped to the tones that they

	would produce on a standard telephone keypad. The dash (-) character inserts a delay of 50 milliseconds when dialing and the comma (,) character inserts a delay of 500 milliseconds when dialing.
where	A symbol, either `'speaker` or `'modem`, specifying whether to dial through the speaker or modem, respectively.

SetVolume

`SetVolume(`*volume*`)`

Sets the output level for all sounds. The default level is `4`, which is the highest volume level. This function always returns `nil`.

volume	An integer value from 0 to 4, specifying the level at which sound is to be played. The value 0 turns sound output off completely and the value 4 specifies the highest available sound output level.

PlaySoundAtVolume

`PlaySoundAtVolume(`*soundFrameRef*`,` *volume*`)`

Plays a sound defined by the specified sound frame. The sound is played asynchronously; that is, this function returns immediately and the sound is played as a background process. This function always returns `true`. The sound sets the volume before playing and restores it when it is complete.

soundFrameRef	The sound frame to be played. See "Sound Frame" (page 11-1) for a list of slots.
volume	An integer value from 0 to 4 specifying the level at which sound is to be played. The value 0 turns sound output off completely and the value 4 specifies the highest available sound output level. If volume is nil, the current sound volume is used.

PlaySoundIrregardless

`PlaySoundIrregardless(`*soundFrameRef*`)`

Plays a sound, independent of the user sound preference settings (action and pen sound effects). The sound is played asynchronously; that is, this function returns immediately and the sound is played as a background process. This function always returns `true`.

soundFrameRef	The sound frame to be played. See "Sound Frame" (page 11-1) for a list of slots.

PlaySoundIrregardlessAtVolume

`PlaySoundIrregardlessAtVolume(`*soundFrameRef*`, `*volume*`)`

Plays a sound at the specified volume, independent of the user sound preference settings (action and pen sound effects), and restores the sound when it completes. The sound is played asynchronously; that is, this function returns immediately and the sound is played as a background process. This function always returns `true`.

soundFrameRef	The sound frame to be played. See "Sound Frame" (page 11-1) for a list of slots.
volume	The volume at which to play the sound.

PlaySoundEffect

`PlaySoundEffect(`*soundFrameRef*`, `*volume*`, `*type*`)`

Plays the sound at the specified volume, if user preferences allow the sound, and restores the sound volume when it completes.

volume	The volume at which to play the sound.
soundFrameRef	The sound frame to be played. See "Sound Frame" (page 11-1) for a list of slots.
type	Can be one of `'pen`, `'alarm`, or `'action`.

Clicker

`Clicker()`

Plays a different “click” sound each time you call it. Use this for pen sounds instead of `PlaySound`.

Sound Resources

The system provides a number of sounds in ROM that are available to accompany various events; these sounds are referenced by the following constants.

Note
Don't rely on anything about these sound objects except being able to play them. Sound characteristics such as sampling rate and format, and the sounds themselves, will change in future system versions. ◆

`ROM_alarmWakeup`
: The sound played when the Newton powers on automatically to display an alarm.

`ROM_click`
: The sound played when the user taps items such as buttons and close boxes.

`ROM_crumple`
: The first sound played when deleting an item from the Notepad; it accompanies an animated simulation of the note being wadded into a ball.

`ROM_drawerClose`
: The sound played as the Extras Drawer closes.

`ROM_drawerOpen`
: The sound played as the Extras Drawer opens.

`ROM_flip`
: The sound played when turning pages in a Book Maker book.

`ROM_funBeep`
: The Trill sound in the user preferences Sound panel.

`ROM_hiliteSound`	The sound played to indicate to the user that the Newton device is in highlight mode, rather than inking mode. This sound plays when the user presses the stylus against the screen continuously; it is accompanied by the display of the highlighting mark.
`ROM_plinkBeep`	The Xylo sound in the user preferences Sound panel.
`ROM_simpleBeep`	The Bell sound in the user preferences Sound panel.
`ROM_wakeupBeep`	The sound played when the Newton is powered on.
`ROM_plunk`	The second sound played when deleting an item from the Notepad; it depicts the sound of the crumpled note hitting the Trash.
`ROM_poof`	The sound played when an item is scrubbed; it accompanies the animated cloud that depicts the item "going up in smoke."

Filing Reference

This section describes data structures, system prototypes, functions, and methods your application can use to support the Filing service.

Target Information Frame

The frame returned by the `GetTargetInfo` method. The root view supplies a default version of this method that returns frames used by the Filing and Routing services. The built-in applications override this method to return their own target information frames. You can override this method to return your own information frame as well. In addition to any slots that you supply, the frame your override method returns must contain the slots described here.

Slot descriptions

`target` The data item that is the object of the operation in progress; that is, the item to be filed, routed, or otherwise manipulated. The default version of the `GetTargetInfo` method retrieves this value from the `target` slot in the view that receives this message, using full proto and parent inheritance to find the slot.

`targetView` The view to which the Filing service sends messages; for example, the view to which Filing sends the `FileThis` message. For Routing, this is the view that contains the target item. The default version of the `GetTargetInfo` method retrieves this value from the `targetView` slot in the view that receives this message, using full proto and parent inheritance to find the slot.

`targetStore` The store selected when the Filing slip is opened. This value must be the store on which the `target` entry resides or `nil`. For Routing, this slot identifies the store on which the target entry resides. The default version of the `GetTargetInfo` method returns `nil` for this value.

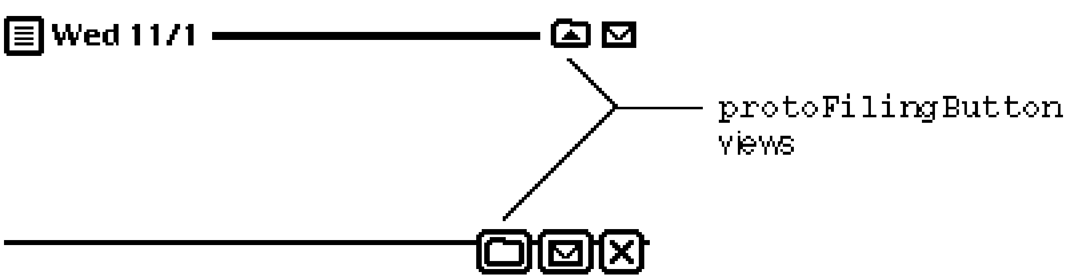
Filing Protos

This section describes system-supplied button and folder tab prototypes that you can use to implement filing support.

protoFilingButton

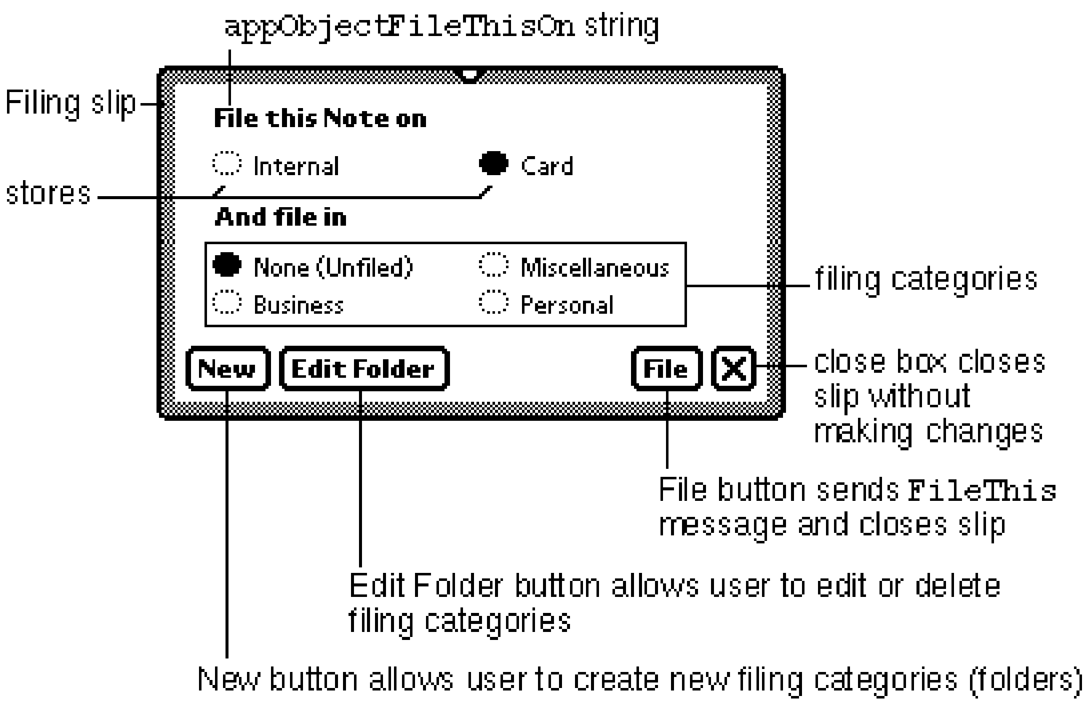
Used to create the filing button that appears on your application's status bar or title bar, as shown in Figure 12-1. This proto is used in conjunction with the `protoNewFolderTab` or `protoClockFolderTab` system prototype to implement filing for an application.

Figure 12-1 Two examples of filing button views



When the user taps the filing button, the system displays a Filing slip similar to the one shown in Figure 12-2. The Filing slip contains radio buttons that the user can tap to specify the filing category with which the target is to be associated. When appropriate, this slip can also display buttons that specify the store on which a soup entry is to reside when it is filed. This slip also contains buttons that allow the user to create new filing categories and edit or delete existing categories. When the user taps the File button in this slip, the system sends the `FileThis` message to your application's target view.

Figure 12-2 The Filing slip



IMPORTANT
Do not override the `ViewClickScript` method that the `protoFilingButton` proto supplies; this method is for system use only. Your filing button view can supply a `ButtonClickScript` method instead of a `ViewClickScript` method. Your filing button view's `ButtonClickScript` method must call the inherited `ButtonClickScript` method that the `protoFilingButton` proto supplies. ▲

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

Slot descriptions

`viewBounds`	Set to the size and location where you want the filing button to appear. If your application provides a status bar or title bar, it is recommended that you put the filing button on one or both of these bars.
`viewJustify`	Optional. The default setting is `vjCenterH + vjCenterV + vjSiblingLeftH`.
`viewFormat`	Optional. The default setting is `vfFillWhite + vfFrameBlack + vfPen(2) + vfRound(4)`.

The Filing slip provides default versions of the `ViewSetupFormScript`, `ButtonClickScript`, and `Update` methods. If you provide your own version of one of these methods, be sure that it calls the inherited method; otherwise, the Filing slip may not work correctly. To call the inherited method only when it is defined, use the conditional message-send operator (`:?`), as shown in the following code fragment:

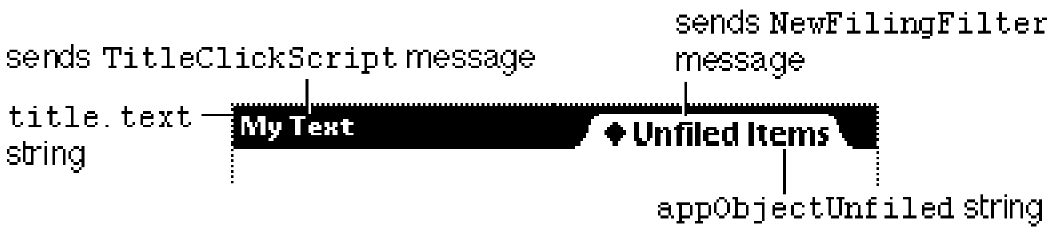
```
inherited:?ViewSetupFormScript()
```

protoNewFolderTab

The `protoNewFolderTab` proto provides a folder tab view that displays an optional text string, as shown in Figure 12-3. This proto is used with the `protoFilingButton` proto to support the Filing service.

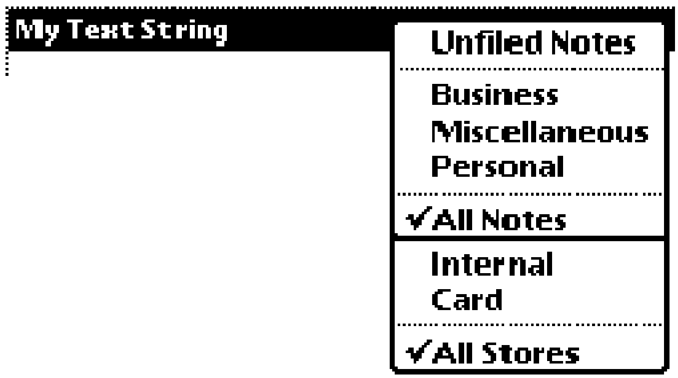
The folder tab view is positioned at the top of your application's base view. You can supply an optional text string that is displayed at the left of the folder tab; when the user taps this text, the folder tab view sends the `TitleClickScript` message to the target view.

Figure 12-3 A `protoNewFolderTab` view with optional title text



When the user taps the folder tab, it displays a picker that includes all currently available filing categories (folders) as well as the "Unfiled" and "All *Items*" categories. Optionally, this picker may include items allowing the user to specify the store on which displayed items must reside. A check mark appears next to the currently selected filing category and store in this picker, as shown in Figure 12-4.

Figure 12-4 The picker displayed by a `protoNewFolderTab` view



The user can tap an item in the list to display a different filing category or store. When the user chooses a filing category or store from the picker, the system sends the `NewFilingFilter` message to your application, collapses the picker, and updates the folder tab text to display the currently selected filing category.

To include title text in a `protoNewFolderTab` view, create a child view that is declared to the folder tab view and include in the child view a title slot containing the string that is the optional text.

▲ **WARNING**
Do not create a `title` slot in your folder tab view. Optional title text must reside in the `text` slot of the `title` child view provided by the `protoNewFolderTab` view. ▲

The `protoNewFolderTab` proto provides the following slots of interest to developers:

Slot descriptions

`viewBounds` Optional. The default view bounds supplied by this proto position it at the top of its parent. Do not set this slot unless you need to change the normal positioning of the `protoNewFolderTab` view.

`title` Optional. Do not create this slot yourself; it holds the child view that images an optional text string at the left side of the `protoNewFolderTab` view.This view contains a `text` slot that holds your optional title text string. This view is declared to the folder tab view.

`text` Optional. The string that is your folder tab view's optional title text.

TitleClickScript

myFolderTabView`:TitleClickScript`

Optional application-defined method that is invoked when the user taps the title text at the left of the folder tab. The default version of this method does nothing.

myFolderTabView A view based on the `protoNewFolderTab` or `protoClockFolderTab` system prototype.

ViewDrawScript

myFolderTabView`:ViewDrawScript`

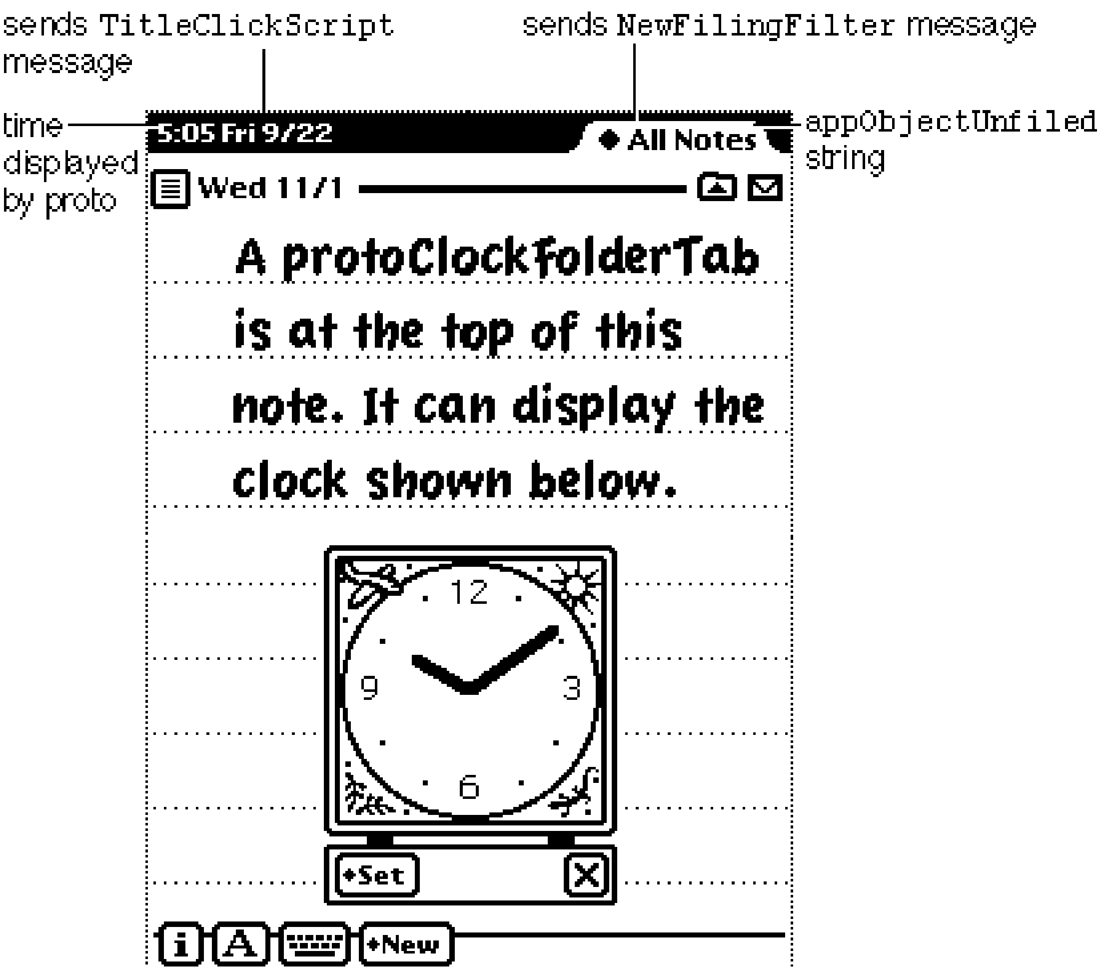
For internal use. If you provide your own version of this method, make sure it calls the inherited version; otherwise the folder tab view may not work as expected.

myFolderTabView A view based on the `protoNewFolderTab` or `protoClockFolderTab` system prototype.

protoClockFolderTab

The `protoClockFolderTab` proto provides a folder tab view that displays the current time, as shown in Figure 12-5. This illustration also depicts the built-in Clock application that is opened by this proto's default `TitleClickScript` method. This proto is used with the `protoFilingButton` proto to support the Filing service.

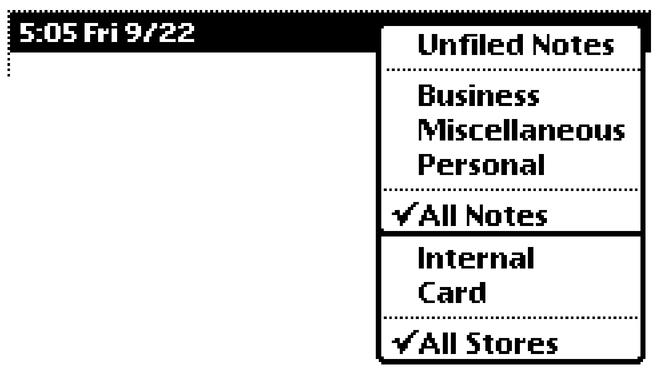
Figure 12-5 The `protoClockFolderTab` view



The folder tab view is positioned at the top of your application's base view. When the user taps the time displayed at the left of the folder tab, the folder tab view sends the `TitleClickScript` message to the target view. The default version of this method opens the built-in Clock application. You can override the `TitleClickScript` method to take other action in response to this event. Do not attempt to replace the display of the current time in this view; to provide your own text here, use a `protoNewFolderTab` view (page 12-4) instead of a `protoClockFolderTab` view.

When the user taps the folder tab, it displays a picker that includes all currently-available filing categories (folders) as well as the "Unfiled" and "All *Items*" categories. Optionally, this picker may include items allowing the user to specify the store on which displayed items must reside. A check mark appears next to the currently selected filing category and store in this picker, as shown in Figure 12-6.

Figure 12-6 Selecting a filing category and store in a `protoClockFolderTab` view



The user can tap an item in the list to display a different filing category or store. When the user chooses a filing category or store from the picker, the system sends the `NewFilingFilter` message to your application, collapses the picker, and updates the folder tab text to display the currently selected filing category.

▲ **WARNING**
Do not attempt to replace the title text that displays the current time in `protoClockFolderTab` views. ▲

The `protoNewFolderTab` proto provides the following slots of interest to developers:

Slot descriptions

`viewBounds` Optional. The default view bounds supplied by this proto position it at the top of its parent. Do not set this slot unless you need to change the normal positioning of the `protoNewFolderTab` view.

`title` Optional. Do not create this slot yourself. The child view that images an optional text string to the left of the folder tab child view. This view contains a `text` slot that holds your optional title text string.

`text` Optional. The string that is your folder tab view's optional title text.

TitleClickScript

myFolderTabView`:TitleClickScript`

Optional application-defined method invoked when the user taps the time displayed at the left of the folder tab. The default version of this method opens the built-in Clock application.

myFolderTabView A view based on the `protoNewFolderTab` or `protoClockFolderTab` system prototype.

ViewDrawScript

myFolderTabView`:ViewDrawScript`

For internal use. If you provide your own version of this method, make sure it calls the inherited version; otherwise the folder tab view may not work as expected.

myFolderTabView A view based on the `protoNewFolderTab` or `protoClockFolderTab` system prototype.

System-Supplied Filing Methods

This section describes the functions and methods you can use to provide Filing features.

GetTargetInfo

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

Returns a target information frame required by system services such as Filing and Routing. The frame this method returns specifies the item that is the object of the action (such as the item to file or route), the view to which the system service sends messages (usually your application's base view) or which contains the target item and, when necessary, the store on which the target item resides.

The default version of this method is provided by the root view; the built-in applications override this method. You can override this method to provide additional information in your target info frame or to define additional values for the *reason* parameter. If you override this method, your override method must call the inherited version of the `GetTargetInfo` method.

reason — A symbol specifying the operation for which the target information is required. The default method recognizes the symbols `'filing` and `'routing` as valid values for this parameter. This parameter is useful if you override this method. It is provided as a hook for you to implement special behavior depending on its value.

For descriptions of the slots in the target information frame that this method returns, see the section "Target Information Frame" beginning on page 12-1.

When using this method for routing, it must return only a single target item. If multiple items are selected for routing, you need to create a single object that encapsulates them. You can use the function `CreateTargetCursor` (page 18-24) to create a multiple-item target object that can be stored in a soup. (Normal soup cursors can't be stored in a soup.)

MoveTarget

targetView:`MoveTarget(`*target*`,` *destStore*`)`

Moves or copies the specified target to the specified store. If the target is an entry in a read-only soup, its data is copied rather than moved to the destination soup; that is, the original entry is not deleted from the source soup.

The default version of this method moves a soup entry to the same-named soup on the specified store; it is used by the system-supplied filing service. You can override this method to move data other than soup entries. Your override method should call the default method supplied by the root view to move soup entries.

target	The target data to be moved. If this argument is not a soup entry, the default `MoveTarget` method does nothing; thus, your override method can call the default `MoveTarget` method to handle soup entries.
destStore	The store to which this method moves the target data, expressed as an index into the stores array; for example, `GetStores()[0]; // the internal store`

RegFolderChanged

`RegFolderChanged(`*callbackID*`,`*callBackFn*`)`

Registers a callback function to execute when the user adds, removes, or edits a folder. The return value of this method is unspecified; do not rely on it.

callbackID	Unique symbol identifying the *callBackFn* function to the folder change mechanism. Because this symbol must be unique among all symbols registered with the folder change registry, your application's `appSymbol` or some variation on it is normally used as this parameter's value.
callBackFn	A function object that is executed when a folder changes. The function must be of the form `func(`*oldFolder*`,`*newFolder*`);`

Its parameters are

oldFolder	A string that is the name of the folder that changed.
newFolder	A string that is the new name of the folder specified by the *oldFolder* parameter. The value of this parameter is `nil` if the *oldFolder* folder was deleted.

The value returned by the *callBackFn* function is ignored.

UnRegFolderChanged

`UnRegFolderChanged(`*callbackID*`)`

Unregisters the specified callback function from the folder change notification mechanism. The value returned by this function is unspecified.

callbackID	A unique symbol identifying the closure to be unregistered. This symbol was passed to the `RegFolderChanged` function to register this callback function with the folder change notification mechanism. Normally, the value of this parameter is the application symbol or some variation on it.

AddFolder

`AddFolder(`*newFolderStr*`, `*appSymbol*`)`

Creates a local folder having the specified name for the specified application, transmits a folder change notification, and returns the tag that represents the new folder. If a folder having the specified name already exists, this function returns that folder's tag without creating a new folder.

This function returns `nil` without creating a new folder when the addition of another folder would exceed the number of unique folders allowed by the system. Version 2.0 of the Newton operating system allows twelve global folders system-wide and twelve local folders per application. Only the user can create global folders; however, applications can use the `AddFolder` function to create local folders.

Note that the symbol this function returns may differ from the one that the `Intern` global function would create from the *newFolderStr* string. In particular, note that the `AddFolder` function accepts non-ASCII string values, while the `Intern` function does not.

newFolderStr — String displayed to the user as the name of the new folder.

appSymbol — Symbol of the application to which the new folder is local.

RemoveFolder

`RemoveFolder(`*folderSym*`,` *appSymbol*`)`

Removes the specified folder from the specified application's list of local folders. Items formerly filed in the removed folder are marked as unfiled. Such items may be viewed by selecting "All Items" or "Unfiled Items" from the folder list displayed by a `protoNewFolderTab` or `protoClockFolderTab` view.

The return value of this function is unspecified; do not rely on it.

folderSym — Symbol identifying the folder to delete

appSymbol — Symbol of the application to which the removed folder is local.

GetFolderStr

`GetFolderStr(`*folderSym*`)`

Returns the user-visible string associated with the specified symbol. Returns `nil` when passed a symbol not associated with a folder or a symbol that is not found. Returns the string `"Unfiled"` when passed `nil` as its argument.

folderSym — The symbol for which this function returns a folder name string.

RemoveAppFolders

`RemoveAppFolders(`*appSym*`)`

Removes all folders local to the specified application. Any folder used by an application other than the specified application is untouched. Items filed in the removed folders are subsequently considered unfiled; however, no change notification message is broadcast because the change presumably affects only the caller of this function. Unless your application uses global folders only, you normally call this function from your application's `DeletionScript` method. (The `DeletionScript` method is invoked when the application package is scrubbed from the Extras Drawer; for more information, see the description of this method in Chapter 2, "Getting Started,"in *Newton Programmer's Guide*.)

GetFolderList

`GetFolderList(`*appSymbol*`,` *localOnly*`)`

Returns an array of symbols representing the folders available for use by the specified application; the symbols are ordered according to an alphabetic sort of the user-visible folder strings associated with them. The *localOnly* parameter can be used to specify whether this function includes global folders in its result.

appSymbol — Symbol identifying the application for which this function returns local folders.

localOnly — Set to `true` to specify that this function not return the symbols of global folders.

RenameFolder

`RenameFolder(`*folderSym*`,`*newFolderStr*`)`

Generates a new folder symbol from the specified string, associates this string with the specified folder, and notifies applications of the change. Returns the new folder symbol if successful; otherwise, returns the value `nil`.

folderSym — Symbol identifying the folder to rename.

newFolderStr String specifying the folder's new user-visible name.

Application-Defined Filing Methods

You must provide these methods to support the Filing service.

FileThis

targetView`:FileThis(`*target*`,` *labelsChanged*`,` *newLabels*`,` *storesChanged*`,` *newStore*`)`

This developer-defined method must do everything required to file the current data item. This message is sent to the view specified by the `GetTargetInfo` method when an item is filed or moved to another store by the Filing service.

target The item(s) to be filed, as specified by your application's `GetTargetInfo` method.

labelsChanged This value is non-`nil` when the target's filing category has changed. When the value of this parameter is `nil`, the value of the *newLabels* parameter is undefined.

newLabels When the value of the *labelsChanged* parameter is non-`nil`, this argument is the symbol that is the new value of the target's `labels` slot. When the value of the *labelsChanged* parameter is `nil`, the value of the *newLabels* parameter is undefined.

storesChanged This value is non-`nil` when the store specified for filing the target has changed. When the value of this parameter is `nil`, the value of the *newStore* parameter is undefined.

newStore The new store only when the value of the *storesChanged* parameter is non-`nil`; otherwise, this value is undefined.

IMPORTANT

If you support `FileThis` you are responsible for performing all tasks necessary to file the entry. That is, you must change the value of its `labels` slot and move the entry to the new store as necessary. The Filing service does not handle these changes for you. ▲

NewFilingFilter

targetView`:NewFilingFilter(`*newFilterPath*`)`

The system sends the `NewFilingFilter` message to the target view when the user picks a new category of items in a folder tab. This developer-supplied method must perform any actions necessary to display items in the filing category specified by the `labelsFilter` and `storesFilter` slots. Typically, this method queries the application's soups for items in the new filing category and then redraws views affected by the change in the filing filter.

The value of the *newFilterPath* parameter specifies which of the `storesFilter` or `labelsFilter` slots changed, but does not provide the new value of the specified slot. Your implementation of this method must test the value of the appropriate slot for use in the construction of a query spec.

This method replaces the `FilterChanged` method. If the `NewFilingFilter` method is defined, the `FilterChanged` message is not sent at all. If the `NewFilingFilter` method is not defined, the `FilterChanged` message is sent to the target view. The system uses proto and parent inheritance to find your implementation of the `NewFilingFilter` method.

newFilterPath The filter path that changed, as specified by either the `'storesChanged` or `'labelsChanged` symbol.

Find Reference

This describes system prototypes (protos), functions, methods, and data structures used by the Find service.

Finder Protos

The system supplies two finder protos on which to base your application's finders.

ROM_SoupFinder

System-supplied prototype that supports the enumeration of found items in soup-based data. If your application stores its data in soups, base the finder frame resulting from your search method on the `ROM_SoupFinder` proto.

Your finder frame, based on this proto, must contain the slots described in this section. The slots in the frame returned by a date find are the same as those in the returned finder frame for a text find. You can also add your own slots to this frame; the Find service ignores them.

Slot description

`owner`	Required. Set to a view that receives the `ShowFoundItem` message (usually your application's base view).
`cursor`	Required. The cursor returned by your search method's query.
`title`	Required. A string that is your application's user-visible name. The system uses this string in the Find overview to separate matches found in each application when conducting global or selected finds. You can omit this slot if the frame referenced by `owner` has a `title` slot.
`findType`	Required. Specifies whether the search is for text or date. The value of this slot is always one of these symbols: `'text`, `'dateBefore`, `'dateOn`, or `'dateAfter`.
`findWords`	Required. A n array of strings that specify the text to match or the date to compare.

The following slot is used by the system:

`selected`	An array of currently-selected items. The format of this array is not documented. You may determine the number of selected items in it by passing this array to the `Length` function.

The following methods are defined in the `ROM_SoupFinder` proto.

Count

soupFinder`:Count()`

Returns an integer value representing the total number of found items.

Note

Do not override this method. ◆

Delete

soupFinder:`Delete()`

Deletes all currently selected items from writeable stores.

If you override this method, items can still be deleted and the crumple effect still occurs, even if your override method does not call the inherited method.

FileAndMove

soupFinder:`FileAndMove(`*labelsChanged*`,` *newLabel*`,` *storeChanged*`,` *newStore*`)`

Files and/or moves the selected items.

labelsChanged	When this parameter is `true`, it signals that a new label is being assigned.
newLabel	The new value for the `label` slot when the *labelsChanged* parameter has the value `true`. This value is undefined when the value of the *labelsChanged* parameter is not `true`.
storeChanged	When `true`, a new store is being assigned.
newStore	The new store when the *storeChanged* parameter has the value `true`. This value is undefined when the value of the *storeChanged* parameter is not `true`.

You can override this method to perform additional application-specific tasks; however, it is suggested that your version of this method call the inherited method to actually file or move items. Note that the `FileAndMove` message may be sent when no items are selected; thus, your override method must check whether any items are selected before doing any work.

ForEachSelected

soupFinder:`ForEachSelected(`*callbackFunction*`)`

Calls the callback function with each of the currently selected items as a parameter.

callbackFunction A function object you supply. This function must accept one argument that is a soup entry, as in:

```
func(soupEntry) begin ... end;
```

Note
Do not override this method. ◆

GetTarget

soupFinder:`GetTarget()`

Returns a cursor for use by routing. You may override this method.

IsSelected

soupFinder:`IsSelected(`*item*`)`

Returns `true` if the specified item is selected in the Find overview.

item The found item to test.

Note
Do not override this method. ◆

Reset

soupFinder:`Reset()`
Resets a soup finder's cursor to the first found entry. This method performs none of the housekeeping tasks that the `ReSync` method does. In general, you should use the `ReSync` method for resetting a soup finder.

Note
Do not override this method. ◆

ReSync

soupFinder`:ReSync()`

Returns the finder to its initial state and resets the cursor to the first entry in the set of found items. Generally, you should use this method to reset a soup finder, rather than using the `Reset` method, which only resets the soup finder's cursor.

Call this method when disposing of the Find overview or when the user changes items in the Find overview, to update and redisplay the overview. For example, you need to call this method when moving or deleting an item from the overview. You can also use this method to recover from errors encountered when the attempt to display an item fails, such as when advancing the cursor to an item returns `nil` or the `'deleted` symbol.

Note

Do not override this method. ◆

SelectItem

soupFinder`:SelectItem(`*item*`)`

Marks the specified item as selected in the Find overview.

Your soup finder can replace this method with a slot containing the value `nil` to suppress the display of checkboxes in the Find overview.

item	The found item to mark as selected.

ShowFoundItem

soupFinder`:ShowFoundItem(`*item*, *finder*`)`

Displays the overview item passed to it as a parameter.

item	An item returned as a result of a Find operation.
finder	A frame that enumerates the items found via the Find slip. Usually finder frames are based on the system protos `ROM_SoupFinder` or `ROM_CompatibleFinder`.

ShowEntry

soupFinder:`ShowEntry(`*entry*`)`

Causes the finding application to display the specified entry, opening the application if necessary. This method does not close the Find overview.

entry The soup entry to display.

Note
Do not override this method. ◆

ShowOrdinalItem

soupFinder:`ShowOrdinalItem(`*ordinal*`)`

Shows an entry that is specified by an ordinal integer or symbol; it can be used to scroll items in the Find overview. For example, to scroll to the next item in the overview, you may increment or decrement the `currentItem` index appropriately, call the appropriate cursor function to set the current item to the new index, and then redisplay the overview.

ordinal One of the symbols `'first`, `'prev`, or `'next`, or an integer; it is used to call the appropriate cursor method to retrieve the specified entry.

For more information on cursor methods see “Data Storage and Retrieval” (page 11-1) in *Newton Programmer’s Guide*

ZeroOneOrMore

soupFinder:`ZeroOneOrMore()`

Returns 0 if no entries were found, 1 if one entry was found, or another number if more than one entry was found.

Note
Do not override this method. ◆

ROM_CompatibleFinder

This system-supplied prototype supports finder frames for data that is not stored in soups. If your application stores its data in arrays, for instance, you should base your result frame on the `ROM_CompatibleFinder` proto. The finder frame resulting from your searches must contain the slots described here. You can also add your own slots to this frame; the Find service ignores them.

Slot description

`owner` — Required. The view that receives the `ShowFoundItem` message; usually your application's base view.

`title` — Required. A string that is your application's user-visible name. In the Find overview, this string groups items returned by applications that participated in a global or selected find. You can omit this slot if the frame referenced by `owner` has a `title` slot.

`findType` — Required. Specifies whether the search is for text or date. The value of this slot is always one of the symbols `'text`, `'dateBefore`, `'dateOn`, or `'dateAfter`.

`findWords` — Required. A n array of strings that specify the text to match or the date to compare.

`items` — Required. An array of found items returned by your search method.

Each frame in the `items` array must contain these slots:

`_proto` — Optional. However, it is recommended that you reference the data item rather than using the data item directly, because global searches alter the `items` frame destructively. Referencing the application data frame through the `_proto` slot ensures that the original data remains intact.

`title` — Required. The string that represents this item in the Find overview.

Your `items` frame should look like the following code sample, which depicts the frame returned by a search that found two items, currently stored in an array named `foundItems`:

```
items := [
            { // protect data by putting in proto chain
            _proto:foundItems[0],

            // string displayed in the Find overview
            title:"First",

            // create as many slots as needed for
            // application-specific information
            slotName: "some more data"
            },

            {_proto: foundItems[1],
             title: "Second",
             slotName: "some more data"
            }
         ];
```

IMPORTANT

Global searches destructively alter the `items` frame. Because your application's `Find` method (the same one used for local searches) is called by the system when the user requests a global search, each element of your `items` array should use a `_proto` slot to reference the data found in the search, rather than accessing the data directly. ▲

The following slot is used by the system:

`selected` An array of currently-selected items. The format of this array is not documented. You may determine the number of selected items in it by passing this array to the `Length` function.

The following methods are defined in the `ROM_CompatibleFinder` proto.

ConvertToSoupEntry

compatibleFinder`:ConvertToSoupEntry(`*item*`)`

Returns a soup entry corresponding to the specified item from the found items array.

item An element of the `items` array in the finder frame.

Count

compatibleFinder`:Count()`

Returns an integer value representing the total number of found items.

Delete

compatibleFinder`:Delete()`

Deletes all currently selected items from writeable stores.

You should override this method if your data is stored in anything other than a single soup. If you do not want to override this method, you should considering not allowing the checkbox to appear by your found items. This is done by including a `SelectItem` slot set to `nil` in your finder frame.

If you override this method, items can still be deleted and the crumple effect still happens, even if your override method does not call the inherited method.

FileAndMove

compatibleFinder`:FileAndMove(`*labelsChanged*`,` *newLabel*`,` *storeChanged*`,` *newStore*`)`

Files and/or moves the selected items.

You should override this method if your data is stored in anything other than a single soup. If you do not want to override this method, you should considering not allowing the checkbox to appear by your found items. This is done by including a `SelectItem` slot set to `nil` in your finder frame.

labelsChanged	When this parameter is `true`, it signals that a new label is being assigned.
newLabel	The new value for the `label` slot when the *labelsChanged* parameter has the value `true`. This value is undefined when the value of the *labelsChanged* parameter is not `true`.
storeChanged	When `true`, a new store is being assigned.
newStore	The new store when the *storeChanged* parameter has the value `true`. This value is undefined when the value of the *storeChanged* parameter is not `true`.

You can override this method to perform additional application-specific tasks; however, it is suggested that your version of this method call the inherited method to actually file or move items. Note that the `FileAndMove` message may be sent when no items are selected; thus, your override method must check whether any items are selected before doing any work.

ForEachSelected

compatibleFinder`:ForEachSelected(`*callbackFunction*`)`

Calls the callback function with each of the currently selected items as a parameter. Note that for a compatible finder, you must override this method since the callback function expects a soup entry as a parameter.

callbackFunction	A function object you supply. This function must accept one argument that is a soup entry.

GetTarget

compatibleFinder`:GetTarget()`

Returns a cursor for use by routing. You may override this method.

IsSelected

compatibleFinder:`IsSelected(`*item*`)`

Returns `true` if the specified item is selected in the Find overview.

item The found item to test.

ReSync

compatibleFinder:`ReSync()`

Resets the finder to its initial state. Call this method when disposing of the Find overview or when the user changes items in the Find overview, to update and redisplay the overview. For example, you need to call this method when moving or deleting an item from the overview.

SelectItem

compatibleFinder:`SelectItem(`*item*`)`

Marks the specified item as selected in the Find overview.

Your soup finder can replace this method with a slot containing the value `nil` to suppress the display of checkboxes in the Find overview. If you store your data in something other than a single soup, you must either disable the checkbox or override the Routing methods in your finder.

item The found item to mark as selected.

ShowFakeEntry

compatibleFinder:`ShowFakeEntry(`*index*`)`

You should override this method to show the found item referenced by the integer value *index*. This method should open your application and send it a `ShowFoundItem` message.

index An integer index that references an item returned as a result of a Find operation.

System Functions and Methods

The following functions and methods are supplied by the system.

RegFindApps

`RegFindApps(`*appSymbol*`)`

Registers an application for Global finds; that is, after the `RegFindApps` function executes, the Find service sends messages to the `GetRoot().(`*appSymbol*`)` view.

appSymbol The application symbol for the application that you want to register for global finds.

Note
To ensure your application's compatibility with future versions of Newton System Software, use this function to register for global and selected finds. Applications running on older Newton devices can use the `kRegFindAppsFunc` function provided by NTK for this purpose. ◆

UnRegFindApps

`UnRegFindApps(`*appSymbol*`)`

Unregisters an application for global finds; that is, after the `UnRegFindApps` function executes, the system no longer sends `Find` messages to the view `GetRoot().(`*appSymbol*`)` when the user taps the All button in the Find slip.

appSymbol The application symbol for the application that you want to unregister for global finds.

Note

To ensure compatibility with future versions of Newton System Software, use this function to unregister for Global and Selected finds. Applications running on older Newton devices can use the `kUnregFindAppsFunc` function provided by NTK for this purpose. ◆

SetMessage

statusView`:SetMessage(`*message*`);`

Displays the *message* string in the Find Progress slip. The *message* string should be similar to those displayed by the built-in applications, that is `"Searching in` *applicationName*`..."`.

statusView	The Find Progress slip. A reference to this view is passed to your search method in the *statusView* parameter.
message	The message string to display.

StandardFind

view`:StandardFind(`*what*`,`*soupName*`,`*results*`,`*statusView*`,`*indexPath*`)`

Uses a finder frame based on the `ROM_SoupFinder` proto to search for strings beginning with the specified text. This method reports status to the user and appends the finder frame resulting from the search to the system-supplied `results` frame.

what	The user-specified string for which this method is to search your application's data.
soupName	A string that is the name of your application's data soup. `StandardFind` uses this name to call `GetUnionSoup` for you.
results	The system-generated `results` array, passed to the `StandardFind` method by the system. The `StandardFind` method appends the finder frame resulting from your search to this array. The content of

your finder frame depends on the kind of finder proto used. If a global find is in progress, the `results` array may contain slots created by other applications' search methods.

statusView The frame to which the `SetMessage` message should be sent to.

indexPath The index path used in the query that this method makes against your application's soup data. Pass `nil` for this value if you don't want to sort the entries in the cursor on this value. For more information, see "Data Storage and Retrieval" (page 11-1) in *Newton Programmer's Guide*.

You must call the `GetUnionSoupAlways` function, saving the result, before calling StandardFind. The following example illustrates the use of this method in an application's `Find` method:

```
MyApplicationBase.Find :=
func(what, results, scope, statusView)
begin
   local temp := GetUnionSoupAlways (kMySoupName);
   :StandardFind(what, kMySoupName, results,
                 statusView, nil);
end;
```

Application-Defined Methods

The following methods should be included in your application's base view. you must supply at least a `ShowFoundItem` and either `Find` or `DateFind`. If you are using the `ROM_SoupFinder` you must also supply a `FindSoupExerpt` method. If you wish to support targeted finds, you must also supply an `AppFindTargets` and the targeted version of your search method (`FindTargeted` or `DateFindTargeted`). Supply a `CustomFind` method if you wish to override the system-supplied find slip when your application is frontmost.

AppFindTargets

myAppBase.`AppFindTargets()`

Returns an array of frames of the form:

`{name: "`*userVisibleText*`", target:` *thisDataForYourUse*`},`

Each frame in this array represents an item that is displayed in the view which allows the user to pick applications for a Selected find. The string in the `name` slot is shown as thought it were an application, allowing your application to search different data sets independently. The object in the `target` slot is entirely for your use. This object will be passed to your `FindTargeted` (or `DateFindTargeted`) method as a parameter.

DateFind

myAppBase.`DateFind(`*findTime*, *compareHow*, *results*, *scope*, *statusView*`)`

Appends a frame containing entries that meet the specified date comparison criteria to the system supplied results array, which is passed in as the *results* argument. If you wish to support text finds you must also supply a `Find` method.

The return value of this method is ignored.

findTime — Specifies the date selected by the user. The date is represented as an integer that is the number of minutes passed since midnight, January 1, 1904.

compareHow — Specifies whether the user chose to find items before, on, or after the date specified by the value of the *findTime* parameter. The value of the *compareHow* parameter is always one of the symbols `'dateBefore`, `'dateOn`, or `'dateAfter`.

results — An array of frames passed to your `DateFind` method by the system; your `DateFind` method appends a finder frame to this array. The content of your finder frame depends on the kind of finder proto used to create the frame returned by your search method. If you used the `ROM_SoupFinder` proto, the frame contains a cursor. If you used the `ROM_CompatibleFinder` proto, the frame contains an array of found items. If a global find is in progress, the *results* array may contain slots created by other applications' search methods.

scope — Either `'localFind` or `'globalFind`. The value of this parameter indicates whether the search is local or global, allowing you to handle these two cases differently, if you prefer.

statusView — A frame to which you send the message `SetMessage`. The `SetMessage` method accepts as its sole argument a string to display to the user while the search is in progress.

DateFindTargeted

myAppBase`.DateFindTargeted(`*findTime*, *compareHow*, *results*, *scope*, *statusView*, *target*`)`

Finds data in a particular data set, and appends a frame containing entries that meet the specified date comparison criteria to the system supplied results array, which is passed in as the *results* argument. The particular data set to search is specified by the *target* parameter, which is the object your `AppFindTargets` returned in the `target` slot. If you supply this method you must define a `DateFind` method.

The return value of this method is ignored.

findTime — Specifies the date selected by the user. The date is represented as an integer that is the number of minutes passed since midnight, January 1, 1904.

compareHow — Specifies whether the user chose to find items before, on, or after the date specified by the value of the *findTime* parameter. The value of the *compareHow*

	parameter is always one of the symbols `'dateBefore`, `'dateOn`, or `'dateAfter`.
results	An array of frames passed to your `Find` method by the system; your `Find` method appends a finder frame to this array. The content of your finder frame depends on the kind of finder proto used to create the frame returned by your search method. If you used the `ROM_SoupFinder` proto, the frame contains a cursor. If you used the `ROM_CompatibleFinder` proto, the frame contains an array of found items. If a global find is in progress, the *results* array may contain slots created by other applications' search methods.
scope	Always the symbol `'globalFind`.
statusView	A frame to which you send the message `SetMessage`. The `SetMessage` method accepts as its sole argument a string to display to the user while the search is in progress.
target	The object your `AppFindTargets` returned in the `target` slot.

Find

myAppBase`.Find(`*what*`,`*results*`,`*scope*`,`*statusView*`)`

Appends a frame containing instances of the specified string beginning to the array passed in the *results* argument.

The system supplies the global function, `StandardFind`, that you can use to implement your application's `Find` method for soup-based text data. If you want to support date finds, you must implement your application's `DateFind` method yourself.

The return value of this method is ignored.

what Contains the user-specified string for which to search your application's data.

results An array of frames passed to your `Find` method by the system; your `Find` method appends a finder frame to this array. The content of your finder frame depends on the kind of finder proto used to create the frame returned by your search method. If you used the `ROM_SoupFinder` proto, the frame contains a cursor. If you used the `ROM_CompatibleFinder` proto, the frame contains an array of found items. If a global find is in progress, the *results* array may contain slots created by other applications' search methods.

scope Either `'localFind` or `'globalFind`. The value of this parameter indicates whether the search is local or global, allowing you to handle these two cases differently, if you prefer.

statusView A frame to which you send the message `SetMessage`. The `SetMessage` method accepts as its sole argument a string to display to the user while the search is in progress.

FindTargeted

myAppBase.`FindTargeted(`*what*, *results*, *scope*, *statusView*, *target*`)`

Finds text data in a particular data set, and appends a frame containing entries that meet the specified date comparison criteria to the system supplied results array, which is passed in as the *results* argument. The particular data set to search is specified by the *target* parameter, which is the object your `AppFindTargets` returned in the `target` slot. If you supply this method you must define a `Find` method.

The return value of this method is ignored.

what — Contains the user-specified string for which this method is to search your application's data.

results — An array of frames passed to your `Find` method by the system; your `Find` method appends a finder frame to this array. The content of your finder frame depends on the kind of finder proto used to create the frame returned by your search method. If you used the `ROM_SoupFinder` proto, the frame contains a cursor. If you used the `ROM_CompatibleFinder` proto, the frame contains an array of found items. If a global find is in progress, the *results* array may contain slots created by other applications' search methods.

scope — Always the symbol `'globalFind`.

statusView — A frame to which you send the message `SetMessage`. The `SetMessage` method accepts as its sole argument a string to display to the user while the search is in progress.

target — The object your `AppFindTargets` returned in the `target` slot.

FindSoupExcerpt

ownerView`.FindSoupExcerpt(`*entry*`,`*finderFrame*`) // for ROM_SoupFinder`

Extracts the name of a specified item from the result frame and returns it as a string. The system displays this string to identify the item in the Find overview. If no items are found, the `FindSoupExcerpt` message is not sent.

ownerView — The view specified by the `owner` slot in the result frame returned by the search method, usually your application's base view. For more information, see the

	section “Returning Search Results” (page 16-21) in *Newton Programmer’s Guide*.
entry	Soup entry whose title is needed.
finderFrame	The finder frame your application added to the system’s `results` array.

ShowFoundItem

ownerView.`ShowFoundItem(`*foundItem*`,`*finderFrame*`)`

Locates the specified item in your application’s data and displays it, performing any scrolling or highlighting that is appropriate. A typical `ShowFoundItem` method may need to do the following:

- open a view appropriate for displaying the target
- set the cursor or the `target` slot to reference the target
- scroll the contents of the display view to make the target visible
- highlight the target in the display view

ownerView	The view specified by the `owner` slot in the result frame returned by the search method, usually your application’s base view. For more information, see the section “Returning Search Results” (page 16-21) in *Newton Programmer’s Guide*.
foundItem	Found item to display. This is a soup entry if using `ROM_SoupFinder`, and an element of the `items` array if using `ROM_CompatibleFinder`.
finderFrame	Finder frame your application added to the system’s `results` array.

CustomFind

myAppBase.`CustomFind()`

Application-defined method that opens your own customized Find slip and does anything else required to implement a customized search and display its results.

System Services Reference

This chapter describes functions, methods, and data structures that support the system services described in Chapter 17, “Additional System Services,” in *Newton Programmer’s Guide*. Items are grouped according to the system service they support; for example, all functions, methods, and data structures pertaining to the Undo service are described in the “Undo Reference” section.

Undo Reference

This section describes functions and methods your application can use to provide Undo/Redo behavior.

AddUndoCall

`AddUndoCall(`*callBackFn*`,` *argArray*`)`

Registers a function object to be called unconditionally when the user taps Undo. The return value of this function is unspecified—do not rely on it.

callBackFn A function object that performs the undo operation *of the form:*

```
func ( args )
begin
//do something
end;
```

argArray Array of arguments to pass to the function object specified by the function parameter.

AddUndoSend

`AddUndoSend(`*receiver*`, `*message*`, `*argArray*`)`

Registers a message and arguments to be sent to a specified receiver unconditionally when the user taps Undo. The return value of this function is unspecified—do not rely on it.

receiver Frame to which the specified message is sent.

message Symbol that is the message to send.

argArray Array of arguments to pass with the message.

AddUndoAction

view`:AddUndoAction(`*methodName*`, `*argArray*`)`

Registers an undo action for the specified view with the system.

view View for which this method is registering an undo action.

methodName A symbol (it must be preceded by a single quotation mark) that is the name of the method to be called when the user taps the Undo button. This method must always return `true`.

argArray An array of parameters to be passed to the method specified by the *methodName* parameter.

ClearUndoStacks

`ClearUndoStacks()`

Removes all pending undo actions from the system, including those destined for other applications; use this function sparingly and cautiously. It is recommended that applications call this method from their `ViewQuitScript` method, but only if they have previously called the `AddUndoAction` function.

IMPORTANT
Do not call this function from the application's `RemoveScript` function. ▲

Idler Reference

This section describes functions and methods you can use to perform periodic tasks.

SetupIdle

view`:SetupIdle(`*milliseconds*`)`

Installs or changes an idler object for the specified view. (An idler object calls the specified view's `ViewIdleScript` method periodically.) The `SetupIdle` method always returns `nil`.

view	The view to which an idler object is to be installed.
milliseconds	The number of milliseconds to wait before calling the `ViewIdleScript` method for the first time. After the first time, the view's `ViewIdleScript` method returns an integer which is the delay until this method is next called. This number should be no less than 100.

You can call the `SetupIdle` method at any time to reset the idle time immediately.

To remove the idler object, call this method again, passing `0` as the value of the *milliseconds* parameter. You can also remove the idler by returning `nil` from the view's `ViewIdleScript` method. The idler object is removed in any case when the view is closed.

Note

When you install an idler for a view, the time the `ViewIdleScript` message is sent next is not guaranteed to be the exact interval you specify. This is because the idler may be delayed if a method is executing when the interval expires. The `ViewIdleScript` message cannot be sent until an executing method returns. ◆

Note

The `clParagraphView` class internally uses the idle event mechanism to implement some of its features. Unfortunately, any `ViewIdleScript` methods provided by developers also execute when the system idle events are processed. Only the "heavyweight" views do this, "lightweight" paragraph views (in other words, simple static text views) do not.

There is no workaround available in the Newton 1.x OS or Newton 2.0 OS. You can either accept the extra idle script calls, or use some other non-`clParagraphView` based view to implement your idle functions. ◆

Alerts and Alarms Reference

This section describes the alert and alarm functions, and the proto used to edit periodic alarms.

Alerts and Alarms Proto

This section describes the proto used to edit periodic alarms.

protoPeriodicAlarmEditor

Provides a view used to schedule periodic (repeating) alarms. You must use this proto to allow your application's user to set the periodic alarms.

Slot descriptions

`title`	A string displayed at the top of the view.
`ownerSymbol`	Your application symbol. The alarm messages are sent to this frame.
`ownerApp`	A string that is your application's name shown in the user interface.
`viewBounds`	The bounds frame for the view. Do not change the size of the view from what is defined in ROM, though different ROMs may have different `viewBounds` defined. You may change its position on the screen while leaving the size constant. Use `LocalBox` to check the size of the proto.

IMPORTANT

Do not add child views to any views which proto from `protoPeriodicAlarmEditor`. ▲

The following application-defined methods must be included in your application's base view: `AlarmsEnabled` and `PeriodicAlarm`. They are described in the following subsections.

AlarmsEnabled

myApp`:AlarmsEnabled()`

Return `true` if the `PeriodicAlarm` message should be sent, `nil` otherwise. If the alarms are not a feature that can be disabled, you may define this method simply as

```
func() true;
```

PeriodicAlarm

myApp:PeriodicAlarm(*alarm*)

The method that implements a periodic alarm. This method is invoked when the alarm executes.

alarm A frame with information about the alarm, containing the following slots:

Slot descriptions

owner	The symbol in the ownerSymbol slot of the protoPeriodicAlarmEditor that set this periodic alarm.
ownerName	The string in the ownerApp slot of the protoPeriodicAlarmEditor that set this periodic alarm.
time	The time this alarm executed expressed in the number of minutes since midnight, January 1, 1904.
hours	An integer expressing the hour at which this alarm executed. This number is an integer in the range 0...23.
minutes	An integer expressing the minute at which this alarm executed. This number is an integer in the range 0...59.
name	A string (displayed in the view created from protoPeriodicAlarmEditor) representing the times the alarm is to go execute. The format of this string is: *hour*s:*minutes* am|pm and a designation for the days on which the alarm is set. This day designator can be the strings "Everyday", "Weekdays", or "Weekends" if these labels apply; otherwise it is either the first three letters of the unique day for which it is set, or the first letter of each of the multiple days for which it's set.

Note
This frame may contain additional slots. Do not rely on the value of any undocumented slots. ◆

Alerts and Alarm Functions

This section describes in detail the alarm and notification functions.

Notify

view:`Notify(`*level*`,` *headerStr*`,` *messageStr*`)`

Uses the system notification facility to display a message or otherwise notify the user.

level — Specifies the notification level to use; it can be one of the following constants:

`kNotifyLog`
: The notice is only entered into the notification log; the user is not alerted.

`kNotifyMessage`
: The user is alerted by blinking the notify icon that a message is pending. Tapping the icon causes pending messages to be displayed in an alert view.

`kNotifyAlert`
: The notice is immediately displayed to the user in an alert view and the system beep is played.

`kNotifyQAlert`
: The notice is immediately displayed to the user in an alert view.

headerStr — A string displayed as a title on the notification slip. Usually this is the name of your application or a major component of it.

messageStr — A string that is the message to the user.

AddAlarm

`AddAlarm(`*alarmKey*`,` *timeSpec*`,` *argsArray*`,` *cbFn*`,` *cbParms*`)`

Registers an alarm to execute at a specified time and returns its alarm key. When the alarm executes, it wakes the Newton if necessary, and displays a specified notification message. You can take additional action by specifying a callback function and its arguments.

alarmKey A string that uniquely identifies the alarm; only the first 24 characters of an alarm key are significant. Use your developer signature or application symbol as a suffix to ensure the uniqueness of this string; for example, `"wakeUp2:llamaApp:NewtDTS"` specifies the `wakeUp2` alarm set by the `llamaApp` application from the developer `NewtDTS`. If an alarm having the specified key already exists, this function removes it and replaces it with the new alarm.

timeSpec The time the alarm is to execute, specified as either an integer or a date frame. If specified as an integer, the value represents the alarm time in minutes since midnight, January 1, 1904 (similar to the encoding of the value returned by the `Time` function). To specify as a date frame, use the value returned by the `Date` global function.

argsArray An array of either two or three arguments passed to the function that actually displays the notification slip to the user. Two-element arrays `[`*title*`,` *message*`]` are passed to the `AlarmUser` function when the alarm goes off. See the description of the `AlarmUser` function for details. Three-element arrays `[`*level*`,` *title*`,` *message*`]` are passed to the `Notify` function. See the description of the `Notify` function for details.

If the value of *argsArray* is `nil`, the alarm does not call the `Notify` or `AlarmUser` functions when it executes.

cbFn — A function object to be executed when your alarm goes off. Passing `nil` as the value of this argument specifies that no function object is to be executed.

cbParms — Arguments to be passed to *cbFn*. Pass `nil` for this argument if no callback function is being used. If you have a callback function that takes no arguments, pass in the empty array, `[]`, for this parameter.

AddAlarmInSeconds

`AddAlarmInSeconds(`*alarmKey*`,`*timeSpec*`,`*argsArray*`,`*cbFn*`,`*cbParms*`)`

Registers an alarm to execute at a specified time and returns its alarm key. This function is the same as the `AddAlarm` function except that it allows you to specify the alarm's execution time more precisely. See the description of the `AddAlarm` function for additional information.

alarmKey — A string that uniquely identifies the alarm; only the first 24 characters of an alarm key are significant. Use your developer signature or application symbol as a suffix to ensure the uniqueness of this string; for example, `"wakeUp2:llamaApp:NewtDTS"` specifies the `wakeUp2` alarm set by the `llamaApp` application from the developer `NewtDTS`. If an alarm having the specified key already exists, this function removes it and replaces it with the new alarm.

timeSpec — The time the alarm is to execute, specified as either an integer or a date frame. If specified as an integer, the value represents the alarm time in seconds since midnight, January 1, 1993 (similar to the encoding of the value returned by the `TimeInSeconds` function). To specify this value as a date frame, use the value returned by the `Date` global function.

argsArray — An array of either two or three arguments passed to the function that actually displays the notification slip to the user. Two-element arrays `[`*title*`,`*message*`]` are passed to the `AlarmUser` function when the alarm goes off. See the description of the `AlarmUser` function for details. Three-element arrays `[`*level*`,` *title*`,` *message*`]` are passed to the `Notify` function. See the description of the `Notify` function for details.

If the value of *argsArray* is `nil`, the alarm does not call the `Notify` or `AlarmUser` functions when it executes.

cbFn — A function object to be executed when your alarm goes off. Passing `nil` as the value of this argument specifies that no function object is to be executed.

cbParms — Arguments to be passed to *cbFn*. Pass `nil` for this argument if no callback function is being used. If you have a callback function that takes no arguments, pass in the empty array, `[]`, for this parameter.

AlarmUser

`AlarmUser(`*title*`,` *message*`)`

Plays an alarm sound and displays a notification slip with a snooze button; this notification slip is illustrated in Figure 17-2 in *Newton Programmer's Guide*.

Normally, the `AlarmUser` function is called by the `AddAlarm` function rather than the application. The `AlarmUser` function respects the user's settings for the alarm sound and volume when executing the alarm. The return value of this function is unspecified; do not rely on it.

title — The string that is the title of the notification slip this function displays.

message — The string that is the body text of the notification slip this function displays.

RemoveAlarm

`RemoveAlarm(`*alarmKey*`)`

Unschedules an alarm that has not yet executed. This function returns `nil` if it is unable to find an alarm having the specified key. If the alarm is found and removed, this function returns an unspecified non-`nil` value. If you want your application's alarms to execute only when your application is installed, you need to call this function in your application's `RemoveScript` function.

alarmKey	A string that uniquely identifies the alarm; it is passed to the `AddAlarm` function when the alarm is created. See the description of the `AddAlarm` function for more information.

GetAlarm

`GetAlarm(`*alarmKey*`)`

Returns a frame containing information about the alarm associated with the specified key; this frame and its contents must not be modified.

IMPORTANT

Do not modify the frame this function returns. ▲

alarmKey	A string that uniquely identifies the alarm; it is passed to the `AddAlarm` function when the alarm is created. See the description of the `AddAlarm` function for more information.

The alarm frame returned by this function contains the slots described immediately following; do not rely on the values of any other (undocumented) slots that you may find in this frame.

`key`	The alarm key. For more information, see the description of the `AddAlarm` function.
`time`	The time at which the alarm is to execute, expressed as the number of minutes since midnight, January 1, 1904.
`notifyArgs`	Array of arguments to be passed to the `Notify` or `AddAlarm` functions when this alarm executes.
`callBackFn`	Function object specifying a callback function to be executed with this alarm (or `nil`).
`callBackParams`	Array of arguments to this alarm's callback function (or `nil`).

GetAppAlarmKeys

`GetAppAlarmKeys(`*alarmKeySuffix*`)`

Returns an array of all alarm key strings having the specified suffix; if the alarm keys are implemented according to Newton DTS recommendations, this array contains all alarm keys associated with the application using the specified suffix. The returned keys are sorted in execution order, with the key representing the first alarm to execute occupying the first position in the array.

alarmKeySuffix A string used as the suffix in all alarm keys created by a particular application; for example `":AlarmSample1:NewtDTS"`.

RemoveAppAlarms

`RemoveAppAlarms(`*alarmKeySuffix*`)`

Removes all alarms having key strings ending in the specified suffix; if the alarm keys are implemented according to Newton DTS recommendations, this function can be used to remove all alarms created by a particular application. This function returns an integer value specifying the number of alarms it removed. If your application's alarms can't execute meaningfully when the application is not installed, you need to remove them by calling this function from the application's `RemoveScript` function.

alarmKeySuffix A string used as a suffix in all alarm keys created by a particular application; for example `":AlarmSample1:NewtDTS"`.

Progress-Reporting Reference

This section describes the protos and methods used for progress reporting.

Progress-Reporting Proto

This section describes the `protoStatusTemplate`.

protoStatusTemplate

The `protoStatusTemplate` is a configurable status view used to report the progress of lengthy operations to the user. You can use this proto to create views containing animated graphical elements and status messages similar to those used by the built-in applications and the system itself.

Note

Many applications can use the `DoProgress` function to report progress to the user. The `DoProgress` function handles much of the work that you must take care of yourself if creating your own `protoStatusTemplate` view. For a list of criteria to use in making this decision, see “Using DoProgress or Creating Your Own protoStatusTemplate” beginning on page 17-18 in *Newton Programmer’s Guide*. ◆

The `protoStatusTemplate` view is a container view based on `protoFloater` that itself supplies a `protoStatusIcon` view and a `protoStatusCloseBox` view as its view children. The system supplies several special child protos to add graphical elements to this basic container view, which declares itself as the base of this view hierarchy. These child protos are described in the section “Status View Components,” immediately following.

The `protoStatusTemplate` view provides two methods, `ViewSet` and `UpdateIndicator`, that you can use to initialize or update the set of child views displayed by a `protoStatusTemplate` view.

Slot description

`initialSetup` A frame specifying initial values for configuring the status slip and its components. For a complete description this frame, see the description of the `ViewSet` method.

The following subsections discuss the status view components, the built-in status view configurations, and the following methods: `ViewSet`, `UpdateIndicator`, and `CancelRequest`.

Status View Components

Figure 14-1 illustrates the system-supplied protos used to add view components to a `protoStatusTemplate` slip.

Figure 14-1 Status view components

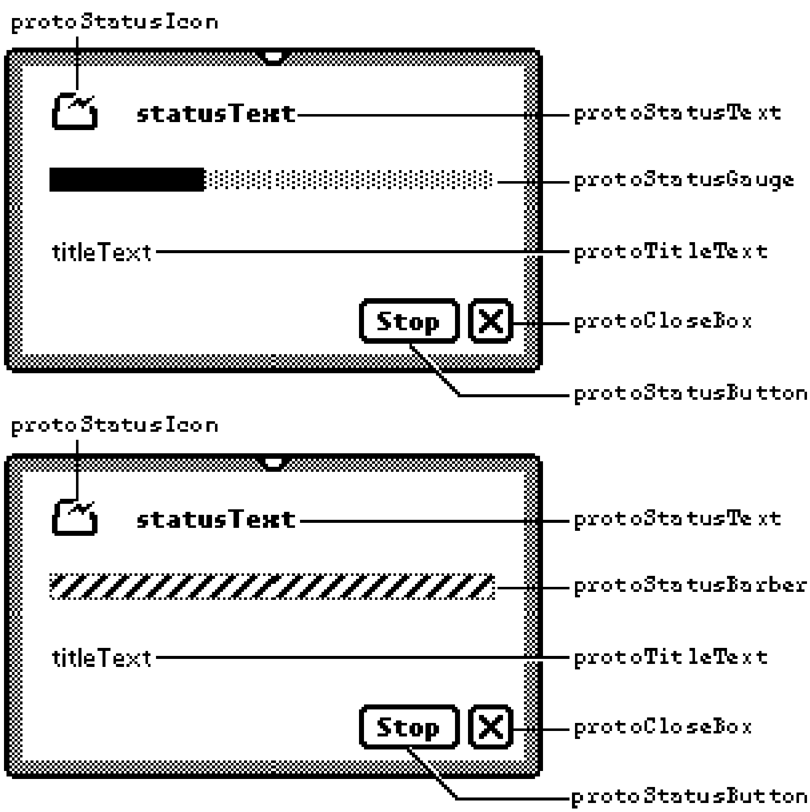


Figure 14-1 Status view components *(continued)*

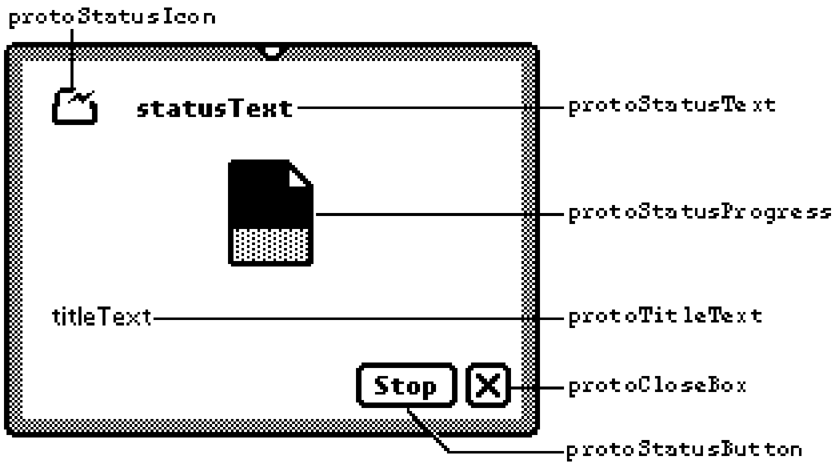


Table 14-1 names the slot each component checks to update its screen display. To update a particular view, include the slot specified in the "Name of slot" column in the `values` slot of the *setup* argument to the `ViewSet` or `UpdateIndicator` methods.

Table 14-1 Status view components

Proto	Name of slot	Description
`protoStatusIcon`	`icon`	An icon.
`protoStatusText`	`statusText`	Text.
`protoTitleText`	`titleText`	Text.
`protoStatusProgress`	`progress`	A thumbnail gauge.
`protoStatusGauge`	`gauge`	A horizontal gauge.

Table 14-1 Status view components (continued)

Proto	Name of slot	Description
`protoStatusButton`	`primary` (in a `vConfirm` view this may be `secondary`)	A button.
`protoStatusCloseBox`	`closeBox`	A close box.
`protoStatusBarber`	`barber`	A horizontal animated barber pole.

The values you supply for the slots specified by Table 14-1 must follow the rules described here for each proto:

`protoStatusIcon`

The value in the `icon` slot must be a bitmap frame, as returned from `GetPictAsBits` function. Save this value in the `icon` slot of your `values` frame.

`protoStatusText`

The value in the `statusText` slot must be a string. Save this value in the `statusText` slot of your `values` frame.

`protoTitleText`

The value in the `titleText` slot must be a string. Save this value in the `titleText` slot of your `values` frame.

`protoStatusProgress`

The value in the `progress` slot must be either a single integer (for example, 50) that reflects the current value of the gauge, or an array of integers giving the current value, minimum and maximum (for example, [50, 0, 100]). By default, the minimum value is 0 and the maximum value is 100. Save this value in the `progress` slot of your `values` frame.

`protoStatusGauge`

The value in the `gauge` slot must be either a single integer (for example, 50) that reflects the current value of the gauge, or an array of integers giving the current value, minimum and maximum (for example, [50, 0, 100]). By default, the minimum value is 0 and the maximum value is 100. Save this value in the `gauge` slot of your `values` frame.

`protoStatusBarber`

Always set the value of the `barber` slot to `true`.

`protoStatusButton`

The value in the `primary` slot must be a frame with a required `text` slot (the button's text) and an optional `script` slot (the button's `ButtonClickScript` method). If only the text slot is included, the default `ButtonClickScript` method calls the status slips, or the application's base view's `CancelRequest` method.

If you specify `nil`, or if you specify a frame and its `text` slot is `nil`, the button is not drawn.

Also, if you include a `ShiftItem` method that returns another view, the button "adjusts" its view if the view returned by `ShiftItem` is not visible.

Save this value in the `primary` slot of your `values` frame, or in the `secondary` slot in `vConfirm` view.

`protoStatusCloseBox`

The value in the `closeBox` slot must be either `nil` or the close box's `ButtonClickScript` method. If `nil`, then the close box is not drawn. Note that the default behavior is `base:Close`. Your `ButtonClickScript` should hide the view and add an action to the notify icon to reopen the status view. This way the user is still made aware that the operation is in progress, and can reopen the status view to cancel the operation.

For more information, see the sections "Notify Icon" beginning on page 17-5 and "Using the Notify Icon" beginning on page 17-15, both in the *Newton Programmer's Guide*.

Save this value in the `closeBox` slot of your `values` frame.

IMPORTANT

The templates of each component view have `viewBounds` and `viewJustify` slots defined. Many of these templates use sibling justification. This can cause views to be drawn improperly if they are defined in a particular order within the `kids` array of the status template. This is because the sibling order is determined by the position of these component views within this `kids` array.

You may override the `viewBounds` and `viewJustify` slots of any component view as necessary, however. Table 14-2 lists the internally defined `viewBounds` and `viewJustify` slots for each component view template. ▲

Table 14-2 Internally defined `viewBounds` and `viewJustify` slots

proto	viewBounds	viewJustify
`protoStatusIcon`	`SetBounds (3,0,35,32)`	None.
`protoStatusText`	`RelBounds (42,4,138,25)`	`vjParentLeftH+ vjParentTopV+ vjTopV+vjLeftH`
`protoTitleText`	`RelBounds (10,6,170,25)`	`vjParentLeftH+ vjSiblingBottomV +vjTopV+vjLeftH`
`protoStatusProgress`	`RelBounds (0,7,32,40)`	`vjParentCenterH+ vjSiblingBottomV`

Table 14-2 Internally defined `viewBounds` and `viewJustify` slots

proto	viewBounds	viewJustify
`protoStatusGauge`	`RelBounds (0,6,166,14)`	`vjParentCenterH+ vjSiblingBottomV`
`protoStatusBarber`	`RelBounds (0,8,166,10)`	`vjParentCenterH+ vjSiblingBottomV`
`protoStatusButton`	`SetBounds (-25 - StdButtonWidth(`*theStringShown*`), -18,-25,-5)`	`vjParentRightH+ vjParentBottomV+ oneLineOnly+ vjCenterV`

Built-in Status View Configurations

There are six built-in configurations of the `protoStatusTemplate`: `vGauge`, `vBarber`, `vStatus`, `vStatusTitle`, `vConfirm`, and `vProgress`. Figure 14-2, "Built-in status view configurations" on page 14-20 shows each type of status view as it appears on a Newton device. The arrows in this graphic point to the name of the slot in the values frames by which you would refer to each particular element in a status view. The values frame is used in the setup parameter to the `ViewSet` and `UpdateIndicator` methods, or in the `InitialSetup` frame. For example, to set the value of the string in a `vStatus` view called `myView`, use the following code:

```
myView:ViewSet({name:'vStatus,
               values:{statusText:theStringToDisplay}
               });
```

To change the value of the string displayed under the gauge in a `vGauge` view called `myOtherView`, use the following code:

```
myOtherView:ViewSet({ name:'vGauge,
                      values:{titleText:theStringToDisplay}
                  });
```

Figure 14-2 Built-in status view configurations

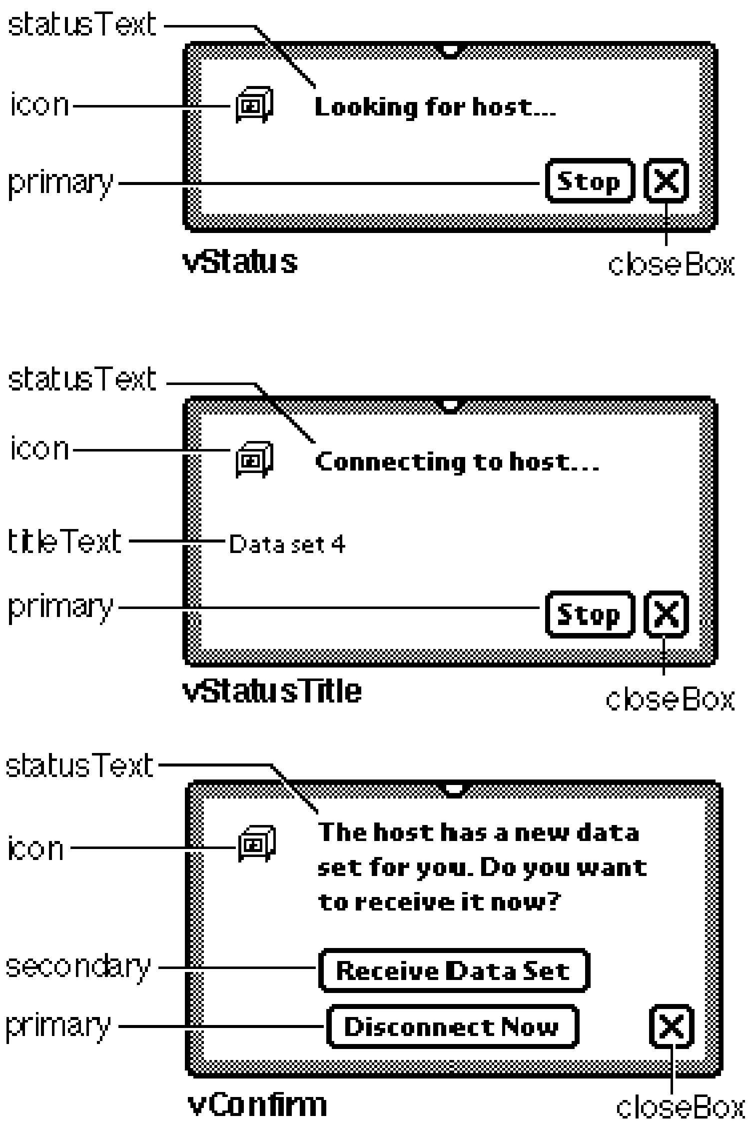


Figure 14-2 Built-in status view configurations *(continued)*

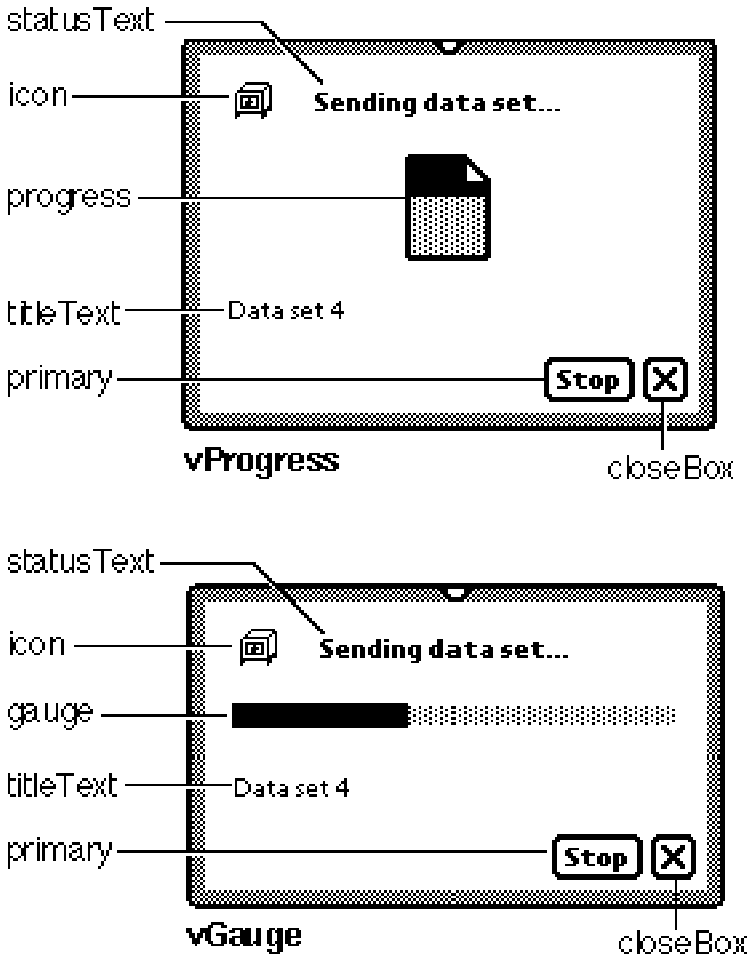
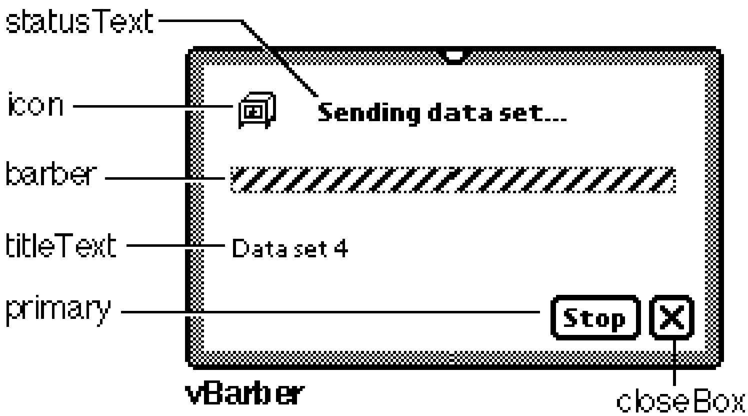


Figure 14-2 Built-in status view configurations *(continued)*



Any element not included in the initial call to `ViewSet` or in the `InitialSetup` frame is not included in the status view. The close box is an exception to this rule, however. It is included unless explicitly omitted (by including a `closebox` slot with the value `nil` in the `values` frame).

ViewSet

statusView`:ViewSet(`*setup*`)`

Initializes or updates status view components and values as specified by the *setup* frame. When this message is sent to a closed status view it must be followed by the `Open` message to display the view. When this message is sent to an open status view, it redraws the view hierarchy in addition to setting up the view children.

When using this method to initialize the status view—in other words, the first time you invoke this method, before actually opening the status view—you must supply all the values the status view requires, including those specifying the components of the view (such as a `vGauge` indicator) and any others that are appropriate (such as the indicator’s position). Once the status view is open, you need only pass those values you need to update, such as the position of the `vGauge` indicator—values that are not changed remain in effect. See also the description of `UpdateIndicator` method on 14-24.

setup — A frame specifying the set of view templates and other values used to instantiate or update the status view. This frame can contain the following slots:

- `appSymbol` — The application symbol of the application displaying the status slip.
- `name` — A symbol specifying the template that provides one or more components of the status view, such as a gauge, title text, message text, an icon, and so on. This symbol can be one of the system-supplied values `'vGauge`, `'vBarber`, `'vStatus`, `'vStatusTitle`, `'vConfirm`, `'vProgress`, or a symbol representing your own template. If you provide your own template, be sure to declare all its component views to the `protoStatusTemplate` view.
- `values` — A frame containing the values to be set or updated in the view component specified by the `name` slot. This frame may contain slots that supply text, an icon, and other configurable elements of the specified view component. This frame should contain a slot for each view component you wish to update. The name of the slot must be one of the names listed under the "Name of slot" column of Table 14-1. The value of this slot should be set as described in the list immediately following Table 14-1.

  For example, Table 14-1 states that the value of a `protoStatusText` view is held in its `statusText` slot. Thus, your `values` frame needs to contain a `statusText` slot. The list following

Table 14-1 states that the value stored in this slot must be a string, so your `values.statusText` slot must contain a string.

IMPORTANT

You cannot change the minimum and maximum values of a gauge by calling `ViewSet` once initial values have been declared. However you may use the following code to convert a three element `[`*minValue*`,` *currValue*`,` *maxValue*`]` array into an integer with the proper gauge setting:

```
theInteger := Floor(((value - minValue)/
(maxValue - minValue)) * (oldMax - oldMin) +
oldMin);
```
▲

UpdateIndicator

statusView`:UpdateIndicator(`*setup*`)`

Updates values and redraws `protoStatusGauge`, `protoStatusBarber`, and `protoStatusProgress` views. Using this method is faster than performing the same action with the `ViewSet` method. Use this method only on views that are already open.

setup See the description of the `ViewSet` method on page 14-22. You need only include in this frame the values that have changed, rather than the entire set-up frame you would pass to the `ViewSet` method.

CancelRequest

myAppBaseOrMyStatusSlip:`CancelRequest(`*why*`)`

Provides an opportunity for you to perform any necessary housekeeping when the user cancels an operation in progress. This method is only called if you do not provide a `ButtonClickScript` (via the script slot) in your `protoStatusButton.`

myAppBaseOrMyStatusSlip
Your status slip or application base view. This method is sent to `your` status slip if it has a `CancelRequest` method, otherwise it is sent to your application base view (or transport object).

why
A symbol specifying why the operation was aborted. If the user tapped a `protoStatusButton` the symbol `'userCancel` is passed. If the operation was cancelled for power-related reasons, the symbols `'powerOff` or `'emergencyPowerOn` might be sent.

Progress-Reporting Functions

This section describes the progress-reporting methods and functions.

DoProgress

`DoProgress(`*kind*`, `*options*`, `*workFunc*`)`

Displays a status slip, calls the function object you pass as one of its arguments, and returns a value indicating how the slip was dismissed. The slip can optionally include a title, message text, and an animated bar gauge or barber pole progress indicator. This method returns the `'cancelled` symbol when the user cancels the operation; otherwise, this method returns `nil`.

kind
The kind of gauge view component to display in the status slip. The `'vGauge` symbol specifies that a horizontal progress gauge is to be displayed. The `'vBarber` symbol specifies that a barber pole gauge is to be displayed. The value `nil` specifies that no gauge is to be displayed.

options A frame specifying optional characteristics of the progress slip. This frame contains the following slots:

`closebox` Required. You must place the value `nil` in this slot to hide the close box normally provided by the status slip.

`gauge` Required when the *kind* parameter has the `'vGauge` value. An integer specifying the percentage of the operation completed.

`barber` Required when the *kind* parameter has the `'vBarber` value. The value `true` specifies that the barber pole gauge is to be animated when the *workFunc* function calls the `SetStatus` method.

`icon` Optional. A bitmap icon displayed in the upper-left corner of the status slip. Typically it identifies the operation (such as Find) or the application displaying the progress slip.

`statusText` Optional. A string displayed at the top of the status slip. It displays the name of the operation in progress or the name of the application that displays the slip. If the slip displays an optional icon, the `statusText` string is displayed to the right of it.

`titleText` Optional. A string displayed at the bottom of the status slip. This string can be used to provide additional information regarding the operation's progress.

workFunc A function object accepting as its sole argument the view that is the status slip. This function object performs the operation on which `DoProgress` reports status. As the operation proceeds, this function updates the progress slip's gauge and title text periodically by calling the `SetStatus` method of the object passed as its argument. For example, the following code fragment does some work and updates the progress gauge and title text with each iteration of the loop:

```
local myOpts :=
        {closebox:nil,
        icon: kMyIcon,
        statusText: kAppName,
        gauge: 10,
        titleText:"One moment, please…"}

        workFunc := func (contextView) begin
           for x := 1 to 10 do begin
              myOpts.gauge := :SomeWork();
              contextView:SetStatus('vGauge,myOpts);
           end; // for loop
        end; // workFunc
```

The following variation displays a barber pole gauge instead of a progress gauge; the only difference is the substitution of the `barber` slot for the `gauge` slot in the frame passed as the second argument to the `SetStatus` method:

```
        func (contextView) begin
           for x := 1 to 10 do begin
              local busyStr := :SomeWork();
              contextView:SetStatus('vGauge,
                             {  titleText:busyStr,
                                barber: True}
           end; // loop
        end; // workFunc
```

The parameters to the `SetStatus` method are the same as the first two parameters to the `DoProgress` function. Any slots specified in options passed to the `SetStatus` method override the original slot values passed to the `DoProgress` function; those that are not specified remain as originally passed to the `DoProgress` function.

Your *workFunc* function must be of the form:

`func (`*contextView*`) begin ... end`

contextView The view that is the status slip containing the gauge, text, and icon displayed by the `DoProgress` method.

SetStatus

contextView`:SetStatus(`*kind*`,` *options*`)`

Updates the status view provided by the `DoProgress` method. The `SetStatus` method must be called from within the work function passed as an argument to the `DoProgress` method. If the user taps the Stop button, the `SetStatus` function throws an exception. It is very important that your own error handling code passes this exception on to the system. For details see "Using the DoProgress Function" beginning on page 17-16 in *Newton Programmer's Guide*.

See also the description of the *workFunc* parameter to the `DoProgress` method, beginning on page 14-25.

kind The kind of gauge view component being displayed in the status slip. The `'vGauge` symbol specifies that a horizontal progress gauge is being displayed. The `'vBarber` symbol specifies that a barber pole gauge is being displayed. The value `nil` specifies that no gauge is being displayed.

options A frame specifying optional characteristics of the progress slip. This frame contains the following slots:

`gauge` Required when the *kind* parameter has the `'vGauge` value. An integer specifying the percentage of the operation completed.

`barber` Required when the *kind* parameter has the `'vBarber` value. The value `true` specifies that the barber pole gauge is to be animated when the *workFunc* function calls the `SetStatus` method.

`icon` Optional. A bitmap icon displayed in the upper-left corner of the status slip. Typically it identifies the operation (such as Find) or the application displaying the progress slip.

`statusText` Optional. A string displayed at the top of the status slip. It displays the name of the operation in progress or the name of the application that displays the slip. If the slip displays an optional icon, the `statusText` string is displayed to the right of it.

`titleText` Optional. A string displayed at the bottom of the status slip. This string can be used to provide additional information regarding the operation's progress.

ShowBusyBox

`ShowBusyBox(`*showIt*`) //platform file function`

Shows or hides the automatic busy cursor.

IMPORTANT

This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

`call kShowBusyBoxFunc with (`*showIt*`);` ▲

showIt A Boolean that specifies whether to show or hide the automatic busy cursor. Specify `true` to show the busy icon until control returns to the system. Specify `nil` to hide the busy icon for the rest of the current iteration of the system event loop.

The return value of this function is undefined; do not rely on it.

AddAction

notifyIcon`:AddAction(`*title*`, `*cbFn*`, `*args*`)`

Registers the specified callback function with the notify icon, adds a text item to the notify icon's menu and returns an object representing the callback function that was added. Your application should save this object to pass to the `KillAction` method.

If no actions were present when the `AddAction` method is called, the notify icon appears. If the menu is displayed when this method is called, its behavior is undefined. (Currently this function closes the menu but you must not rely on this behavior.)

notifyIcon The notify icon view. You can get a reference to this view by using code similar to the following example:

```
local icon := GetRoot().notifyIcon
```

title String that appears in the notify icon's pop-up menu.

cbFn Function object to be executed when the user chooses the *title* item from the notify icon's menu.

args Array of arguments to the *cbFn* function. Pass `nil` for this value if the *cbFn* function accepts no arguments.

KillAction

notifyIcon`:KillAction(`*obj*`)`

Removes an action from the notify icon's menu. If the action removed is the last action, the notify icon disappears. If the menu is displayed when this method is called, its behavior is undefined.

notifyIcon The notify icon view. You can get a reference to this view by using code similar to the following example.

```
local icon := GetRoot().notifyIcon
```

obj Saved object returned when this action was added by the `AddAction` method.

Power Registry Reference

This section describes functions that provide power-management information and that register callback functions to be executed when the Newton device is powered on or off.

BatteryCount

`BatteryCount()`

Returns the count of installed battery packs. Battery 0 is always the primary cell pack. Battery 1 is always the backup battery.

BatteryStatus

`BatteryStatus(`*which*`)`

Returns a status frame for the specified battery.

which An integer identifying the battery for which to return status information. The value 0 specifies the primary battery pack. The value 1 specifies the backup battery.

The status frame returned contains the following slots:

`batteryType` Contains one of the following symbols, or an integer:

- `'alkaline` Battery is standard alkaline.
- `'nicd` Battery is nickel-cadmium.
- `'nimh` Battery is nickel-metal hydride.
- `'lithium` Battery is lithium.

`batteryVoltage` A real number giving the current battery voltage.

`batteryCapacity`
An integer, indicating the percentage of a full charge that the battery contains.

`batteryLow` An integer, indicating the percentage of a full charge at which the “low battery” warning should be triggered by the system.

`batteryDead` An integer, indicating the percentage of a full charge at which the “dead battery” warning should be triggered and the unit shut down by the system.

`acPower` Contains a symbol (`'yes` or `'no`) indicating whether or not the unit has AC power applied. Note that this does not imply that the battery is charging. See `chargeState` to determine that.

`acVoltage` A real number giving the AC voltage being supplied by an AC adapter, or `nil` if AC power is not supplied.

`chargeState` Contains one of the following symbols, or an integer:

- `'discharging`
  The battery is not charging.
- `'trickleCharging`
  The battery is trickle-charging.
- `'fastCharging`
  The battery is fast-charging.
- `'fullyCharged`
  The battery is fully charged.

`chargeRate` An integer giving the number of minutes until the battery is charged or discharged, depending on `chargeState`.

`chargeCurrent` A real number indicating the current, in milliamps, being supplied to charge the battery, if it is charging. If the battery is discharging, this is the current supplied from the battery to the system.

`ambientTemp` A real number indicating the ambient temperature in degrees Celsius.

`batteryTemp` A real number indicating the battery temperature in degrees Celsius.

Note

A `nil` value for a slot means the underlying hardware cannot supply this information. The slots containing symbol values (`batteryType`, `chargeState`, `acPower`) may contain integers if the battery driver returned something other than the values listed here. ◆

RegPowerOff

`RegPowerOff(`*callbackID*`,`*callBackFn*`)`

Registers a function object to be executed when the Newton powers off. The arguments passed by the system to your callback function indicate the reason for the shutdown operation and its current state. Your callback must respond to all cases and must return a value indicating to the system whether to proceed with shutdown.

The value returned by the `RegPowerOff` function is unspecified and may change in the future; do not rely on values returned by this function.

callbackID A unique symbol identifying the function object to be registered; normally, the value of this parameter is the application symbol or some variation on it.

callBackFn The function object to be executed when the Newton powers off. This function object accepts two arguments and must be of the form

```
func(what, why) begin... end;
```

IMPORTANT

This callback function must not call the `RegPowerOff` or `UnRegPowerOff` functions. ▲

This function object must return a value indicating whether to continue the power-off sequence or delay it. When responding to the `'okToPowerOff` symbol, the value `true` specifies that shutdown may continue and the value `nil` cancels the shutdown process. Returning the value `'holdYourHorses` delays the impending shutdown until you call the `PowerOffResume` function.

what The state of the shutdown sequence, as indicated by the `'okToPowerOff` and `'powerOff` symbols. Table 14-3 summarizes the meanings of these symbols.

why The reason for the shutdown operation, as indicated by one of the symbols `'user`, `'idle`, or `'because`. Table 14-4 summarizes the meanings of these symbols.

Table 14-3 Values for *what* parameter to `RegPowerOff` function

Argument	Meaning	Possible response	Meaning
`'okToPowerOff`	Shutdown requested.	`nil`	Cancel shutdown.
`'okToPowerOff`	Shutdown requested.	`true`	Continue shutting down.
`'powerOff`	Shutdown imminent.	`'holdYourHorses`	Delay shutdown until `PowerOffResume` is called.
`'powerOff`	Shutdown imminent.	`nil`	Continue shutting down.
any other value	Unspecified.	`nil`	N/A

Table 14-4 Values for *why* parameter to `RegPowerOff` function

Argument	Meaning
`'user`	User cycled power switch.
`'idle`	Going to sleep.
`'because`	Unspecified.

For more information, see "Registering Power-Off Functions" beginning on page 17-25 in *Newton Programmer's Guide*.

UnRegPowerOff

`UnRegPowerOff(`*callBackID*`)`

Unregisters the specified callback function from the power-off notification mechanism. The value returned by this function is unspecified; do not rely on it.

callbackID A unique symbol identifying the function object to be unregistered. This symbol was passed to the `RegPowerOff` function to register this callback function with the power-off notification mechanism. Normally, the value of this parameter is the application symbol or some variation on it.

PowerOffResume

`PowerOffResume(`*callbackID*`)`

Used to resume a final power-off sequence which you have temporarily delayed. For details, see the description of the `RegPowerOff` function beginning on page 14-33. The value returned by the `PowerOffResume` function is unspecified and may change in the future; do not rely on values returned by this function.

callbackID A unique symbol identifying the power-off handler that delayed the power-off sequence. This symbol was passed to the `RegPowerOff` function to register the handler with the power-off notification mechanism. Normally, the value of this parameter is the application symbol or some variation on it.

RegPowerOn

`RegPowerOn(`*callbackID*`,`*callBackFn*`)`

Registers a function object to be executed when the Newton powers on. The arguments passed by the system to your callback function indicate the reason the Newton device was powered on.

The value returned by the `RegPowerOn` function is unspecified and may change in the future; do not rely on values returned by this function.

callbackID — A unique symbol identifying the function object to be registered; normally, the value of this parameter is the application symbol or some variation on it.

callBackFn — The function object to be executed when the Newton powers on. This function object accepts a single argument and must be of the form

```
func(why) begin... end;
```

IMPORTANT

This callback function must not call the `RegPowerOn` or `UnRegPowerOn` functions. ▲

why — The reason the Newton device was powered on, as indicated by one of the symbols `'user`, `'emergencyPowerOn`, `'serialgpi`, `'alarm`, or `'cardlock`. Table 14-5 summarizes the meanings of these symbols. For more information, see “Registering Power-On Functions” beginning on page 17-24 in *Newton Programmer’s Guide*.

Table 14-5 Values for *why* parameter to `RegPowerOn` function

Symbol	Meaning
`'user`	User cycled power switch.
`'emergencyPowerOn`	Last shutdown did not complete correctly.
`'serialgpi`	+5 volts on serial port GPI Pin (pin 7).
`'alarm`	Power-on caused by alarm.
`'cardlock`	Card inserted or removed.

UnRegPowerOn

`UnRegPowerOn(`*callbackID*`)`

Unregisters the specified callback function from the power-on notification mechanism. The value returned by this function is unspecified; do not rely on it.

callbackID A unique symbol identifying the function object to be unregistered. This symbol was passed to the `RegPowerOn` function to register this callback function with the power-on notification mechanism. Normally, the value of this parameter is the application symbol or some variation on it.

RegLogin

loginScreen`:RegLogin(`*callbackID*`,` *callBackFn*`)`

Registers a function object to be executed when the user gets past the login screen—either by entering the correct password or because no password is in use. For tasks involving human interface, use of the login screen script is usually more appropriate than using a power-on script. The value returned by the `RegLogin` method is unspecified and may change in the future; do not rely on values returned by this function.

For related information, see the description of the `RegPowerOn` function beginning on page 14-36.

loginScreen The view displayed just after the Newton is powered on. If the Newton device is password protected, this view will contain a number pad to enter the password into. You can use code similar to the following fragment to obtain a reference to the login screen:

```
local login := GetRoot().sleepScreen;
```

callbackID A unique symbol identifying the function object to be registered; normally, the value of this parameter is the application symbol or some variation on it.

callBackFn The function object to be executed when the Newton powers on. This function object accepts no arguments and must be of the form

```
func () begin ... end;
```

IMPORTANT

This callback function must not call the `RegLogin` or `UnRegLogin` functions. ▲

UnRegLogin

loginScreen`:UnRegLogin(`*callbackID*`)`

Removes the specified callback function from the registry of functions called by the login screen. The value returned by this function is unspecified; do not rely on it.

loginScreen The view that is displayed just after the splash screen when the Newton is powered on. You can use code similar to the following fragment to obtain a reference to the login screen:

```
local login := GetRoot().sleepScreen;
```

callbackID A unique symbol identifying the function object to be unregistered. This symbol was passed to the `RegPowerOn` function to register this callback function with the power-on notification mechanism. Normally, the value of this parameter is the application symbol or some variation on it.

Intelligent Assistant Reference

This chapter describes slots, frames, templates, functions, and methods used by the Assistant.

Data Structures

This section describes templates (action templates, task templates, and target templates) used by the Intelligent Assistant, including system-supplied templates for implementing Assistant support in your own application. This section also describes the task frame that the Assistant creates by matching user input strings to registered templates.

Task Frame

This frame, which is returned by the `ParseUtter` function, contains the following slots, as well as any created by your `PostParse` method:

Slot descriptions

`origPhrase` Holds the original user input phrase as an array of strings. Each element of this array is a single word from the user input phrase, and the words appear in the array in the order in which they appeared in the user input phrase.

`phrases` A simple array of strings derived from the *inputString* string. Each element of this array is a string that matches a template currently registered with the Assistant. These elements may be phrases themselves; under certain conditions, for example, the full name of the fax recipient ("Bob Dobbs") may be stored as a single element in this array. For more information, see "The Phrases Slot" (page 18-11) in *Newton Programmer's Guide*.

`noiseWords` An array of strings derived from the *inputString* string. Each element of this array is a string that did not match any template currently registered with the Assistant. Because the parser breaks unmatched phrases on word delimiters such as spaces, tabs, and return characters, the elements of this array are always single words.

`entries` Aliases to soup entries that were matched. Your `PostParse` method can use these aliases to retrieve matched soup entries instead of querying for them. Do not access this slot directly; instead, use the `GetMatchedEntries` function to retrieve these entries. For more information about entry aliases, see

`value` An optional slot that holds formatted strings such as phone numbers, currency values, and dates. The Assistant typically uses the `value` slot to return the results of a parse conducted using a lexical dictionary. Your `PostParse` method can use the `value` slot for this purpose as well. An example describing the use of the `value` slot appears in "The Value Slot" (page 18-12) in *Newton Programmer's Guide*.

`PostParse` Your task template supplies this method.

Action Template

The action template defines to the Assistant a frame representing a single action such as to call, pay, or remind. The completion of a complex task may require the use of several action templates, each defining a discrete task that is completed as part of the primary task. The action template also stores a list of words or phrases, called the lexicon, that the Assistant uses to match this template with words or phrases from user input.

The Assistant provides several predefined action templates. They are summarized in "System-Supplied Action Templates" (page 15-4). You use the system-supplied `dyna_user_action` template to define new actions to the Assistant.

Your action template must provide the following required slots:

`value` The Assistant uses this slot only when using a lexical dictionary to parse a special-format string such as a phone number. You can use this slot to hold a comment string that indicates the name of this template.

`isa` The value of this slot identifies the object type of the frame created from this template. You must store a symbol in this slot that identifies this template as being an action that you defined (as opposed to one defined by the system). The symbol `'dyna_user_action` is acceptable, as would be the symbol for any template derived from a template having the value `'dyna_user_action` in its `isa` slot. For more information, see "Defining Your Own Frame Types to the Assistant" (page 18-16) in *Newton Programmer's Guide*.

`lexicon` This slot holds an array of one or more words or phrases to match with this template. The Assist slip displays the first value in this array as an item in the Please pop-up menu when this template is matched as the primary action.

System-Supplied Action Templates

This section describes the system-supplied action templates, which are the templates at the top level of the action template hierarchy.

`dyna_user_action`
Generic action template having no lexicon. All your action templates must descend from this template to enable the Assistant to resolve verb matching conflicts.

`call_act` Action template for using the built-in Call application. This template's lexicon includes the strings `"call"`, `"phone"`, "ring", and `"dial"`.

`find_act` Action template for invoking the Find service. This template's lexicon includes the strings `"find"`, `"locate"`, `"search for"`, and `"look for".`

`fax_act` Action template for faxing the target data item. This template's lexicon includes the string `"fax"`.

`print_act` Action template for printing the target data item. This template's lexicon includes the string `"print"`.

`about_act` Action template for displaying the About box. This template's lexicon includes the string `"about newton"`.

`time_act` Action template for retrieving time values from the Time Zones application. This template's lexicon includes the strings `"time"`, `"time in"`, `"the time in"`, `"what time is it"`, `"what time is it in"`, `"the time in"`, `"what time"`, `"what is the time"`, and `"what is the time in"`.

`remind_act` Action template for creating To Do items. This template's lexicon includes the strings `"remember"`, `"remind"`, `"remind me"`, `"to do"`, `"todo"`, `"don't forget to"`, and `"don't let me forget to"`.

`mail_act` Action template for sending electronic mail. This template's lexicon includes the strings `"mail"`, `"send"`, and `"email"`.

`schedule_act` Action template for scheduling meetings and events in the Dates application. This template's lexicon includes the string `"schedule"`.

`meet_act` Action template for scheduling meetings and events in the Dates application. This template is based on the `schedule_act` template. Its lexicon includes the strings `"meet"`, `"meet me"`, `"see"`, and `"talk to"`.

`meal_act` Action template for scheduling meals in the Dates application. Because meals are considered meetings (events with a beginning and ending time), this template is based on the `schedule_act` template.

Meals

These system-supplied action templates are used to schedule meals in the built-in Dates application. All of these templates provide a string (such as `"breakfast"`) that is used as the default title of the meeting. These templates also define a `usualTime` slot that provides a default value for the starting time of the meal. These templates are based on the `meal_act` template.

`breakfast_act` Action template for scheduling breakfast in the Dates application. Its lexicon includes the string `"breakfast"`. The default starting time for this meeting is 7:00 A.M.

`brunch_act` Action template for scheduling brunch in the Dates application. Its lexicon includes the string `"brunch"`. The default starting time for this meeting is 10:00 A.M.

lunch_act — Action template for scheduling lunch in the Dates application. Its lexicon includes the string "lunch". The default starting time for this meeting is 12:00 P.M. (noon).

dinner_act — Action template for scheduling dinner in the Dates application. Its lexicon includes the string "dinner". The default starting time for this meeting is 7:00 P.M.

Special Events

This section describes templates that define special_event_act frames. These action frames define methods used to schedule events in the Dates application. With the exception of the holiday template, all these action frames schedule events that recur annually on a specified date. The event that the holiday template schedules does not repeat because holidays do not necessarily fall on the same date each year. The special_event_act template is derived from the schedule_act template.

birthday — Action template for scheduling an annual repeating birthday event in the Dates application and adding this information to an appropriate Names soup entry if one exists. Its lexicon includes the strings "birthday", "bday", and "b-day".

anniversary — Action template for scheduling an annual repeating anniversary event in the Dates application and adding this information to an appropriate Names soup entry if one exists. Its lexicon includes the string "anniversary".

holiday — Action template for scheduling a non repeating holiday event in the Dates application. Its lexicon includes the string "holiday".

Developer-Supplied Action Templates

You must supply the action template specified by the value of your task template's primary_act slot.

You must also supply any additional action templates specified by the `signature` slot of the task template.

Target Template

The target template defines a frame that represents to the Assistant the target of an action; targets are generally people, places, or things. The target template also stores a list of words or phrases, called the lexicon, that the Assistant uses to match the template with words or phrases from user input.

The Assistant provides several predefined target templates. They are summarized in "System-Supplied Target Templates" (page 15-8). You need to use the `dyna_user_obj` template to define new targets to the Assistant.

Your target template must provide the following required slots:

`value` Currently unused, but required for compatibility with future versions of the Assistant. You can put a comment string indicating the name of the template in this slot.

`isa` The value of this slot identifies the object type of the frame created from this template. You must store a symbol in this slot that identifies this template as a target that you defined (as opposed to one defined by the system). The symbol `'dyna_user_obj` is acceptable, as is the symbol for any template derived from a template having the value `'dyna_user_obj` in its `isa` slot. For more information, see "Defining Your Own Frame Types to the Assistant" (page 18-16) in *Newton Programmer's Guide*.

`lexicon` Required unless your template is derived from a system-supplied template, in which case your template can use the system-supplied lexicon. This slot holds an array of one or more words or phrases to match with this template.

System-Supplied Target Templates

The Assistant provides the predefined target templates described in this section. You use the `dyna_user_obj` template to define new targets to the Assistant.

Places

The following system-supplied templates define `where_obj` templates:

`address, city, region, country, postal_code, phone, parsed_phone, phone_tag, faxPhone, homePhone, workPhone, carPhone, mobilePhone, beeper, places, company, city, county, state, country, town`, and `province`

No lexicons are associated with these templates because the Assistant uses lexical dictionaries to match them. The `where_obj` template is derived from the `user_obj` template.

Note that in addition to the items you would expect to be treated as places (such as postal codes and the names of cities, states, and provinces), the Assistant treats phone numbers as places.

Times

The templates described here define `when_obj` frames. The `when_obj` template is derived from the `parsed_number` template.

`time, date`

User Object Template

The system-supplied user object template provides the basis for the Assistant's conflict resolution mechanism. This section describes system-supplied templates for persons, groups, titles, and custom targets, all of which are based on the user object template. You must use the `dyna_user_obj` template to define new targets to the Assistant.

For more information, see "Resolving Template-Matching Conflicts" (page 18-13) in *Newton Programmer's Guide*.

`dyna_user_obj` — Generic target template having no lexicon. All of your target templates must descend from this template to enable the Assistant to resolve conflicts. This template is derived from the system-supplied user object template.

`who_obj` — Abstract target template having no lexicon, descended from the system-supplied user object template. Do not base your templates on the `who_obj` template. Instead, base your target templates on the `dyna_user_obj` template.

`what_obj` — Abstract target template having no lexicon, descended from the system-supplied user object template. Do not base your templates on the `what_obj` template. Instead base your target templates on the `dyna_user_obj` template.

`where_obj` — Abstract target template having no lexicon, descended from the system-supplied user object template. Do not base your templates on the `where_obj` template. Instead, base your target templates on the `dyna_user_obj` template.

People

The system-supplied `person` template defines a `who_obj` frame. The `title`, `affiliate`, `custom`, and `group` templates are based on the `person` template. These templates have no lexicons associated with them because they are the equivalent of abstract classes—you do not instantiate frames based on these templates but derive your own templates from them or use system-supplied templates derived from them.

`person` — Target template for frames representing an individual person. You can base your own templates representing individual persons on this template.

`title` — Target template for frames representing the title of an individual person, such as `"Manager"`, `"Owner"`, and so on. You can base your own templates representing titles of individual persons on this template.

`affiliate` Target template for frames representing a person affiliated with an individual, such as a friend, co-worker, and so on. You can use this template to create your own templates representing affiliated persons.

`group` Target template for frames representing groups of people, such as `"writers"`, `"engineers"`, and so on. You can base your own templates representing groups of people on this template.

`custom` Target template for frames representing customized categories of persons, such as those taller than a specified height. You can base your own customized categories of individual persons on this template.

Miscellaneous Templates

This section describes the `salutationPrefix` template, which is derived from the system-supplied `parser_obj` template.

`salutationPrefix`

Action template for creating `parser_obj` frames. These frames are used to assign meaning to words that would normally be parsed as noise words. This template's lexicon includes the strings `"dear"`, `"to"`, `"attention"`, `"attn"`, `"attn."`, `and "hey"`.

Developer-Supplied Target Templates

You must supply any required target template not supplied by the system. Required target templates are specified by the task template's `signature` slot.

Task Template

The task template defines an application behavior to the Assistant. A behavior consists of an action, such as `"call"`, `"pay"`, or `"remind"`, that is generally directed at a target, such as `"Bob"` or `"Apple"`. An action and its target are defined by an action template and a target template, respectively.

All task templates must define the following required slots:

Slot descriptions

`isa`	This slot identifies the object type of the frame created from this template. Task templates must store only the value `'task_template` in this slot. You cannot use the symbol for another template derived from this one instead.
`primary_act`	This slot stores the name of the action template that defines an application behavior to the Assistant. The action template that this slot identifies may itself require the use of additional action templates and target templates.
`preConditions`	This slot stores an array of symbols specifying the names of slots that the Assistant creates to store action frames and target frames. The `preConditions` array must have the same number of elements as the `signature` array because the Assistant uses these two arrays in parallel. For more details, see "The Signature and PreConditions Slots" (page 18-10) in *Newton Programmer's Guide*.
`signature`	This slot holds an array of frame types that may be stored in the slots specified by the `preConditions` array. The `signature` array must hold the frame type of at least one action frame and zero or more target frames. The `signature` array must have the same number of elements as the `preConditions` array because the Assistant uses these two arrays in parallel.

For more details, see “The Signature and PreConditions Slots” (page 18-10) in *Newton Programmer’s Guide*.

`PostParse` The method to be invoked after the Assistant parses the user input. Frequently, the task template’s primary action is actually invoked by the `PostParse` method—for example, if the user asks to `"fax Bob"` and Newton cannot do so until the Assistant has retrieved Bob’s fax number, the primary action of sending the fax would correctly be invoked after the `ParseUtter` function returns the task frame.

Another common use for the `PostParse` method is to display a task slip view that provides the user with an opportunity to confirm, modify, or dismiss the primary action before it is executed.

`taskslip` Optional. This slot holds a view template associated with the task template. Commonly this view is a task slip that displays information about the primary action for confirmation or editing by the user.

`score` Used internally by the Assistant. Place the value `nil` in this slot.

Developer-Supplied Task Template

You must always supply a task template, which defines the application behavior made available through the Assistant.

Help Topic Slot

Your application’s base view can supply a `viewHelpTopic` slot that the Assistant uses to open a help book to the appropriate topic.

```
viewHelpTopic: // slot specifying your app’s help topic
```

Assistant Functions and Methods

This section describes functions and methods used by the assistant. The first part of this section describes system-supplied functions and methods. The second part describes application-defined functions and methods.

RegTaskTemplate

`RegTaskTemplate(`*theTemplate*`)`

Registers a specified task template with the Assistant.

theTemplate The template to register.

UnRegTaskTemplate

`UnRegTaskTemplate(`*theTemplate*`)`

Unregisters a specified task template with the Assistant.

theTemplate The template to unregister.

ParseUtter

`ParseUtter(`*inputString*`);`

This function takes the following actions and calls your `PostParse` method:

- parses the input string passed as its argument. If this string contains more than 15 words, the `ParseUtter` function returns nil and takes no further action.
- matches words and phrases in the input string to templates currently registered with the Assistant
- creates action frames and target frames from the matched templates
- creates a task frame based on matching an action frame to a task template
- creates slots holding action frames and target frames in the task frame

- as necessary, creates the `origPhrase`, `phrases`, `noiseWords`, `entries`, and `value` slots in the task frame

See "Programmer's Overview" (page 18-5) in *Newton Programmer's Guide* for a detailed description of these tasks.

inputString	The string this function attempts to match with registered templates.

GetMatchedEntries

`GetMatchedEntries(`*which*`,` *entries*`);`

Returns an array of entry aliases to soup entries that were matched by the Assistant.

which	Symbol specifying a subset of entries to return. Acceptable values are any of the `'person`, `'places`, or `'allEntries` symbols.
entries	The `entries` slot from the result frame returned by the Assistant.

Typically you would call this function from your `PostParse` method, passing to it the `entries` slot of the result frame as in the following code fragment.

```
// self is the task frame
local candidates := GetMatchedEntries('allEntries,
                                       self.entries);
```

Developer-Supplied Assistant Functions and Methods

This section describes functions, methods, and templates that you must supply.

PostParse

taskFrame:`PostParse();`

Your `PostParse` method must do anything necessary to perform the action specified by the frame in the `primary_act` slot of *taskFrame*, such as handling error conditions, extracting further information from the result frame returned by the `ParseUtter` function or displaying a task slip to the user. The Assistant calls your `PostParse` function after matching all the templates specified by the task template.

taskFrame	The frame created by the Assistant from the task template.

Built-in Applications and System Data Reference

This chapter describes the constants, data structures, protos, functions, and soup formats used to interface with the built-in applications and other system data.

Names Reference

This section describes the constants, data structures, protos, methods, functions, and soup formats of the Names application.

Names Constants

The constants described in this section are used by the Names application.

Table 16-1 Names card layouts

Constant	Value	Description
`kSquiggle`	0	Layout that uses squiggly line.
`kPlain`	1	Plain layout.
`kSeparate`	2	Layout with dashed lines.
`kCross`	3	Layout with crossed lines.
None Available	4	Layout with bullet holes all over.
None Available	5	Layout with dotted fading line.
None Available	6	Layout with big bullet holes in a single line.

Names Data Structures

This section describes the special slots used in Names dataDefs and viewDefs.

Names Data Definition Frame

Names data definition frames contain the following special slots, in addition to the standard data definition slots. For information on the standard set of dataDef slots, see "newtStationery" (page 4-3).

Slot descriptions

`overviewIcon` A tiny version of the icon in the `icon` slot, to use when displaying this kind of card in an overview. You should keep the icon smaller than 11x11 pixels; a larger icon looks awkward in the overview. `Nil` values are not allowed.

`viewsToDisplay` An array of symbols for the names of viewDefs registered for this data definition. This is needed so that all the viewDefs will show up in the All Info view.

Names View Definition Frame

Names view definition frames contain a special slot called `infoFrame`, in addition to the standard view definition slots; see “viewDef Frame” (page 4-1).

The `infoFrame` should have the following slots:

Slot descriptions

`checkPaths` — An array of paths to data collected by this layout.

`checkPrefix` — An array of two path expressions. The first is the path for the first data set, or `true` if the first data set should be stored at the top level of the soup entry. The second is the path for subsequent data sets (or `nil` if no multiple data sets are allowed). If the second path is `nil`, the viewDef appears in the Add picker until the user chooses it and adds data to it. At that point, no more data of that type can be added, and the item is removed from the Add picker.

`stringData` — Set to `true` if the data sets consist of single strings. A `nil` value means frames are created for each item rather than strings.

`format` — A string to be passed to the `ParamStr` utility function along with the data set. The string returned by `ParamStr` is used to display in the All Info view.

Alternatively, you can define a `FormatFunc` method in this frame.

`FormatFunc` — Instead of including a `format` slot, you may define this method, but you must do one or the other.

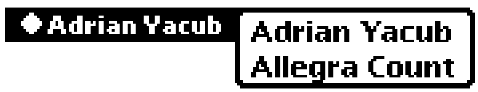
This method is passed one argument, *pathArray*, an array of the data in the data set, corresponding to the paths in the `checkPaths` slot. It should return a string to be displayed in the All Info view.

Names Protos

The `protoPersonaPopup` and `protoEmporiumPopup` protos provide pickers for personae and emporia.

protoPersonaPopup

Lets the user maintain and switch between different owner cards, or "personae." Here's an example:



The diamond appears only if there is more than one owner card; otherwise you see just a name without a diamond. Tapping the name produces a picker showing the names of all owner cards stored by the Names application in this Newton device.

The methods `JamIt` and `SetupText` described below are defined in this proto.

JamIt

myPersonaPopup`:JamIt()`

Calls `SetupText` and updates the screen. This method should be called if the current settings change.

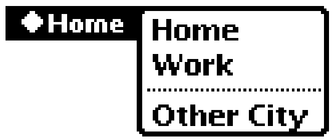
SetupText

myPersonaPopup`:SetupText()`

Returns a string to display as the current persona. If more than one persona is available, a diamond is appended to the beginning of the string.

protoEmporiumPopup

This proto is used for a picker that lets the user maintain and switch between different information relevant to various places she may be in. Here's an example:



When the user chooses a different emporium or city, information like time zone, area code, and so on is changed to reflect the different location. Choosing "Other City" allows the user to pick a different city anywhere in the world.

The methods `JamIt` and `SetupText` described below are defined in this proto.

JamIt

myEmporiumPopup`:JamIt()`

Calls `SetupText` and updates the screen. This method should be called if the current settings change.

SetupText

myEmporiumPopup`:SetupText()`

Returns a string to display as the current emporium. If more than one emporium is available, a diamond is appended to the beginning of the string.

Names Functions and Methods

This section lists the Names functions and methods. To obtain a reference to the Names application to send one of these messages, use this code:

```
GetRoot().cardfile
```

Note that future Newton devices may not include the Names application. You should therefore check for the existence of the Names application before trying to access it. Use the following code to test for this:

```
if GetRoot().cardfile then ...
```

AddCard

cardfile:`AddCard(`*dataDefType*`,` *entryFrame*`)`

Creates a new card in the Names application.

dataDefType A symbol giving the data definition type for the new Names entry. The following symbols are allowed for the built-in dataDefs: `'person`, `'owner`, `'group`, `'company`, or `'worksite`.

entryFrame A frame containing any number of those slots listed under "Names Soup Format" (page 16-15) for the type of soup entry specified in the *dataDefType* parameter. See "Person Entries" (page 16-15), "Owner Entries" (page 16-18), "Group Entries" (page 16-20), "Company Entries" (page 16-21), or "Worksite Entries" (page 16-22).

This function returns the newly created entry, or `nil` if none was created. (The entry is not created if *dataDefType* is invalid.)

AddCardData

cardfile:`AddCardData(`*entry*`,` *layoutSym*`,` *newData*`)`

Adds information to a Names soup entry. It allows you to specify an entry and add a string or frame to it.

entry The entry in the Names soup to which you want to add data.

layoutSym A symbol giving the data definition type for the additional Names layout symbol. The possible values of this parameter depend on the *dataDefType* of *entry*.

This parameter can be the name registered with the Names application for a custom viewDef.

If the *dataDefType* of *entry* is `'person`, *layoutSym* can be `'name`, `'company`, `'address`, `'phone`, `'email`, `'pager`, or `'personal`.

If the *dataDefType* of *entry* is `'owner`, *layoutSym* can be `'name`, `'company`, `'address`, `'phone`, `'email`, `'pager`, `'personal`, `'signature`, `'creditCard`, or `'bankAccount`.

If the *dataDefType* of *entry* is `'company`, *layoutSym* can be `'name`, `'address`, `'phone`, or `'email`.

If the *dataDefType* of *entry* is `'worksite`, *layoutSym* can be `'connection` or `'mailInfo`.

newData The data you wish to add. The type of this parameter must be appropriate for the value of *layoutSym*. See the description under the appropriate section of "Names Soup Format" (page 16-15).

This parameter can be an object of the proper format for a custom viewDef.

Note that some of the possible values for *layoutSym* do not have a corresponding slot in the description of the soup entries.

The symbols `'phone`, `'pager`, `'creditCard`, and `'bankAccount` are elements of the arrays described in the soup format. That is, if you pass in `'phone` for *layoutSym*, *newData* should be an element to add to the `'phones` array.

If you pass in `'personal` for *layoutSym*, *newData* should be a frame with either an `'anniversary` or a `'bday` slot (or both). These slots have the same meaning as the soup entry slots.

Similarly, if you pass in `'connection` for *layoutSym*, *newData* should be a frame with a `'connectionPhone` and `'connectionNetwork` slots. These slots have the same meaning as the soup entry slots.

And, if you pass in `'emailInfo` for *layoutSym*, *newData* should be a frame such as that returned by the `BcEmailNetwork` method (page 16-10). That is, it should have the following three slots: `'mailPhone`, `'baud`, and `'mailNetwork`.

For example, to add a new affiliate to a person card:

```
GetRoot().cardfile:AddCardData( aPersonEntry, 'name,
      {first: "Test", last: "This"});
```

To add a fax number to a company:

```
GetRoot().cardfile:AddCardData ( aCompanyEntry, 'phone,
      SetClass( MakePhone( {areacode: "617",
      phone:"555-1212"}), 'faxPhone));
```

AddLayout

cardfile:`AddLayout(`*layout*`)`

Adds a layout to the Show picker, under a line below Card and All Info. The layout should be based on the `newtLayout` proto. Remove layouts added with this method by using the Names method `SafeRemoveLayout`.

layout — The layout you want to add. This layout must have the following slots:

`name`	A string shown in the Show picker.
`symbol`	A symbol, which includes your developer signature, uniquely identifying this layout. This symbol must be passed to the `EnsureInternal` function.
`type`	Set this slot to the symbol `'viewer`.

`protection`
: Set this slot to the symbol `'private`.

BcCreditCards

cardfile:`BcCreditCards(`*inEntry*`, `*inWhich*`)`

Returns the frame stored in the `creditCards` slot (page 16-19) of an owner soup entry.

inEntry — An owner entry in the Names soup.

inWhich — The class of the string in the `creditCards.creditCardName` slot of owner entries to find. Or an array of class symbols, in which case an array of matches for each symbol is returned.

Phone card classes begin with a `'|string.card.phoneCard|` prefix. Credit card classes begin with a `'|string.card.creditCard|` prefix. For a full list of the built-in classes see the description of the `creditCards.creditCardName` slot of owner entries in "Owner Entries" (page 16-18).

BcCustomFields

cardfile:`BcCustomFields(`*inEntry*`, `*inWhich*`)`

Returns an array containing frames with custom field information. These frames have two slots: the `label` slot contains the label for the custom field, and the `value` slot contains the value stored in the slot of that custom field. The method returns `nil` if no frames are found.

inEntry — An entry in the Names soup.

inWhich — The name of the custom slot. `Nil` returns all custom fields. If this parameter is an array of symbols instead of a single symbol, the matches for all the symbols are returned.

BcEmailAddress

cardfile:`BcEmailAddress(`*entry*`, `*which*`)`

Takes a soup entry and an e-mail type, and returns an array of frames with e-mail information. The `email` slot in these frames is a string representing the e-mail address. If the entry is an owner card, the frame may also contain an `emailPassword` slot, which holds the password string. The method returns `nil` if no frames are found.

entry	An entry in the Names soup.
which	The class of the e-mail address to find. The built-in e-mail address classes are listed under the `email` slot of person entries in "Person Entries" (page 16-15). If this parameter is an array of symbols instead of a single symbol, matches for all symbols are returned.

BcEmailNetwork

cardfile:`BcEmailNetwork(`*entry*`, `*type*`)`

Takes a soup entry and the type of e-mail network, and returns information about the entry's e-mail network.

entry	An entry in the Names soup.
type	The type of network (`'sprint` or `'concert`). If this parameter is an array of symbols instead of a single symbol, matches for all the symbols are returned.

This method returns an array of frames with the following slots:

Slot description

`mailNetwork`	A network type symbol.
`mailPhone`	A string for a phone number.
`baud`	An integer, the baud rate.

The method returns `nil` if no frames are found.

BcPhoneNumber

cardfile:`BcPhoneNumber(`*inEntry*`,` *inWhich*`)`

Returns an array of phone numbers, as strings, for the soup entry specified in *inEntry*.

inEntry — An entry in the Names soup.

inWhich — The class of the phone numbers to return. The built-in phone address classes are listed under the `phones` slot of person entries in “Person Entries” (page 16-15). For example, to get voice numbers pass in `'phone`; to get fax numbers, pass in `'faxPhone`.

OpenTo

cardfile:`OpenTo(`*entry*`,nil)`

Opens the card specified by the *entry* parameter. This method opens the Names application if necessary. If Names is open, you should use the `ShowFoundItem` method.

entry — The Names soup entry to show.

`nil` — Always pass `nil` for the second parameter.

The return value of this function is undefined.

ReplaceInkData

cardfile:`ReplaceInkData(`*entry*`,` *layoutSym*`,` *oldString*`,` *checkPath*`,` *newString*`)`

Replaces a specified ink string in a Names soup entry with a recognized string.

entry — The entry in the Names soup to which you want to add data.

layoutSym — A symbol identifying the data definition of *entry*. See the *layoutSym* parameter to the `AddCardData` method.

oldString — The ink string to replace.

checkPath The additional path where the data is found; see "Names View Definition Frame" (page 16-3).

newString The recognized string to replace *oldString*.

RegNamesRouteScript

`RegNamesRouteScript(`*symbol*`, `*routeScriptFrame*`) // platform file function`

Adds an application-defined action to the Action picker in the Names application. The companion to this function is `UnRegNamesRouteScript` (page 16-14).

IMPORTANT

This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

`call kRegNamesRouteScriptFunc with (`*symbol*`, `*routeScriptFrame*`);`

▲

symbol A unique symbol identifying the action you are adding. You should append your developer signature to ensure that this symbol is unique.

routeScriptFrame A frame describing the routing action, as described in Chapter 21, "Routing Interface," of *Newton Programmer's Guide*. This frame is summarized below:

```
{
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
}
```

Here's an example of using the `RegNamesRouteScript` function:

```
call kRegNamesRouteScriptFunc with
('|EntryDumper:PIEDTS|,
   {  GetTitle: func(target) begin
         if GetTargetCursor(target, nil):entry() then
            "Dump entry";
         else
            nil;// no selections, so don't show in list
      end,
      icon: nil,
      RouteScript: func(target, targetView) begin
         local curs:=GetTargetCursor(target, nil);
         local e := curs:Entry();
         while e do begin
            print(e);
            e:=curs:Next();
         end;
      end
   });
```

Note

The return value of this function is undefined; do not rely on it. ◆

SafeRemoveLayout

cardfile`:SafeRemoveLayout(`*layout*`) // platform file function`

Safely removes a cardfile layout added by `AddLayout` (page 16-8) from the Names application.

IMPORTANT

This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

`call kCardFileSafeRemoveLayoutFunc with (`*layout*`);`

▲

layout A symbol identifying the cardfile layout you want to remove. This is the same symbol you passed to the cardfile method `AddLayout` to add the layout.

The return value of this function is undefined; do not rely on it.

ShowFoundItem

cardfile`:ShowFoundItem(`*entry*`,nil)`

Opens the card specified by the *entry* parameter. This method requires that the Names application be open. If Names is closed, you should use the `OpenTo` method.

entry The Names soup entry to show.

`nil` Always pass `nil` for the second parameter.

The return value of this function is undefined.

UnRegNamesRouteScript

`UnRegNamesRouteScript(`*symbol*`)// platform file function`

Removes an application-defined action from the Action picker in the Names application. It removes only actions added by `RegNamesRouteScript` (page 16-12).

IMPORTANT

This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

`call kUnRegNamesRouteScriptFunc with (`*symbol*`);`

▲

symbol A symbol identifying the action you are removing.

Note

The return value of this function is undefined; do not rely on it. ◆

Here's an example of using the `UnRegNamesRouteScript` function:

```
call kUnRegNamesRouteScriptFunc with
('|EntryDumper:PIEDTS|);
```

Names Soup Format

This section describes the format of entries in the Names soup. Five different types of entries are stored in this soup: persons, owners, groups, companies, and worksites. You can identify an entry by calling the `ClassOf` function on the entry. `ClassOf(`*entry*`)` returns one of the following symbols: `'person`, `'owner`, `'group`, `'company`, or `'worksite`.

The slots contained in these entry frames are described in "Person Entries" (page 16-15), "Owner Entries" (page 16-18), "Group Entries" (page 16-20), "Company Entries" (page 16-21), and "Worksite Entries" (page 16-22).

Person Entries

Person entries consist of a frame with the following slots:

Slot descriptions

`version`	The version number of the Names application.
`class`	The symbol `'person`.
`cardType`	An integer; see Table 16-1 "Names card layouts."
`name`	A frame with the following slots: `honorific` A string or rich string for an honorific title; e.g., "Ms." or "Dr." `first` A string or rich string for a first name. `last` A string or rich string for a last name. `title` A string or rich string for a job title.
`names`	An array of affiliated names, such as company contacts, family members, and so on, added by the user by picking "Affiliate" from the "Add" picker. This array contains frames such as that for the `name` slot.

`company`	A string or rich string for the company name.
`title`	A string or rich string representing this person's title at the company stored in `company`.
`companies`	An array of frames of type `{company:` *stringOrRichString*`, title:` *stringOrRichString*`}`.
`address`	A string or rich string for the first line of an address.
`address2`	A string or rich string for the second line of an address.
`addresses`	An array of frames for additional addresses. These frames contain the following slots: `address`, `address2`, `city`, `region`, `postal_code`, and `country`.
`city`	A string or rich string for a city.
`region`	A string or rich string for region (a state in the U.S.).
`postal_code`	A string or rich string for postal code (zip code in U.S.).
`country`	A or rich string naming the country. If this is a standard (not rich) string that is recognized by the system as the name of a country, it will have a class as set by `SetCountryClass`; for more information on this, see the description of `SetCountryClass`.

`phones`	An array that contains strings or rich strings for phone numbers. The user can set the class of this string by picking from the Phones popup. The built-in phone classes are:
	`phone`
	`homePhone`
	`workPhone`
	`faxPhone`
	`carPhone`
	`mobilePhone`
	`homefaxPhone`

`email` — A string or rich string for an e-mail address. The user can set the class of this string by picking from the Email popup. The built-in e-mail classes are:

`|string.email|`
`|string.email.internet|`
`|string.email.aol|`
`|string.email.compuserve|`
`|string.email.mcimail|`
`|string.email.attmail|`
`|string.email.easylink|`
`|string.email.prodigy|`
`|string.email.genie|`
`|string.email.delphi|`
`|string.email.msn|`
`|string.email.interchange|`
`|string.email.radiomail|`

`emailAddrs` — An array of frames of additional e-mail addresses with the following slot:

- `email` — A string or rich string. The user can set the class of this string by picking from the Email popup. The built-in e-mail classes are listed under the `email` slot.

`emailPassword` — Always `nil`.

`pagers` — An array of pager information frames. Each frame can have the following slots:

- `pagerNum` — A string or rich string for a pager number. The user can set the class of this string by picking from the Pagers popup. The built-in pagers classes are:

  `|string.pager|`
  `|string.pager.skytel|`
  `|string.pager.mobilcomm|`
  `|string.pager.embarc|`

`pagerPIN` A string or rich string for a pager PIN.

`bday` Either an integer for the date the user entered for the birthday in the number of minutes passed since midnight, January 1, 1904, or a string or rich string.

`bdayEvent` An alias to a Dates event. For more information on entry aliases, see Chapter 11, "Data Storage and Retrieval," in *Newton Programmer's Guide*.

`anniversary` Either an integer for the date the user entered for the anniversary, in the number of minutes passed since midnight, January 1, 1904, or a string or rich string.

`anniversaryEvent`
An alias to a Dates event. For more information on entry aliases, see Chapter 11, "Data Storage and Retrieval," in *Newton Programmer's Guide*.

`notes` An array of note objects. Each element in this array is a frame with the same format as the `data` slot in a soup entry in the Notes application; see "Notes Soup Format" (page 16-82).

`sorton` A string (not a rich string because of sorting).

Owner Entries

Owner entries consist of a frame with the same slots as in "Person Entries" (page 16-15). However, three of the slots hold different values, and there is an additional slot. The following three slots exist in person entries, but have different meanings:

Slot descriptions

`class` The symbol `'owner`.

`emailPassword` A string for the owner's e-mail password.

`emailAddrs` An array of frames of additional e-mail addresses with the following slots:

`email` A string or rich string. The user can set the class of this string by picking from the Email popup. The built-in e-mail classes are listed under the `email` slot for person entries.

`emailpassword`
A string.

There is one additional slot in owner entries:

`owner` A frame with the slots shown below.

Slot descriptions for `owner` frame

`bankAccounts` An array of frames corresponding to bank accounts. These frames contain the following slots:

`bankAcctNum`
A string or rich string that contains the account number.

`bankContactNum`
A phone number string or rich string for the bank account contact.

`creditCards` An array of frames corresponding to credit card accounts. These frames contain the following slots:

`creditCardName`
A string or rich string for the credit card's name. The user can set the class of this string by picking from the Card popup. The built-in credit card classes are:

```
|string.card|
|string.card.phonecard|
|string.card.creditcard|
|string.card.phonecard.att|
|string.card.phonecard.mci|
|string.card.phonecard.sprint|
|string.card.creditcard.visa|
|string.card.creditcard.mastercard|
|string.card.creditcard.amex|
|string.card.creditcard.discover|
```

`creditCardNum`
: A string or rich string for the account number.

`creditCardExpDate`
: Either an integer for the expiration date, in number of minutes since midnight, January 1, 1904, or a string or rich string.

`creditCardContactNum`
: A string or rich string for the phone number of the credit card account contact.

`signature` The signature the user entered in the signature slip. It is an ink frame.

Group Entries

A group entry consists of a frame with the following slots:

`version` The version number of the Names application.

`class` The symbol `'group`.

`cardType` An integer; see Table 16-1 “Names card layouts.”

`group` A string or rich string for the group’s name.

`groupInfo` Contains a frame with the following slot:

`nowShowing` Either the value `'selected` or `'all`. This reflects whether the user has checked the Selected Only box in the view that adds people to the group.

`members` An array containing name reference frames representing the members of the group. These frames are described in “Name References” (page 5-1) in *Newton Programmer’s Reference*.

`notes` An array of note objects. Each element in this array is a frame with the same format as the `data` slot in a soup entry in the Notes application; see “Notes Soup Format” (page 16-82).

`sorton` A string (not a rich string because of sorting).

Company Entries

Company entries consist of a frame with the following slots:

`version`	The version number of the Names application.
`class`	The symbol `'company`.
`name`	The name the user added by picking "affiliate" from the Add picker. It is a frame with the following slots: `honorific` A string or rich string for an honorific title, e.g., "Ms." or "Dr." `first` A string or rich string for a first name. `last` A string or rich string for a last name. `title` A string or rich string for a job title.
`names`	An array of affiliated names; as added by the user by picking "Affiliate" from the "Add" picker, such as company contacts, family members, and so on. This array contains frames such as that for the `name` slot.
`cardType`	An integer; see Table 16-1 "Names card layouts."
`company`	A string or rich string for the company name.
`address`	A string or rich string for the first line of the address.
`address2`	A string or rich string for the second line of the address.
`addresses`	An array of frames for additional addresses. These frames contain the following slots: `address`, `address2`, `city`, `region`, `postal_code`, and `country`.
`city`	A string or rich string containing the name of a city.
`region`	A string or rich string for region (state in U.S., province in Canada).
`postal_code`	A string or rich string for postal code (zip code in U.S.).
`country`	A or rich string naming the country. If this is a standard (not rich) string that is recognized by the system as the name of a country, it will have a class as set by `SetCountryClass`; for more information on this, see the description of `SetCountryClass`.

`phones`	An array of strings or rich strings containing phone numbers. The user can set the class of these string by picking from the Phone popup. The built-in phone classes are listed under the `phones` slot for person entries.
`email`	A string or rich string for an e-mail address. The user can set the class of this string by picking from the Email popup. The built-in e-mail classes are listed under the `email` slot for person entries.
`emailAddrs`	An array of frames of additional e-mail addresses of the format `{email:`*stringOrRichString*`}`.
`notes`	An array of note objects. Each element in this array is a frame with the same format as the `data` slot in a soup entry in the Notes application; see "Notes Soup Format" (page 16-82).
`sorton`	A string (not a rich string because of sorting).

Worksite Entries

Worksite entries consist of a frame with the following slots:

`version`	The version number of the Names application.
`class`	The symbol `'worksite`.
`cardType`	An integer; see Table 16-1 "Names card layouts."
`place`	A string or rich string for the place name.
`dialingPrefix`	A string or rich string for dialing prefix needed to get an outside line from this worksite.
`areaCode`	A string or rich string for the area code of this worksite.
`printer`	A string representing the printer the user has chosen from among network printers.
`mailAccess`	An array of frames of the following form: `{mailPhone:`*stringOrRichString*`,` `mailNetwork:'concert,` `baud:1200}`

`connectionPhone` A phone number string to access e-mail at this site.

`connectionNetwork`
: A string for the AppleTalk Name Binding Protocol (NBP) address. The format of this string is `"`*computerName*`:Docker@`*zone*`"`.

`cityAlias`
: An alias to an entry from an undocumented soup which contains informations about the closest city. The soup entry contains slots for the city name, its country, latitude and longitude, etc.

  For more information on entry aliases, see Chapter 11, “Data Storage and Retrieval,” in *Newton Programmer’s Guide*.

`countrySymbol` A symbol representing the country.

`country`
: A or rich string naming the country. If this is a standard (not rich) string that is recognized by the system as the name of a country, it will have a class as set by `SetCountryClass`; for more information on this, see the description of `SetCountryClass`.

`notes`
: An array of note objects. Each element in this array is a frame with the same format as the `data` slot in a soup entry in the Notes application; see “Notes Soup Format” (page 16-82).

`sorton` A string (not a rich string because of sorting).

Dates Reference

This section describes the variables, constants, and protos used by the Dates application. Also covered are all the Dates methods and functions, and the exceptions they throw, and the Dates soup format.

Dates Variables and Constants

This section lists two variables you may set, and the constants used by the Dates application.

Table 16-2 Dates variables

Variable	Description
`firstDayOfWeek`	Specifies what the first day of the week should be, for display purposes. It holds an integer value from 0 to 6, where 0 means Sunday, 1 means Monday, and so on. The default value is 0; that is, display all months with Sunday as the first day of the week. This variable is either part of the user configuration data, or in the locale bundle frame.
`useWeekNumber`	If non-`nil`, the Dates application displays the week number in the upper-left corner of its view. The first week of the year is number 1 and the last week is number 52. This variable is a slot in the locale bundle frame.

Table 16-3 Dates constants for the day of the week

Constant	Value	Description
`kSunday`	`0x00000800`	Sunday
`kMonday`	`0x00000400`	Monday
`kTuesday`	`0x00000200`	Tuesday
`kWednesday`	`0x00000100`	Wednesday
`kThursday`	`0x00000080`	Thursday
`kFriday`	`0x00000040`	Friday
`kSaturday`	`0x00000020`	Saturday
`kEveryday`	`0x00000FE0`	Every day in the week

Table 16-4 Dates constants for `repeatType`

Constant	Value	Description
`kDayOfWeek`	0	Meeting recurs on a specific week day of any week in the month.
`kWeekInMonth`	1	Meeting recurs in a specified week of the month.
`kDateInMonth`	2	Meeting recurs on a certain day of each month.
`kDateInYear`	3	Meeting recurs on a certain day of each year.
`kPeriod`	4	Meeting recurs on a specific day every two weeks.
`kNever`	5	Meeting does not recur.
	6	Reserved for internal use.
`kWeekInYear`	7	Meeting recurs in a specified week of the year.

Table 16-5 Other date constants

Constant	Value	Description
`kForever`	`0x1FFFFFFF`	A special value.
`kMaxyear`	2919	The largest year value handled.
`kYearMissing`	2920	The nearest leap year before `kForever`. The string parser uses it to indicate that the year is missing in the date string.

Table 16-6 Dates constants for the weeks in a month

Constant	Value	Description
`kFirstWeek`	`0x00000010`	The first week in the month.
`kSecondWeek`	`0x00000008`	The second week in the month.
`kThirdWeek`	`0x00000004`	The third week in the month.
`kFourthWeek`	`0x00000002`	The fourth week in the month.
`kLastWeek`	`0x00000001`	The last week in the month.
`kEveryWeek`	`0x0000001F`	Any week in the month.

Table 16-7 Compatible icon and meeting/event types

Icon type	Compatible meeting/event types
`'Meeting`	Any type of meeting, but not events.
`'WeeklyMeeting`	Only meetings that repeat on the same day each week.
`'Event`	Any type of event.
`'MultiDayEvent`	Only events that repeat every day.
`'AnnualEvent`	Only events that repeat on the same day each year.

Dates Protos

The Dates application uses the two protos `protoRepeatPicker` displays and `protoRepeatView` to allow the user to pick how a meeting or event repeats. You may wish to use these if you are creating your own meeting slip.

protoRepeatPicker

Based on `protoLabelPicker`, `protoRepeatPicker` displays a picker of repeating meeting types. This proto assumes the existence of a declared child of an ancestor of the `protoRepeatPicker` proto named `RepeatingView` that derives from `protoRepeatView`. Picking Other from the choice of repeating meeting types opens the `RepeatingView`.

The following graphic shows this picker after it's been tapped:

The `protoRepeatPicker` has the following slots:

Slot descriptions

`selectedMeeting`

Required. This slot must be in the template or inherited by the template. The slot's value is typically a meeting frame; see "Meeting Frames" (page 16-57). The `protoRepeatPicker` uses the `mtgStartDate` slot of the meeting, and the `repeatType`, `mtgInfo`, or `repeatTemplate` slots if they are present.

`originalMtgDate`

Required. If the `selectedMeeting` value is a repeating Meeting soup entry, this slot must contain the date of an instance of that repeating meeting; otherwise, the slot is ignored. The slot should be in the template or inherited by the template.

`newMtgDate` Required. The value of this slot must be the value of the `mtgStartDate` slot of the `selectedMeeting` slot. The slot should be in the template or inherited by the template.

`viewBounds` The bounds of the system prototype.

The `protoRepeatPicker` provides its own `ViewSetupFormScript` and `LabelActionScript` methods. If you override these methods, be sure to call `inherited:ViewSetupFormScript()` and `inherited:LabelActionScript()`.

In addition, `protoRepeatPicker` provides default `TextSetup` and `PickerSetup` methods.

protoRepeatView

This proto displays choices like those in the `protoRepeatPicker`, and a calendar that makes these choices more intuitive. The `protoRepeatView` is displayed when the user chooses Other from the `protoRepeatPicker`. A view that contains a `protoRepeatView` child should also have a `protoRepeatPicker`.

This is a `protoFloatNGo` system prototype that is displayed in its parent to create a draggable view. Its default justification is centered horizontally and flush with the top of the parent. The view is 204 pixels wide by 190 pixels high, so the parent should be at least that wide and high.

The following graphic shows a `protoRepeatView`:

The `protoRepeatView` has the following slots:

Slot descriptions

`viewFlags`	Defaults to `vClickable+vFloating`.
`viewFormat`	Defaults to `vfFillWhite + vfFrameDragger + vfPen(7) + vfInset(1) + vfRound(5)`.
`viewJustify`	Defaults to `vjParentCenterH`.
`viewBounds`	Defaults to `RelBounds(0, 0, 204, 190)`

This proto contains a single method of interest, `GetRepeatSpec`.

GetRepeatSpec

protoRepeatView`:GetRepeatSpec()`

Returns a `repeatTemplate` containing the repeat information. In particular, the `repeatTemplate` returned has the slots: `mtgStartDate`, `repeatType`, and `mtgInfo`.

If the `selectedMeeting` slot of `protoRepeatPicker` is a repeating meeting and if the repeat settings (in the `protoRepeatPicker` and `protoRepeatView`) have not changed, `GetRepeatSpec` returns the `repeatTemplate` of `selectedMeeting`.

If `selectedMeeting` is a repeating meeting and the repeat settings have been changed to “Don’t repeat,” `GetRepeatSpec` returns `nil`.

Dates Methods and Functions

This section includes a description of the methods supplied by the Dates application, which is called `calendar`. To get a reference to the Dates application, use the following code:

```
GetRoot().calendar
```

Note that future Newton devices may not include the Dates application. You should therefore check for the existence of the Dates application before trying to access it. Use the following code to test for this:

```
if GetRoot().calendar then ...
```

AddAppointment

calendar`:AddAppointment(`*mtgText*`, `*mtgStartDate*`, `*mtgDuration*`,` *repeatPeriod*`, `*repeatInfo*`)`

Creates a meeting and adds it to the appropriate Dates soup. It also updates the Dates display, if necessary.

mtgText	A string or rich string that is the meeting text.
mtgStartDate	An integer specifying the start date and time of the meeting, in the number of minutes passed since midnight, January 1, 1904. If the meeting repeats, this is the start date and time of its first occurrence.
mtgDuration	A positive integer specifying the duration of the meeting in minutes.

repeatPeriod
Used to indicate a repeating meeting. Specify one of the following symbols, or `nil`:

`nil`	Meeting is not repeating.
`'daily`	Meeting repeats daily.
`'weekly`	Meeting repeats weekly on the same day.
`'biweekly`	Meeting repeats biweekly on the same day.
`'monthly`	Meeting repeats monthly on the same day.
`'monthlyByWeek`	Meeting repeats monthly by the week in the month.
`'yearly`	Meeting repeats yearly on the same day.
`'yearlyByWeek`	Meeting repeats yearly by the week in the month.

repeatInfo
Used only if *repeatPeriod* is `'weekly`, `'monthlyByWeek`, or `'yearlyByWeek`, to specify when the meeting repeats. If not used, this parameter must be set to `nil`.

If *repeatPeriod* is `'weekly`, this parameter must be an array of one or more numbers between 0 and 6 (where 0 = Sunday, 1 = Monday, etc.) These numbers specify on which days of each week the meeting repeats. You can also specify `nil`, which means the meeting repeats on the same day in which it was originally scheduled.

If *repeatPeriod* is `'monthlyByWeek`, this parameter must be an array of one or more numbers between 1 and 5 (1 is the first week of each month, 2 is the second week of each month, and so on, up to 5, which is the last week of each month). These numbers specify in which weeks each month the meeting repeats. You can also specify `nil`, which means the meeting repeats on the same week in which it was originally scheduled.

If *repeatPeriod* is `'yearlyByWeek`, you can usually specify `nil`, since the week in the month is predetermined by the date you pick for the first instance of the meeting. However, if the day falls during the fourth or fifth week of a month, it is not always possible to determine in exactly which week subsequent instances of the meeting will occur. In this case, you should specify an array containing the single number 4 or 5, to indicate the week. For example, November 1994 had only four Thursdays, so Thursday, November 24th, 1994 could be interpreted as the fourth Thursday or as the last Thursday.

This method returns the soup entry added for the new meeting.

AddEvent

calendar`:AddEvent(`*mtgText*`,` *mtgStartDate*`,` *repeatPeriod*`,` *repeatInfo*`)`

Creates an event and adds it to the appropriate Dates soup. It also updates the Dates display, if necessary.

mtgText — A string or rich string that is the event text.

mtgStartDate — An integer specifying the start date of the event, in the number of minutes passed since midnight, January 1, 1904. If the event repeats, this is the start date of its first occurrence. Note that events don't have a specific time during the day, so by convention, they must always be created at midnight. The Dates application expects this, so don't create events at other times.

`AddEvent` automatically sets *mtgStartDate* to midnight at the beginning of the day.

repeatPeriod
: Used to indicate a repeating event. Specify one of the following symbols, or `nil`:

`nil`	Event is not repeating.
`'daily`	Event repeats daily.
`'weekly`	Event repeats weekly on the same day.
`'biweekly`	Event repeats biweekly on the same day.
`'monthly`	Event repeats monthly on the same day.
`'monthlyByWeek`	Event repeats monthly by the week in the month.
`'yearly`	Event repeats yearly on the same day.
`'yearlyByWeek`	Event repeats yearly by the week in the month.

repeatInfo
: Used only if *repeatPeriod* is `'weekly`, `'monthlyByWeek`, or `'yearlyByWeek`, to specify when the event repeats. If not used, this parameter must be set to `nil`.

If *repeatPeriod* is `'weekly`, this parameter must be an array of one or more numbers between 0 and 6 (where 0 = Sunday, 1 = Monday, etc.) These numbers specify on which days each week the event repeats. You can also specify `nil`, which means the event repeats on the same day on which it was originally scheduled.

If *repeatPeriod* is `'monthlyByWeek`, this parameter must be an array of one or more numbers between 1 and 5 (1 is the first week of each month, 2 is the second week of each month, and so on, up to 5, which is the last week of each month). These numbers specify in which weeks each month or year the event repeats. You can also specify `nil`, which means the event repeats in the same week in which it was originally scheduled.

If *repeatPeriod* is `'yearlyByWeek`, you can usually specify `nil`, since the week in the month is predetermined by the date you pick for the first instance of the event. However, if the day falls during the fourth or fifth week of a month, it is not always possible to determine in exactly which week subsequent instances of the event will occur. In this case, you should specify an array containing the single number 4 or 5 to indicate the week.

This method returns the soup entry added for the new event.

Here is an example:

```
GetRoot().calendar:AddEvent("Mother's Day",
StringToDate("5/14/95 12:00am"), 'yearlyByWeek, nil)
```

DeleteAppointment

calendar`:DeleteAppointment(`*mtgTextOrFrame*`, ` *mtgStartDate*`,` *deleteOneOnly*`)`

Finds the meeting(s) at the given date and time, with the given meeting text, and deletes them all. If an instance of a repeating meeting is found, only that single instance is deleted. If a meeting frame is passed as a parameter, the method ignores the other parameters and deletes that meeting frame.

This method updates the Dates display, if necessary.

mtgTextOrFrame — Either a string or rich string that is the meeting text of the meeting you want to delete, or a meeting frame. A meeting frame can be either a soup entry that contains a meeting frame; see “Dates Soup Formats” (page 16-56), or the frame returned by the `FindAppointment` or `FindExactlyOneAppointment` method.

mtgStartDate — An integer specifying the start date and time of the meeting, in the number of minutes passed since midnight, January 1, 1904. If *mtgTextOrFrame* is a meeting frame, the value of *mtgStartDate* is ignored.

deleteOneOnly — A Boolean value that specifies whether to delete just one or multiple meetings (if multiple meetings are found). If you specify `nil` and more than one meeting is found, all found meetings are deleted. If you specify `true` and more than one meeting is found, this method throws an exception.

This method's return value is unspecified.

DeleteRepeatingEntry

calendar:`DeleteRepeatingEntry(`*mtgTextOrFrame*, *mtgStartDate*, *deleteOneOnly*`)`

Finds the repeating meeting(s) or event(s) at the given date and time, with the given meeting text, and deletes them all. All instances of the repeating meeting/event are deleted, not just the instance at the given time and date. If a meeting frame is passed as a parameter, the method ignores the other parameters and deletes that meeting frame. This method also updates the Dates display, if necessary.

mtgTextOrFrame — Either a string or rich string that is the meeting text of the repeating meeting or event you want to delete, or a meeting frame. A meeting frame can be either a soup entry that contains a meeting frame; see "Dates Soup Formats" (page 16-56), or the frame returned by the `FindAppointment` or `FindExactlyOneAppointment` method.

mtgStartDate — An integer specifying the start date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904. Note that events don't have a specific time during the day, so this method finds all events scheduled during the day of *mtgStartDate*.

deleteOneOnly — A Boolean value that specifies whether to delete just one or multiple meetings/events (if multiple meetings or events are found). If you specify `nil` and more than one meeting or event is found, all found meetings/events are deleted. If you specify `true` and more than one meeting or event is found, this method throws an exception.

This method's return value is unspecified.

DeleteEvent

calendar:`DeleteEvent(`*mtgTextOrFrame*`,` *mtgStartDate*`,` *deleteOneOnly*`)`

Finds the event(s) on the given date and time, with the given text, and deletes them all. If an instance of a repeating event is found, only that single instance is deleted. If a meeting frame is passed as a parameter, the method ignores the other parameters and deletes that meeting frame. This method updates the Dates display, if necessary.

mtgTextOrFrame Either a string or rich string that is the meeting text of the event you want to delete, or a meeting frame. A meeting frame can be either a soup entry that contains a meeting frame; see "Dates Soup Formats" (page 16-56), or the frame returned by the `FindAppointment` or `FindExactlyOneAppointment` method.

mtgStartDate An integer specifying the start date and time of the event, in the number of minutes passed since midnight, January 1, 1904. Note that events don't have a specific time during the day, so this method finds all events scheduled during the day of *mtgStartDate*.

deleteOneOnly A Boolean value that specifies whether to delete just one or multiple events (if multiple events are found). If you specify `nil` and more than one event is found, all found events are deleted. If you specify `true` and more than one event is found, this method throws an exception.

This method's return value is unspecified.

DisplayDate

calendar:`DisplayDate(`*date*, *format*`)`

Displays the Dates meetings, To Do List items, or the agenda for the specified date. Executing this method is equivalent to the user tapping that date in the month view. Note that this method is meant to be called only when the calendar is open.

date — An integer specifying a date, in the number of minutes passed since midnight, January 1, 1904.

format — A symbol specifying what is to be displayed, as follows:

`'Day`	The day view.
`'ToDoList`	The To Do List.
`'Agenda`	The day's agenda.
`nil`	The calendar continues showing the current view, after closing any overviews.

These views are equivalent to the similarly named views listed on the Show button picker in the Dates application.

This method's return value is unspecified.

FindAppointment

calendar:`FindAppointment(`*mtgText*, *findWords*, *dateRange*, *type*, *maxNumberToFind*`)`

Finds one or more meetings and/or events using the appointment title, specific words, and/or a date range as search criteria. Note that if multiple instances of a repeating meeting or event match the search criteria, all those instances are returned.

mtgText — A string or rich string that is the meeting text or event text of the item(s) you want to find. Specify `nil` to match all entries satisfying the other search criteria (*findWords*, *dateRange*, and *type*).

findWords	An array of words or word beginnings (specified as strings) used as search criteria to find meetings or events. If the text of the meeting or event, or of the notes, contains all the words in this array, it satisfies this criterion. The words in the array can be split between the meeting text and the meeting notes. The word search is not case sensitive, and it searches only word beginnings; it will not find a string that occurs inside a word. Specify `nil` if you do not want to use this search criterion.
dateRange	A single time, an array of two times, or `nil`. A time is specified as the number of minutes passed since midnight, January 1, 1904. If you specify a single time, all meetings scheduled at that time satisfy the search criteria. In the case of events, all events on the day containing that time satisfy the search criteria. If you specify an array of two times, all meetings and events between the two times satisfy the search criteria. Specify `nil` if you do want not use this search criterion.
type	Used to limit the found items to meetings, repeating meetings, events, or repeating events. Specify one of the following symbols, an array of these symbols (to include multiple types), or `nil`:

`nil`	This search criterion is not used.
`'Meeting`	Search for nonrepeating meetings.
`'Event`	Search for nonrepeating events.
`'RepeatingMeeting`	Search for repeating meetings.
`'RepeatingEvent`	Search for repeating events.

maxNumberToFind
: An integer that specifies the maximum number of items to find. After this number of items is found, this method stops searching and returns the results. Specify `nil` to use the default value of 50.

This method returns an array of the resulting meeting frames for the meetings and events. The meeting frames are soup entries in the case of nonrepeating meetings and events, as described in "Meeting Frames" (page 16-57). In the case of repeating meetings or events, the meeting frame returned has the following slots:

Slot description

`viewStationery`	The symbol `'RepeatingMeeting` for a repeating meeting, or `'CribNote` for a repeating event.
`class`	The symbol `'meeting`.
`mtgStartDate`	The date and time of this instance of the repeating meeting in the number of minutes passed since midnight, January 1, 1904.
`repeatTemplate`	The soup entry for the defining instance of the repeating meeting or event.

Note that even though the two senses of "meeting frame" are different for repeating and nonrepeating meetings or events, methods that expect a meeting frame accept either type.

FindExactlyOneAppointment

calendar`:FindExactlyOneAppointment(`*mtgText*`, `*findWords*`, `*dateRange*`, `*type*`)`

Finds and returns exactly one meeting or event using specific words and/or a date range as search criteria.

mtgText
: A string or rich string that is the meeting text or event text of the item you want to find. Specify `nil` to match an entry satisfying the other search criteria (*findWords*, *dateRange*, and *type*).

findWords — An array of words or word beginnings (specified as strings) used as search criteria to find meetings or events. If the text of the meeting or event, or of the notes, contains all the words in this array, it satisfies these criteria. The words in the array can be split between the meeting text and the meeting notes. The word search is not case sensitive, and it searches only word beginnings; it will not find a string that occurs inside a word. Specify `nil` if you do want not use this search criterion.

dateRange — A single time, an array of two times, or `nil`. A time is specified as the number of minutes passed since midnight, January 1, 1904.

If you specify a single time, all meetings scheduled at that time will satisfy the search criteria. In the case of events, all events on the day containing that time satisfy the search criteria.

If you specify an array of two times, all meetings and events between the two times satisfy the search criteria.

Specify `nil` if you do want not use this search criterion.

type — Limits the found item to a meeting, repeating meeting, event, or repeating event. Specify one of the following symbols, an array of these symbols (to include multiple types), or `nil`:

`nil` — This search criterion is not used.

`'Meeting` — Search for a nonrepeating meeting.

`'Event` — Search for a nonrepeating event.

`'RepeatingMeeting` — Search for a repeating meeting.

`'RepeatingEvent` — Search for a repeating event.

This method returns the meeting frame for the meeting or event. The meeting frame is a soup entry in the case of a nonrepeating meeting or event.

In the case of an instance of a repeating meeting or event the meeting frame returned is as described in `FindAppointment` (page 16-37).

If no entry is found, or more than one is found, this method throws an exception (error –48418).

FindNextMeeting

calendar`:FindNextMeeting(`*date*`)`

Finds the first meeting after the specified date and time. This method might be useful for an application that wants to find open time in the calendar for scheduling a meeting.

date	An integer specifying a date and time, in the number of minutes passed since midnight, January 1, 1904.

This method returns an array containing the meeting start date as the first element and its duration as the second element. If there is more than one meeting scheduled at that time, the duration of the longest one is returned. If there are no meetings scheduled after the specified date, `nil` is returned.

GetMeetingIconType

calendar`:GetMeetingIconType(`*mtgTextOFrame*`, `*mtgStartDate*`)`

Returns the type of icon used for a particular meeting or event. The following symbols can be returned: `'Event`, `'Meeting`, `'WeeklyMeeting`, `'MultiDayEvent`, or `'AnnualEvent`.

mtgTextOrFrame	Either a string or rich string that is the meeting text of the meeting whose icon you want, or a meeting frame. A meeting frame can be either a soup entry that contains a meeting frame; see "Dates Soup Formats" (page 16-56), or the frame returned by the `FindAppointment` or `FindExactlyOneAppointment` method.
mtgStartDate	An integer specifying the start date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904. Note that events don't have a

specific time during the day, so this method finds an event scheduled any time during the day of *mtgStartDate*.

If the meeting or event uses a custom meeting type defined by `RegMeetingType` (page 16-48), that type is not returned by `GetMeetingIconType`. `GetMeetingIconType` returns either `'Meeting` or `'Event`, depending on whether the custom meeting type is created using `AddAppointment` (page 16-30) or `AddEvent` (page 16-32). You must look at the `meetingType` slot of the custom meeting or event to determine the unique custom meeting type defined by `RegMeetingType` (page 16-48).

GetCalendarMeetingType

`GetCalendarMeetingType() //platform file function`

Returns an array of all meeting types registered with the Dates application. This array includes all the built-in meeting types listed in Table 16-7 as well as any custom meeting types created by a call to `RegMeetingType` (page 16-48).

IMPORTANT

This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

```
call kGetCalendarMeetingTypeFunc with ();
```

▲

GetCalendarMeetingTypeInfo

`GetCalendarMeetingTypeInfo(`*typeSymbol*`) //platform file function`

Returns a frame containing information about the meeting type represented by *typeSymbol*, or `nil` if *typeSymbol* has no associated meeting type.

IMPORTANT

This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

call kGetCalendarMeetingTypeInfoFunc with (*typeSymbol*);

▲

typeSymbol	A symbol associated with a meeting type, as returned by `GetCalendarMeetingType` (page 16-42).

This function returns a frame with the following slots:

Slot description

`label`	A string that is the text displayed in the New menu for this meeting type.
`icon`	A bitmap frame (of the kind returned by `GetPictAsBits`) containing the bitmap displayed in the New menu for this meeting type.
`smallIcon`	A bitmap frame containing the bitmap displayed in the meeting slip for this meeting type.
`shape`	A shape object containing the `icon` bitmap.
`memory`	A symbol under which the most recently used meeting title strings are stored. (These are stored and accessed using the functions `AddMemoryItem` and `GetMemoryItems`.)

GetMeetingInvitees

calendar:`GetMeetingInvitees(`*mtgText*`,` *mtgStartDate*`)`

Returns the list of invitees for a meeting, or returns `nil` if there are none.

mtgText	A string or rich string that is the meeting text of the meeting for which you want to get the list of invitees.
mtgStartDate	An integer specifying the start date and time of the meeting, in the number of minutes passed since midnight, January 1, 1904.

The list returned is an array of name reference frames; these frames are described in “Name References” (page 5-1) in *Newton Programmer’s Reference*.

GetMeetingLocation

calendar`:GetMeetingLocation(`*mtgText*`, `*mtgStartDate*`)`

Returns the location for a meeting, or returns `nil` if there is none.

mtgText A string or rich string that is the meeting text of the meeting for which you want to get the location.

mtgStartDate An integer specifying the start date and time of the meeting, in the number of minutes passed since midnight, January 1, 1904.

The meeting location is returned as a name reference frame; these frames are described in “Name References” (page 5-1) in *Newton Programmer’s Reference*.

GetMeetingNotes

calendar`:GetMeetingNotes(`*mtgText*`, `*mtgStartDate*`)`

Returns the notes for a meeting or event, or returns `nil` if there are none.

mtgText A string or rich string that is the meeting text of the meeting or event for which you want to get the notes.

mtgStartDate An integer specifying the start date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904.

The notes are returned as an array of frames with the same format as the `data` slot of a Notes soup entry; see “Notes Soup Format” (page 16-82).

GetSelectedDates

calendar`:GetSelectedDates()`

Returns an array of the currently selected and displayed dates. This array always has at least one element. If the Dates application is closed, the method returns `nil`.

These dates are integers specifying a date and time, in the number of minutes passed since midnight, January 1, 1904. Note that the time for each date in the array is set to midnight at the beginning of the day.

MoveAppointment

calendar:`MoveAppointment(`*mtgText*`,` *mtgStartDate*`,` *newStartDate*`,` *newDuration*`)`

Finds the unique meeting or event with the given text at the given date and time, and changes the start date and/or the meeting duration.

mtgText	A string or rich string that is the meeting text of the meeting or event you want to move.
mtgStartDate	An integer specifying the start date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904. Note that events don't have a specific time during the day, so this method finds an event scheduled at any time during the day of *mtgStartDate*.
newStartDate	An integer specifying the new start date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904. If `nil`, the start date and time remain unchanged.
newDuration	A positive integer specifying the new duration of the meeting in minutes. If `nil`, the duration is unchanged. If the entry is an event, specify `nil`.

This method's return value is unspecified.

If you specify a repeating meeting or event that is not an exception case, this method changes the start time and duration of all repeating instances of the meeting or event that are not exceptions. However, if *newStartDate* is not a day that matches the original repeating pattern, this method changes *newStartDate* to the first day after the one specified that does match the repeating pattern.

For example, if the repeating meeting normally occurs on Tuesdays and Thursdays and you specify a *newStartDate* that is on a Friday, this method changes the *newStartDate* to the following Tuesday.

If no meeting or event is found, or more than one is found, this method throws an exception.

MoveOnlyOneAppointment

calendar`:MoveOnlyOneAppointment(`*mtgText*`, `*mtgStartDate*`,` *newStartDate*`, `*newDuration*`)`

Finds the unique meeting or event with the given text at the given date and time, and changes the start date and/or the meeting duration.

mtgText	A string or rich string that is the meeting text of the meeting or event you want to move.
mtgStartDate	An integer specifying the start date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904. Note that events don't have a specific time during the day, so this method finds an event scheduled at any time during the day of *mtgStartDate*.
newStartDate	An integer specifying the new start date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904. If `nil`, the start date and time remain unchanged.
newDuration	A positive integer specifying the new duration of the meeting in minutes. If `nil`, the duration is unchanged. If the entry is an event, specify `nil`.

This method's return value is unspecified.

If you specify a repeating meeting or event that is not an exception case, this method changes it to an exception case and applies the new start time and duration to the new exception.

If no meeting or event is found, or more than one is found, then this method throws an exception.

OpenMeetingSlip

calendar:`OpenMeetingSlip(`*meetingFrame*`,` *date*`,` *openDefaultSlip*`)`

Opens the meeting slip for the specified meeting or event.

meetingFrame — Either a soup entry that contains a meeting frame; see "Dates Soup Formats" (page 16-56), or the frame returned by the `FindAppointment` (page 16-37) or `FindExactlyOneAppointment` (page 16-39) method.

If this parameter is a soup entry for a repeating meeting or event, also specify the *date* parameter.

date — Used only if *meetingFrame* is a repeating meeting or event. This parameter must be the date and time of a particular instance of the repeating meeting or event.

openDefaultSlip — A Boolean. Set to `true` to cause the Dates application to open the default meeting slip for the meeting or event. Set to `nil` to cause the Dates application to send the `OpenMeeting` (page 16-50) message to the frame registered for this meeting type, if there is one.

This method's return value is unspecified.

RegInfoItem

calendar:`RegInfoItem(`*symbol*`,` *frame*`)`

Adds an item to the end of the Info button picker in the base view of the Dates application.

symbol — A unique symbol identifying the item. Use your developer signature in this symbol.

frame — A frame containing two slots:

`item`	A string or a bitmap frame. This is the item to display in the picker.
`DoAction`	This method is called if the user picks this item. It is passed no parameters.

Items added by this method are not persistent across a system restart.

This method's return value is unspecified.

To remove the item added by this method, call `UnRegInfoItem` (page 16-56).

RegMeetingType

calendar`:RegMeetingType(`*symbol*`, `*frame*`)`

Registers a new meeting type for the Dates application. The meeting type appears in the New picker of the Dates application.

symbol	A unique symbol identifying the meeting type. Use your developer signature in this symbol.
frame	A frame describing the new meeting type. The frame slots are described below.

The slots in *frame* are as follows:

Slot description

`item`	Required. A string that is the meeting-type name to appear in the picker.
`icon`	Required. A bitmap frame containing a bitmap to appear next to the name in the picker. This bitmap is also displayed in the New pickers, the day view, agenda view, and overview. The bitmap must be no larger than 24 pixels wide by 15 pixels high.
`NewMeeting`	Required. A function called if the user chooses this meeting type in the New picker. See the description of the `NewMeeting` method (page 16-49).
`smallicon`	Optional. A bitmap frame containing a bitmap to be displayed in the meeting slip. The bitmap must be no more than 12 pixels high. If this slot is not included, the icon in `icon` is used, which looks unattractive.
`memory`	Optional. A symbol identifying the system storage location for previous meeting titles of this new meeting type. When the user makes a new meeting of this type, the Title picker in the meeting slip lists previous meeting titles of this meeting type as a convenience. If

	you don't provide this slot, the Dates application uses the storage location allocated for the underlying meeting type (that is, as a meeting).
`OpenMeeting`	Optional. A function called if the user taps the icon of a meeting or event whose `meetingType` slot matches the symbol under which this meeting type was registered; see `OpenMeeting` (page 16-50).

Meeting types added by `RegMeetingType` are not persistent across a system restart.

To remove the meeting type added by `RegMeetingType`, call `UnRegMeetingType` (page 16-56).

This method's return value is unspecified.

NewMeeting

myMeetingType`:NewMeeting(`*date*`, `*parentBox*`)`

Called if the user chooses this meeting type in the New picker. It is a method of the frame registered with `RegMeetingType` (page 16-48).

This method must create a meeting (or event) using `AddAppointment` (or `AddEvent`), and must add a slot called `meetingType` to the appointment created. This slot must be set to the symbol that identified the meeting type in the call to `RegMeetingType` (page 16-48). Remember to call `EntryChange` to save this new slot.

date	This parameter is the current date displayed by the Dates application.
parentBox	This parameter is the global `viewBounds` of the calendar base view.

If `NewMeeting` returns the new meeting frame, the Dates application performs the default action, which is to open the default meeting slip. If this method returns `nil`, the Dates application does nothing and this method should perform any necessary actions.

If this method opens its own meeting slip or other view, it should do so by using the Dates method `RememberedOpen` (page 16-51). That method opens the view and records it so that the Dates application can close the view if the Dates application is closed.

If you are creating a custom meeting slip, you may want to use the `protoRepeatPicker` and `protoRepeatView` protos.

OpenMeeting

myMeetingType`:OpenMeeting(`*meeting*`, `*date*`, `*parentBox*`)`

Called if the user taps the icon of a meeting or event that has a `meetingType` slot that matches the symbol under which this meeting type was registered. It is a method of the frame registered with `RegMeetingType`.

meeting	The soup entry for the tapped item.
date	The date and time of the meeting in number of minutes since midnight, January 1, 1904.
parentBox	The global `viewBounds` of the calendar base view.

If `OpenMeeting` returns a non-`nil` value, the Dates application performs the default action, which is to open the default meeting slip. If this method returns `nil`, the Dates application does nothing and this method should perform any actions necessary.

If this method opens its own meeting slip or other view, it should do so by using the Dates method `RememberedOpen` (page 16-51). That method opens the view and records it so that the Dates application can close the view if the calendar is closed.

RememberedClose

calendar`:RememberedClose(`*view*`)`

Closes a view in the calendar that was opened with `RememberedOpen`. If the view is closed without calling this method, the Dates application keeps a reference to the view until the calendar is closed, which wastes memory.

view	The view to close.

This method's return value is unspecified.

This method should be used to close views opened by `RememberedOpen` from within the `OpenMeeting` (page 16-50) method of `RegMeetingType` (page 16-48) and from within the `DoAction` method of `RegInfoItem` (page 16-47).

RememberedOpen

calendar`:RememberedOpen(`*view*`)`

Opens a view in the Dates application and records it so that if the Dates application is closed, that view is also closed.

view	The view to open.

This method's return value is unspecified.

This method should be used to open views from within the `OpenMeeting` (page 16-50) method of `RegMeetingType` (page 16-48) and from within the `DoAction` method of `RegInfoItem` (page 16-47).

Views opened by `RememberedOpen` should be closed by `RememberedClose`.

SetEntryAlarm

calendar`:SetEntryAlarm(`*mtgText*`,`*mtgStartDate*`,`*minutesBefore*`)`

Sets an alarm on the meeting with the given text at the given date and time. If the meeting is an instance of a repeating meeting, the alarm is set for all instances of the repeating meeting.

mtgText	A string or rich string that is the meeting text of the meeting for which you want to set the alarm time.
mtgStartDate	An integer specifying the start date and time of the meeting, in the number of minutes passed since midnight, January 1, 1904.
minutesBefore	A non-negative integer, which specifies how far in advance of the meeting or event the alarm should go off. A value of 0 means the alarm goes off at the time of the meeting. This integer should specify the number of minutes before *mtgStartDate* that you want the alarm to go off.

You can specify `nil` to clear an alarm that is currently set.

This method's return value is unspecified.

SetMeetingIconType

calendar`:SetMeetingIconType(`*mtgText*`,` *mtgStartDate*`,` *newIconType*`)`

Finds a particular meeting or event and sets its icon type. If the item found is an instance of a repeating meeting or event, the icon type is changed for all instances in that repeating series.

mtgText — A string or rich string that is the meeting text of the meeting or event for which you want to set the icon type.

mtgStartDate — An integer specifying the start date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904. Note that events don't have a specific time during the day, so this method finds an event scheduled at any time during the day of *mtgStartDate*.

newIconType — A symbol specifying the new icon type to set for the meeting or event. You can specify the following icon types: `'Event`, `'Meeting`, `'WeeklyMeeting`, `'MultiDayEvent`, or `'AnnualEvent`.

This method's return value is unspecified.

If the new icon type is incompatible with the type of the meeting or event, then this method throws an exception. Table 16-7 shows the icon types and the meeting and event types with which they are compatible.

SetMeetingInvitees

calendar`:SetMeetingInvitees(`*mtgText*`,` *mtgStartDate*`,` *invitees*`)`

Sets list of invitees for the meeting specified by *mtgText* and *mtgStartDate*.

mtgText A string or rich string that is the meeting text of the meeting for which you want to set the list of invitees.

mtgStartDate An integer specifying the start date and time of the meeting, in the number of minutes passed since midnight, January 1, 1904.

invitees An array specifying the invitees to set, or `nil` to clear the location for the meeting. The array can contain any combination of the following four objects:

- a name reference; see "Name References" (page 5-1) in *Newton Programmer's Reference*.
- a Names soup entry
- an alias to a Names soup entry
- a frame containing first and last name strings, with this format:

`{name: {first:` *string*`,`
`        last:` *string*`}}`

In this frame, you can specify the empty string, or `nil`, or leave the slot out if the first or last name is missing.

This method's return value is unspecified.

If the specified meeting is a repeating meeting and not an exception meeting, this method sets the list of invitees for all meetings in the repeating series. If the specified meeting is a repeating meeting exception, the list of invitees applies to that exception meeting only.

SetMeetingLocation

calendar`:SetMeetingLocation(`*mtgText*`,` *mtgStartDate*`,` *location*`)`

Sets the location for the meeting specified by *mtgText* and *mtgStartDate*.

mtgText A string or rich string that is the meeting text of the meeting for which you want to set the location.

mtgStartDate An integer specifying the start date and time of the meeting, in the number of minutes passed since midnight, January 1, 1904.

location Specifies the location to be set. This parameter must be one of the following objects:

- a name reference; see "Name References" (page 5-1) in *Newton Programmer's Reference*.
- a Names soup entry
- an alias to a Names soup entry
- a string (write-in location)
- `nil`, to clear the location for the meeting

This method's return value is unspecified.

If the specified meeting is a repeating meeting and not an exception meeting, this method sets the location for all meetings in the repeating series. If the specified meeting is a repeating meeting exception, the location applies to that exception meeting only.

SetMeetingNotes

calendar`:SetMeetingNotes(`*mtgText*`, `*mtgStartDate*`, `*notes*`)`

Sets the notes for a meeting or event specified by *mtgText* and *mtgStartDate*. The new notes replace all existing notes.

mtgText A string or rich string that is the meeting text of the meeting or event for which you want to set the notes.

mtgStartDate An integer specifying the start date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904. Note that events don't have a specific time during the day, so this method finds an event scheduled any time during the day specified by *mtgStartDate*.

notes Specifies the notes to set. This parameter must be one of the following objects:

- a string
- an array of frames with the same format as the `data` slot of a Notes soup entry; see "Notes Soup Format" (page 16-82)
- `nil`, to clear the notes for the meeting

This method's return value is unspecified.

If the specified meeting is a repeating meeting, this method sets the notes for only the particular instance of the meeting identified by *mtgStartDate*.

SetRepeatingEntryStopDate

calendar`:SetRepeatingEntryStopDate(`*mtgText*`, `*mtgStartDate*`,` *mtgStopDate*`)`

Sets the stop date for the repeating meeting or event with the given text at the given date and time.

mtgText A string or rich string that is the meeting text of the repeating meeting or event for which you want to set the stop time.

mtgStartDate An integer specifying the start date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904. Note that events don't have a specific time during the day, so this method finds an event scheduled at any time during the day of *mtgStartDate*.

mtgStopDate An integer specifying the stop date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904. The stop date is the date after which the meeting or event will no longer repeat. If you specify `nil`, the meeting or event repeats forever.

This method's return value is unspecified.

UnRegInfoItem

calendar`:UnRegInfoItem(`*symbol*`)`

Removes an item previously added by `RegInfoItem` (page 16-47) to the Info picker in the base view of the Dates application.

symbol The symbol used to identify the item in the `RegInfoItem` method that added the item.

This method's return value is unspecified.

UnRegMeetingType

calendar`:UnRegMeetingType(`*symbol*`)`

Removes a meeting type previously added by `RegMeetingType` (page 16-48) to the New picker in the base view of the Dates application.

symbol The symbol used to identify the item in the `RegMeetingType` method that registered the item.

This method's return value is unspecified.

Dates Soup Formats

This section describes the format of entries in the Dates soups, which consist of either meeting frames or notes frames. The slots contained in these entry frames are described in "Meeting Frames" (page 16-57) and "Notes Frames" (page 16-62). There are four soups managed by the Dates application:

Soup (name string) description

`ROM_CalendarSoupName` ("Calendar")

Entries are meeting frames (page 16-57) for nonrepeating meetings.

`ROM_RepeatMeetingName` ("Repeat Meetings")

Entries are meeting frames (page 16-57) for repeating meetings and notes frames (page 16-62) for notes associated with specific instances of a repeating meeting. A single meeting frame entry describes all instances of a repeating meeting.

`ROM_CalendarNotesName` ("Calendar Notes")
: Entries are meeting frames (page 16-57) for nonrepeating events.

`ROM_RepeatNotesName` ("Repeat Notes")
: Entries are meeting frames (page 16-57) for repeating events and notes frames (page 16-62) for notes associated with specific instances of a repeating event.

Meeting Frames

Each meeting frame contains the following required slots:

Slot descriptions

`viewStationery`
: In the Calendar soup this slot always contains the value `'Meeting` for a meeting. In the Repeat Meetings soup it always contains the value `'repeatingMeeting`. In the Calendar Notes and Repeat Notes soups it always contains the value `'CribNote`.

`mtgStartDate`
: Contains an immediate value: the start date of the meeting (or date the event was entered) in the number of minutes passed since midnight, January 1, 1904. For events the time is midnight at the beginning of the day.

`mtgDuration`
: Contains an integer; the duration of the meeting in minutes. For events this value is meaningless and should be set to zero.

`mtgText`
: A rich string containing the meeting or event text. A rich string is used because the user can enter ink for the text of the meeting or event.

A meeting frame may also contain the following optional slots:

Slot descriptions

`mtgStopDate`
: Used for repeating meetings and events. An immediate value: the date that the meeting should stop repeating, in the number of minutes passed since midnight, January 1, 1904.

`repeatType` — Used for repeating meetings and events. Contains one of the following constants that describes how often the meeting repeats: `kDayofWeek` (0), `kWeekInMonth` (1), `kDateInMonth` (2), `kDateInYear` (3), `kPeriod` (4), `kNever` (5), `kWeekInYear` (7).

`mtgInfo` — Used for repeating meetings and events. An immediate value containing packed repeating meeting information. This slot is interpreted differently, depending on the value of the `repeatType` slot, as follows:

`repeatType` = `kDayOfWeek`

`mtgInfo` is set to any combination of constants from the following two groups added together:

Constants for day of week

`kSunday`	`0x00000800`
`kMonday`	`0x00000400`
`kTuesday`	`0x00000200`
`kWednesday`	`0x00000100`
`kThursday`	`0x00000080`
`kFriday`	`0x00000040`
`kSaturday`	`0x00000020`
`kEveryday`	`0x00000FE0`

Constants for week in month

`kFirstWeek`	`0x00000010`
`kSecondWeek`	`0x00000008`
`kThirdWeek`	`0x00000004`
`kFourthWeek`	`0x00000002`
`kLastWeek`	`0x00000001`
`kEveryWeek`	`0x0000001F`

`repeatType` = `kWeekInMonth`
: `mtgInfo` is set to a single constant from the first group above, added to any combination from the second group.

`repeatType` = `kDateInMonth`
: `mtgInfo` is set to the date in the month on which the meeting or event is to repeat.

`repeatType` = `kDateInYear`
: `mtgInfo` is set to (*month*<<8) + *date*, where *month* is the number of the month in the year (January = 1) and *date* is the date in the month on which the meeting is to repeat.

`repeatType` = `kPeriod`
: `mtgInfo` is set to (*mtgDay*<<8) + *period*, where *mtgDay* is the date, measured in days, of the meeting. This is the same as `mtgStartDate`, but in days, instead of minutes—that is, more simply, `mtgStartDate DIV 1440`. *period* is the number of days between meetings. Technically, *period* can range between 1 and 255; however, the current Newton user interface allows the user to choose only every other week (14 days) for this kind of meeting. Opening a `kPeriod` meeting always displays it as an "Every other week" meeting type and resets its *period* to 14.

`repeatType = kWeekInYear`
: `mtgInfo` is set to (*month*`<<`12) plus a single constant from the day-of-week constants (for example, `kThursday`) plus a single constant from the week-in-month constants (for example, `kThirdWeek`).

IMPORTANT

`mtgInfo` uses only the least-significant 24 bits of the integer. The remaining bits are reserved for future expansion. Always be sure to mask out the upper bits so a future change in format will not overflow your values. ▲

`mtgAlarm` Contains an immediate value. For single (nonrepeating) meetings or events, this value is the time when the alarm should occur. This value is represented as the number of minutes passed since midnight, January 1, 1904. For repeating meetings or events, this slot contains the number of minutes before the meeting or event at which the alarm should go off.

`mtgIconType` This slot determines what kind of icon to display for this meeting or event and what kind of slip to display when the user taps the meeting marker. The valid values are `'Meeting`, `'WeeklyMeeting`, `'Event`, `'MultiDayEvent`, and `'AnnualEvent`. If the slot doesn't exist, or is `nil`, the icon defaults to the Meeting icon or the Event icon.

`mtgInvitees` If the meeting frame is for a meeting, not for an event, this slot contains an array listing the invitees to the meeting. The elements in the array are name references; see "Name References" (page 5-1) in *Newton Programmer's Reference*. If the meeting has no invitees specified, `nil` is stored in the slot. Use the methods `GetMeetingInvitees` (page 16-43) and `SetMeetingInvitees` (page 16-52) to read or write this slot.

`mtgLocation` If the meeting frame is for a meeting, not for an event, this slot stores the meeting location as a name reference; see "Name References" (page 5-1) in *Newton Programmer's Reference*. If the meeting has no location specified, `nil` is stored in the slot. Use the methods `GetMeetingLocation` (page 16-44) and `SetMeetingLocation` (page 16-53) to read or write this slot.

`notesData` Contains an array of meeting (or event) note objects for nonrepeating meetings and events. Meeting notes are the notes visible when the user taps the Add notes or Edit notes button in a meeting slip to open its notes view. Meeting notes can consist of text objects, polygons, or ink objects. These objects have the same format as data objects in the Notes soup. For information on the format of these objects, see the description of the `data` slot in "Notes Soup Format" (page 16-82). Text objects have, in addition, a `viewFont` slot specifying the font of the text.

Notes for repeating meetings and events are stored in the `instanceNotesData` slot.

`instanceNotesData`

Contains an array of aliases to notes for instances of repeating meetings and events that have notes. Each instance's notes are stored as a separate soup entry in the Repeat Meetings soup containing the repeating meeting, or in the Repeat Notes soup containing the repeating event.

The `instanceNotesData` slot is an array of pairs. Each pair is an array of two elements: `[`*time*, *notesAlias*`]`. The first element, *time*, is the date and time of the meeting or event instance. The second element, *notesAlias*, is an alias to another entry in the same soup, that entry contains the actual notes for that instance. For a description of the format of a note entry see "Notes Frames" (page 16-62). For information on entry aliases, see Chapter 11, "Data Storage and Retrieval," in *Newton Programmer's Guide*.

Use the methods `GetMeetingNotes` (page 16-44) and `SetMeetingNotes` (page 16-54) to access this slot.

Notes for nonrepeating meetings and events are stored in the `notesData` slot.

`version` Contains the integer 2 if the meeting or event was created by version 2.0 of the Dates application. If this slot is missing or its value is `nil`, the Dates application assumes the meeting or event was created by the 1.x version of the application. When the Dates application converts a 1.x meeting or event to 2.0 format, it sets this slot to the value 2.

`viewBounds` A bounds frame used only for meetings that are not at the left edge of the Day view, such as double-booked meetings.

`exceptions` Used for repeating meetings or events. This is an array of arrays representing meetings and events that are exceptions to the normal repeating time; for example, when a user has erased one of the instances of a meeting or event or has changed the starting time or duration. It would then be listed in this array as an exception.

▲ **WARNING**

The internal format of exception meetings is subject to change, so you should treat this array as read-only, and not attempt to add to it. ▲

Each subarray represents one exception. There are two elements in each array. The first is an integer specifying the normal time of the meeting or event, in the number of minutes since midnight, January 1, 1904. The second element is either `nil`, meaning the meeting or event has been erased, or is an exception meeting frame that contains the changed information, such as a different `mtgStartDate` or `mtgDuration`.

Note that the `viewStationery` slot of an exception meeting frame contains the symbol `'exceptionMeeting`. The `viewStationery` slot of an exception event frame contains the symbol `'CribNote`.

Notes Frames

Notes frames occur in the Repeat Meetings and Repeat Notes soups. Notes frames contain the notes for a specific instance of a repeating meeting or event.

Each notes frame contains the following required slots:

Slot descriptions

`notes` — An array of note objects. Notes can consist of text objects, polygons, or ink objects. These objects have the same format as data objects in the Notes soup. For information on the format of these objects, see the description of the `data` slot in the section "Notes Soup Format" (page 16-82). Text objects have, in addition, a `viewFont` slot specifying the font of the text.

`repeatingMeetingAlias` — An alias to the repeating meeting or event soup entry with which these notes are associated. For information on entry aliases see Chapter 11, "Data Storage and Retrieval," in *Newton Programmer's Guide*.

Dates Error Codes

A list of the error codes in the Dates API follows, shown alphabetically by the function or set of functions that throws the exception. More than one method are grouped together if they all throw the same exceptions for the same reasons. The method name, the name of the error, the error code number, the value, and a text string are given. The text string is a detailed explanation of the error, but is not displayed.

Note

A few of the functions listed are not described elsewhere. These functions are not intended for your use. However, since public functions call these, the exceptions they throw are listed below. ◆

`AddAppointment`

Error: `kFramesErrUnexpectedImmediate`
Error Code: –48418
Value: `'mtgDuration` argument
Message: `"Expected positive integer for 'mtgDuration argument."`

Plus all those thrown by `ValidatePeriod` and `ValidateTitleAndDatetimeArgs`.

`AddEvent`

All those thrown by `ValidatePeriod` and `ValidateTitleAndDatetimeArgs`.

`DeleteAppointment`, **`DeleteEvent`**, and **`DeleteRepeatingEntry`**

Error: `kFramesErrNotTrueOrNil`
Error Code: –48424
Value: `'deleteOneOnly` argument
Message: "`Expected true or nil for 'deleteOneOnly argument.`"

Plus all those thrown by `IdentifyAppointments`.

`FindAppointment`

Error: `kFramesErrNotAStringOrNil`
Error Code: –48414
Value: `'mtgText` argument
Message: "`Expected string or nil for 'mtgText argument.`"

Error: `kFramesErrNotAnArrayOrNil`
Error Code: –48413
Value: `'findWords` argument
Message: "`Expected array of strings or nil for 'findWords argument.`"

Error: `kFramesErrNotAnArrayOrNil`
Error Code: –48413
Value: `'dateRange` argument
Message: "`Expected nil or integer or array of two integers for 'dateRange argument.`"

Error: `kFramesErrNotAnArrayOrNil`
Error Code: –48413
Value: `'type` argument
Message: "`Expected nil, a meeting type symbol, or an array of meeting type symbol(s) for 'type argument. The`

possible meeting type symbols are Meeting, Event, RepeatingMeeting, and RepeatingEvent."

Error: kFramesErrUnexpectedImmediate
Error Code: –48418
Value: maxNumberTofind argument
Message: "Expected nil or number for maxNumberTofind argument."

FindExactlyOneAppointment

Error: kFramesErrUnexpectedImmediate
Error Code: –48418
Value: array of appointments found
Message: "More than one matching meeting/event found."

Error: kFramesErrUnexpectedImmediate
Error Code: –48418
Value: nil
Message: "No matching meeting/event found."

Plus all those thrown by FindAppointment.

GetMeetingIconType

All those thrown by IdentifyAppointments.

GetMeetingInvitees, **GetMeetingLocation**, and **GetMeetingNotes**

All those thrown by ValidateTitleAndDatetimeArgs and FindExactlyOneAppointment.

IdentifyAppointments

Error: kFramesErrUnexpectedFrame
Error Code: –48416
Value: 'mtgTextOrFrame argument
Message: "The frame argument is not a meeting frame."

Error: kFramesErrUnexpectedImmediate
Error Code: –48418
Value: nil
Message: "No meeting/event found with that title on that date"

Error: `kFramesErrUnexpectedImmediate`
Error Code: –48418
Value: array of appointments found
Message: "`More than one meeting/event found with that title on that date.`"

Plus all those thrown by `ValidateTitleAndDatetimeArgs` and `FindAppointment`.

`MoveAppointment`, and **`MoveOnlyOneAppointment`**

Error: `kFramesErrUnexpectedImmediate`
Error Code: –48418
Value: `'newDatetime` argument
Message: `"Expected positive integer or nil for 'newDatetime argument."`

Error: `kFramesErrUnexpectedImmediate`
Error Code: –48418
Value: `'newDuration` argument
Message: `"Expected positive integer or nil for 'newDuration argument."`

`RegMeetingType`

Error: `kFramesErrUnexpectedFrame`
Error Code: –48416
Value: `'frame` argument
Message: `"The 'frame argument to RegMeetingType is missing the item, icon or newMeeting slots."`

`SetEntryAlarm`

Error: `kFramesErrUnexpectedImmediate`
Error Code: –48418
Value: `'minutesBefore` argument
Message: "`Expected nil or non-negative integer for 'minutesBefore argument.`"

Plus all those thrown by `ValidateTitleAndDatetimeArgs` and `FindExactlyOneAppointment`.

SetMeetingIconType

Error: kFramesErrNotASymbol
Error Code: –48410
Value: 'newIconType argument
Message: "Expected 'Meeting, 'WeeklyMeeting, 'Event, 'MultiDayEvent, or 'AnnualEvent for 'newIconType argument."

Error: kFramesErrUnexpectedImmediate
Error Code: –48418
Value: nil
Message: "New icon type is incompatible with old type."

Plus all those thrown by ValidateTitleAndDatetimeArgs and FindExactlyOneAppointment.

SetMeetingInvitees, **SetMeetingLocation**, and **SetMeetingNotes**

All those thrown by ValidateTitleAndDatetimeArgs and FindExactlyOneAppointment.

SetRepeatingEntryStopDate

Error: kFramesErrNotAnInteger
Error Code: –48406
Value: 'mtgStopDate argument
Message: "Expected non-negative integer for 'mtgStopDate argument."

Plus all those thrown by ValidateTitleAndDatetimeArgs and FindExactlyOneAppointment.

ValidatePeriod

Error: kFramesErrNotNil
Error Code: –48422
Value: 'repeatInfo argument
Message: "Expected nil for 'repeatInfo argument."

Error: kFramesErrUnexpectedImmediate
Error Code: –48418
Value: 'repeatPeriod argument

Message: "Expected nil, 'daily, 'weekly, 'biweekly, 'monthlyByWeek, 'monthly, 'yearly, or 'yearlyByWeek for 'repeatPeriod argument."

Error: kFramesErrNotAnArrayOrNil
Error Code: –48413
Value: 'repeatInfo argument
Message: "Expected nil or array of integers for 'repeatInfo argument."

Error: kFramesErrNotAnArrayOrNil
Error Code: –48413
Value: 'repeatInfo argument
Message: "Since 'period is 'yearlyByWeek, expected nil or array of one integer for 'repeatInfo argument."

Error: kFramesErrNotAnArray
Error Code: –48401
Value: 'repeatInfo argument
Message: "Expected array of integers for 'repeatInfo argument."

Error: kFramesErrUnexpectedImmediate
Error Code: –48418
Value: 'repeatInfo argument
Message: "Since 'repeatPeriod is 'weekly, expected integers between 0 and 6 for 'repeatInfo argument."

Error: kFramesErrUnexpectedImmediate
Error Code: –48418
Value: 'repeatInfo argument
Message: "Since 'repeatPeriod is 'monthlyByWeek or 'yearlyByWeek, expected integers between 1 and 5 for 'repeatInfo argument."

Error: kFramesErrUnexpectedImmediate
Error Code: –48418
Value: 'repeatInfo argument

Message: "`Meeting 'mtgStartDate does not fit in any value in 'repeatInfo argument.`"

`ValidateTitleAndDatetimeArgs`

Error: `kFramesErrNotAString`
Error Code: –48402
Value: `'mtgText` or `'mtgTextOrFrame` argument
Message: `"Expected string for meeting 'mtgText or 'mtgTextOrFrame argument."`

Error: `kFramesErrUnexpectedImmediate`
Error Code: –48418
Value: `'mtgStartDate` argument
Message: `"Expected positive integer for 'mtgStartDate argument."`

To Do List Reference

This sections describes the To Do List methods and soup format.

To Do List Methods

This section describes the methods defined by the To Do List that are available to you. To obtain a reference to the To Do List to send these messages, use the following code:

`GetRoot().calendar:GetToDo()`.

Note that future Newton devices may not include the To Do List application. You should therefore check for the existence of the To Do List application before trying to access it. Use the following code for this test:

```
if GetRoot().calendar then
   if GetRoot().calendar:?GetToDo() then ...
```

CreateToDoItem

toDoFrame:`CreateToDoItem(`*date*, *richString*, *reminder*, *frequency*`)`

Adds a task with the specified text for the specified date. It returns an integer, which is the index of the item made (that is, if the new task is the second one on the specified date, `CreateToDoItem` returns 1).

date — The date of the task in the number of minutes passed since midnight, January 1, 1904.

richString — A string or rich string for the text associated with the task.

reminder — An integer. The number of days notice before the task, or `nil`.

frequency — A frequency frame, or `nil` if the task does not repeat. A frequency frame should have the following slots:

Slot descriptions

`mtgStartDate`
: An integer; the start date of the task in the number of minutes passed since midnight, January 1, 1904.

`mtgStopDate`
: An integer; the date that the task should stop repeating, in the number of minutes passed since midnight, January 1, 1904.

`repeatType`
: One of the following constants that describes how often the meeting repeats: `kDayofWeek` (0), `kWeekInMonth` (1), `kDateInMonth` (2), `kDateInYear`(3), `kPeriod`(4), `kNever`(5), `kWeekInYear`(7).

`mtgInfo`
: An immediate value containing packed repeating task information. This slot is interpreted differently, depending on the value of the `repeatType` slot; see the description of the `mtgInfo` slot in “Meeting Frames” (page 16-57).

CreateToDoItemAll

toDoFrame:`CreateToDoItemAll(`*date*, *richString*, *reminder*, *frequency*, *priority*, *completed*`)`

Adds a task with the specified text for the specified date and sets the priority and completion status. It returns an integer, which is the index of the item made (that is, if the new task is the second one on the specified date, `CreateToDoItem` returns 1).

date — The date of the task in the number of minutes passed since midnight, January 1, 1904.

richString — A string or rich string for the text associated with the task.

reminder — An integer. The number of days notice before the task, or `nil`.

frequency — A frequency frame or `nil` if the task does not repeat. A frequency frame should have the following slots:

Slot descriptions

`mtgStartDate`
: An integer; the start date of the task in the number of minutes passed since midnight, January 1, 1904.

`mtgStopDate`
: An integer; the date that the task should stop repeating, in the number of minutes passed since midnight, January 1, 1904.

`repeatType`
: One of the following constants that describes how often the meeting repeats: `kDayofWeek` (0), `kWeekInMonth` (1), `kDateInMonth` (2), `kDateInYear`(3), `kPeriod`(4), `kNever`(5), `kWeekInYear`(7).

`mtgInfo`
: An immediate value containing packed repeating task information. This slot is

interpreted differently, depending on the value of the `repeatType` slot; see the description of the `mtgInfo` slot in "Meeting Frames" (page 16-57).

priority An integer for the priority level of the task: 0 = high, 1 = medium, 2 = low, 3 = none.

completed A Boolean indicating whether the task is completed or not.

EnsureVisibleTopic

toDoFrame`:EnsureVisibleTopic(`*index*`)`

Scrolls the To Do List to make the task specified in the parameter *index* visible. This method requires the To Do List to be open.

index An integer index referring to the position of a topic in the To Do List. It is the index into the `topics` array of tasks for the day, see "To Do List Soup Format."

This method's return value is unspecified.

GetToDoEntry

toDoFrame`:GetToDoEntry(`*date*`, `*makeNewEntry*`)`

Returns an array of soup entries containing tasks for that date. Note that even though a particular days tasks are stored under the day of the task (or under day 0 for repeating tasks), this array may contain more than one element. This is because these soup entries are on different stores. This method requires the To Do List to be open.

date The date to get soup entries from, in the number of minutes passed since midnight, January 1, 1904.

makeNewEntry A Boolean, if `true` and no soup entry exist for *date*, one is created.

If there are no entries on the specified date, and *makeNewEntry* is `nil`, this method returns `nil`.

GetToDoItemsForRange

toDoFrame:`GetToDoItemsForRange(`*beginDate*`,` *endDate*`)`

Returns an array of frames for each day after *beginDate* and before *endDate*, including the two boundary dates. These frames have the following format:

`{date:` *aDateInTheRange*`,topics:` *anArrayOfTopicsForThatDate*`}`

For information on the `date` and `topics` slots returned in this frame see "To Do List Soup Format" (page 16-77).

beginDate	The date that forms the beginning boundary for the range of dates, in the number of minutes passed since midnight, January 1, 1904.
endDate	The date that forms the ending boundary for the range of dates, in the number of minutes passed since midnight, January 1, 1904.

If *beginDate* is larger than *endDate*, this method returns `nil`.

GetToDoItemsForThisDate

toDoFrame:`GetToDoItemsForThisDate` (*date*)

Returns an array of tasks for the date specified. This array contains frames such as in the `topics` array in To Do List soup entries, see "To Do List Soup Format" (page 16-77). It merges tasks from multiple stores, such as those resulting from duplication on storage cards, repeating tasks, and tasks imported from earlier versions of the software. This method has the side effect of sorting the soup entry's topics; it sorts them as the user sees them, by undone and done first and then by priority.

date	The date to get the tasks array from, in the number of minutes passed since midnight, January 1, 1904.

GetTaskShapes

toDoFrame:`GetTaskShapes(`*originalShapes*`,` *task*`,` *yOffset*`,` *width*`,` *font*`)`

Returns the shapes needed to draw the task. This is used in printing, and is called by `GetToDoShapes`.

originalShapes — An array to which the shapes are added and returned, or `nil` if you do not have shapes to add to. If `nil`, a new array is created with the following style element:

`{font:` *font*`, justification: 'left}`

task — The task specified.

yOffset — All shapes receive this vertical offset. The units are pixels.

width — The width to wrap text to, in pixels.

font — The font you want to draw the text shapes in. If *originalShapes* is `nil`, the array returned has this value in its style element's `font` slot.

This method returns a frame with the following slots:

Slot descriptions

`shapes` — A nested array of shapes as described in "Using Nested Arrays of Shapes" beginning on page 13-10 in *Newton Programmer's Guide*.

`height` — An integer, the height of the shapes in the `shapes` slot.

GetToDoShapes

toDoFrame:`GetToDoShapes(`*date*`,` *yOffset*`,` *width*`,` *font*`)`

Returns the shapes needed to draw the tasks on the specified date. This method is used in printing. It calls `GetTaskShapes`.

date — The date to get shapes for in the number of minutes passed since midnight, January 1, 1904.

yOffset — All shapes receive this vertical offset. The units are pixels.

width — The width to wrap text to, in pixels.

font — The font you want to draw the text shapes in. The array returned has this value in its style element's `font` slot.

This method returns a nested array of shapes as described in "Using Nested Arrays of Shapes" beginning on page 13-10 in *Newton Programmer's Guide*.

LastVisibleTopic

toDoFrame`:LastVisibleTopic()`

Returns the index to the `topics` array of the last task drawn in the view. Note that this is not necessarily equal to the length of the `topics` array, since there may be tasks after the last one drawn that are simply not shown at the current scroll position. For information about the `topics` array, see "To Do List Soup Format" (page 16-77).

NextToDoDate

toDoFrame`:NextToDoDate(`*date*`)`

Returns the date of the next task on or after the specified date, or `nil` if there is none.

date — The date of the task in the number of minutes passed since midnight, January 1, 1904.

RemoveOldToDoItems

toDoFrame`:RemoveOldToDoItems(`*beforeDate*`,` *removeWhich*`, nil)`

Removes any To Do task dated prior to *beforeDate*.

beforeDate — The date of the oldest allowable To Do item, which you can specify in the number of minutes passed since midnight, January 1, 1904.

removeWhich — Set to `'done` or `nil`. If *removeWhich* is `'done`, only completed tasks are removed. If *removeWhich* is `nil`, every task before *beforeDate* is removed.

SetDone

toDoFrame:`SetDone(`*index, topic, done,* `nil, nil)`

Marks the done state of a task. This method requires the To Do List to be open.

index	An integer index referring to the position of a topic in the To Do List. It is the index into the `topics` array of tasks for the day; see "To Do List Soup Format" (page 16-77).
topic	A topic frame, as returned by the `GetToDoItemsForThisDate` and `GetToDoItemsForRange` functions.
done	A Boolean, what to set the `mtgDone` slot of the topic to.
`nil`	Always pass `nil` for the fourth parameter.
`nil`	Always pass `nil` for the fifth parameter.

SetPriority

toDoFrame:`SetPriority(`*index, priority, undo*`)`

Sets or undoes the setting of the priority of a task. Depending on what that priority number is, it also re-orders the topics in the list. This method requires the To Do List to be open.

index	An integer index referring to the position of a topic in the To Do List. It is the index into the `topics` array of tasks for the day; see "To Do List Soup Format" (page 16-77).
priority	An integer to set the priority level of the task: 0 = high, 1 = medium, 2 = low, 3 = none.
undo	This value should be `nil` to set a topic's priority, or `true` if this is part of an Undo operation, as shown in the following line of code: `AddUndoAction('SetPriority, [theIndex, oldPriority, true]);`

To Do List Soup Format

This section describes the format of entries in the To Do List Soup. The To Do List soup is called "To Do List." In this soup each day has a single entry, and all repeating tasks are stored under the 0 date. Each day's frame has the following slots:

Slot descriptions

`class` Always the symbol `'todo`.

`needsSort` A Boolean; whether this day needs to be sorted because of a change in a task's priority, done status, or any other reason.

`date` The date of the task. All repeating tasks are stored under the 0 date.

`topics` An array of tasks for this date. Each task is a frame with the following slots:

Slot descriptions

`text` The text of the task.

`styles` A styles frame for the text of the task.

`mtgDone` A Boolean, whether the task is checked off.

`mtgPriority` An integer for the priority level of the task: 0 = high, 1 = medium, 2 = low, 3 = none.

`repeatDone` An array of dates of the completed repeating tasks. For example if you had a four day repeat task and day 1 and 2 were done, it would contain those two dates.

`viewBounds` A bounds frame for the task from the last time it was displayed. The To Do List recalculates this when necessary.

`source` An integer used internally to specify what store a task is stored in.The To Do List sets this slot when necessary.

`unique` An integer which pseudo-uniquely identifies the task. The To Do List sets this slot when necessary.

`repeatInfo` A frequency frame with the following slots:

Slot descriptions

`mtgStartDate`
An integer, the start date of the task in the number of minutes passed since midnight, January 1, 1904.

`mtgStopDate`
An integer, the date that the task should stop repeating, in the number of minutes passed since midnight, January 1, 1904.

`repeatType`
One of the following constants that describe how often the task repeats: `kDayofWeek` (0), `kWeekInMonth` (1), `kDateInMonth` (2), `kDateInYear`(3), `kPeriod`(4), `kNever`(5), `kWeekInYear`(7).

`mtgInfo`
An immediate value containing packed repeating meeting information. This slot is interpreted differently, depending on the value of the `repeatType` slot; see the description of the `mtgInfo` slot in "Meeting Frames" (page 16-57).

Time Zones Reference

The developer's interface to Time Zone consist of functions that retrieve information about a city or country, and methods that add a city and set the current home city.

Time Zones Functions and Methods

This section describes the Time Zones methods.

To obtain a reference to the Time Zones application to send these messages use the following code:

```
GetRoot().worldClock
```

Note that future Newton devices may not include the Time Zones application. You should therefore check for the existence of the Time Zones application before trying to access it. Use the following code to test for this:

```
if GetRoot().worldClock then ...
```

GetCityEntry

`GetCityEntry(`*cityName*`)`

Returns an array of frames for cities whose name slot matches *cityName*, i.e. if *cityName* is "Portland" the routine returns an array containing both Portland, OR and Portland, ME. If no entries match an empty array is returned. This method may return `nil` if a problem has occurred.

cityName	A string for the city name to search for.

The array returned has frames that have the following slots:

Slot descriptions

`name`	A string for the name of the city.
`longitude`	An integer for the longitude of the city; see "Using Longitude and Latitude Values" beginning on page 19-30 in the *Newton Programmer's Guide*.
`latitude`	An integer for the latitude of the city; see "Using Longitude and Latitude Values" beginning on page 19-30 in the *Newton Programmer's Guide*.
`gmt`	An integer for the offset in minutes from Greenwich Mean Time.
`country`	A symbol representing the country this city is in.

`areaCode` A string for the city's area code.

`region` A string for the region of the city (a state in the U.S., a province in Canada).

`airport` A string for the airport designation of the city, or an array of strings if more than one airport serves this city.

GetCountryEntry

`GetCountryEntry(`*countryName*`)`

Returns an array of frames for countries whose `name` or `symbol` slots matches *countryName*. The *countryName* parameter is compared to the frames for countries in ROM in two ways: using standard string comparison on the `name` slot, and the class of the string is compared to the `symbol` slot. To set the class of a string for this second test call `SetCountryClass`.

This method may return `nil` if a problem has occurred.

countryName A string for the country name to search for.

The frames returned have the following slots:

Slot descriptions

`name` A string for the name of the country.

`symbol` A symbol for the name of the country.

`longitude` An integer for the longitude of the country; see "Using Longitude and Latitude Values" beginning on page 19-30 in the *Newton Programmer's Guide*.

`latitude` An integer for the latitude of the country; see "Using Longitude and Latitude Values" beginning on page 19-30 in the *Newton Programmer's Guide*.

`currency` A string for the name of the national currency.

`continent` A symbol for the continent of the country.

`countryCode` A string for the country's international telephone code.

`outgoing` A string that should be dialed before making an international call from this country.

SetLocation

worldClock`:SetLocation(`*whichCity*`)`

Sets the home city in the Time Zones application.

whichCity	A frame with the same slots as the frames returned by `GetCityEntry`. The following slots are required in this frame: `name`, `country`, `longitude`, `latitude`, and `gmt`.

NewCity

worldClock`:NewCity(`*newCityFrame*`, nil, `*makeHome*`)`

Adds the city specified by *newCityFrame*.

newCityFrame	A frame with the same slots as the frames returned by `GetCityEntry`. The following slots are required in this frame: `name`, `country`, `longitude`, `latitude`, and `gmt`.
`nil`	The second parameter should always be `nil`.
makeHome	A Boolean, whether to make this the home city.

Notes Reference

This section describes the Notes methods and soup format.

Notes Methods

The following methods let you the user about the size of a note, and create notes. To obtain a reference to the Notes application in order to send it messages, use the following code:

```
GetRoot().paperroll
```

Note that future Newton devices may not include the Notes application. You should therefore check for the existence of the Notes application before trying to access it. Use the following code to test for this:

```
if GetRoot().paperroll then ...
```

MakeTextNote

paperroll:`MakeTextNote(`*string*`, `*addIt*`)`

Adds a simple text note to the Notes soup. The height is calculated automatically.

string The string is the text you want appear as a note.

addIt A Boolean. If `true` the note is added to the Notes soup; if `nil` the note frame is returned.

NewNote

paperroll:`NewNote(`*note*`, `*goto*`, `*store*`)`

Adds a note to the Notes soup. Use this method in combination with `MakeTextNote` to add an entry to the Notes soup containing more than a simple string, as described in “Creating New Notes” beginning on page 19-32 in *Newton Programmer’s Guide*.

note A note, as returned by `MakeTextNote`.

goto A Boolean. If `true`, the new note is displayed. If the Notes application is not open, this parameter is ignored.

store The store to add this note to. A value of `nil` specifies the default store.

Notes Soup Format

This section describes the format of entries in the Notes soup. Each entry consists of a frame with the following slots:

Slot descriptions

`viewStationery` This slot always contains the symbol `'paperroll`.

`class` The class symbol varies according to the type of stationery the user creates on the New picker, which may be a note (class `'paperroll`), an outline (class `'list`), or a checklist (class `'checkList`).

`height` This slot contains an immediate value that is the height, in pixels, of the note.

`timeStamp` Contains an immediate value: the date and time that this note was created, in the number of minutes passed since midnight, January 1, 1904. This slot must never contain the value `nil`.

`labels` Optional. A symbol specified by the user as a label (file folder) for the note.

`title` Optional. A string or rich string displayed in the status bar of the note. The user can change this by tapping the notes's icon.

`data` For notes (class `'paperroll`), this slot holds an array of frames, which contains either text, polygon, ink, or image objects. For outlines and checklists (classes `'list` and `'checkList`), this slot is set to `nil`. There is one frame for each text, polygon, ink, or image object in the note.

The text object frames have these slots:

`viewStationery` Required. Always contains the symbol `'para`.

`viewBounds` Required. The bounds of the text object.

`text` Required. A string that is the text contained in the paragraph.

`tabs` Optional. An array of tab stops.

`styles` Optional. An array holding font style information for the text.

For more information on the slots particular to paragraph views, see “Paragraph Views” beginning on page 8-10 in *Newton Programmer’s Guide*.

The polygon object frames have these slots:

`viewStationery`
: This slot always contains the symbol `'poly`.

`viewBounds`
: The bounds of the polygon.

`points`
: Contains a binary data structure, which holds polygon data.

The ink object frames have these slots:

`ink`
: This slot contains a binary data structure of the class `'ink` that holds the ink data.

`viewBounds`
: The bounds of the ink object.

`timeStamp`
: Contains an immediate value: the date and time that this note was created, in the number of minutes passed since midnight, January 1, 1904.

The image object frames have these slots:

`viewStationery`
: This slot always contains the symbol `'pict.`

`viewBounds`
: A bounds frame.

`icon`
: A bitmap frame.

`topics`
: This slot is present only for outline and checklist entries (classes `'list` and `'checkList`). This slot contains an array of frames with the following slots:

`text`
: The text for this item.

`styles`
: A styles frame for the text.

`viewBounds`
: The bounds frame of the text object.

`level`
: The indentation level. The default value, 1, specifies the left margin. This slot will always be set to 1 for checklist entries.

`hideCount`	Specifies how many items are hidden at `level`.
`mtgDone`	This slot appears only in checklist entries. A value of `true` indicates that the topic has a check; `nil` indicates that it does not.

Icons and the Extras Drawer Reference

This section lists the Extras Drawer constants, data structures, functions, and methods.

Extras Drawer Data Constants

The following constants are used by the Extras Drawer.

Table 16-8 Folder symbols

Folder	Symbol
Unfiled	`nil`
Extensions	`'_extensions`
Help	`'_help`
Setup	`'_setup`
Storage	`'_soups`

Extras Drawer Data Structure

This section describes the `soupervisor` frame.

The Soupervisor Frame

This frame must be in the base view of the application in order for the soupervisor mechanism to be able to file or move its soups. It must be in a slot named `soupervisor`, and must contain the following slots:

Slot descriptions

`type`	Required. A symbol, either `'moveOnly`, `'fileOnly`, or `'all`.
`FileSoup`	Optional method. Called to file an entire soup; see `FileSoup` (page 16-86).
`FileEntry`	Optional method. Called to file a soup entry; see `FileEntry` (page 16-87).
`MoveEntry`	Optional method. Called to move a soup entry to a different store; see `MoveEntry` (page 16-87).

FileSoup

`FileSoup(`*newLabels*`, `*newStore*`)`

If you define this method you must do all the work of either moving the soup to a different store or changing the `labels` slot of the entry—which is all filing really is. Note that if you define this method, your `FileEntry` and `MoveEntry` methods (if you defined either) will not be called by the system.

newLabels One of the following objects:

- A symbol. This is a valid folder symbol.
- `Nil`. The entry should be unfiled.
- Anything else. Ignore this entry.

newStore The store to move this soup to, or `nil`. A value of `nil` signifies that the soup should stay in its present store.

It is possible for both of these values to be valid, if the user has opted both to file this soup and change its store. Also note that in some cases the value of the *newLabels* argument can be something other than a symbol or `nil`. You must check for this case or the `labels` slot to the entries could be set to an invalid folder, as in this code fragment:

```
if (newLabels = NIL) or IsSymbol (newLabels) then
begin
   // file this soup
end
else // don't file it
```

To improve performance, you should not use broadcast calls until the last entry has been changed.

FileEntry

`FileEntry(`*entry*`, `*newLabels*`)`

This method is called once for each soup entry if your `soupervisor` frame does not have a `FileSoup` method defined. It must file the entry in *newLabels*.

entry	The entry to file.
newLabels	The value to set the `labels` slot of *entry*.

In some cases the value of the *newLabels* argument can be something other than a symbol or `nil`. You must check for this case or the `labels` slot to the entries could be set to an invalid folder, as in this code fragment:

```
if (newLabels = NIL) or IsSymbol (newLabels) then
begin
   // file this soup
end
else // don't file it
```

MoveEntry

`MoveEntry(`*entry*`, `*newStore*`)`

This method is called once for each soup entry if your `soupervisor` frame does not have a `FileSoup` method defined. It must move the entry to *newStore*. If the entry is currently on a write-protected store, this method must copy the entry to *newStore*.

entry The entry to move to a different store.

newStore The store to move *entry* to.

Extras Drawer Methods

This section describes the Extras Drawer methods. Use the following code in order to get a reference to the Extras Drawer to send it these messages:

```
GetRoot().extrasDrawer
```

AddExtraIcon

extrasDrawer:`AddExtraIcon(`*iconType*`,` *paramFrame*`,` *pkgName*`,` *store*`) // platform file function`

Adds either a script or a soup icon to the specified store. See also “Adding a Soup Icon” (page 19-40), and “Creating a Script Icon” beginning on page 19-42 in *Newton Programmer’s Guide*.

IMPORTANT
This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

`call kAddExtraIconFunc with (`*iconType*`,`*paramFrame*`,`*pkgName*`,` *store*`);`

▲

iconType A symbol specifying the type of icon. This can be either `'soupEntry` or `'scriptEntry`.

paramFrame A frame containing information about the new icon. The slots in this frame vary depending on the value of *iconType*. The *paramFrame* frames for both types of icon share these slots:

`text` Required. A string that is shown under the icon.

`app`	Recommended. A unique symbol used by `SetExtrasInfo` (page 16-92) to find the icon.
`labels`	Optional. A symbol designating the Extras Drawer folder to file this icon in. See "Folder symbols" (page 16-85). Do not specify `nil`.

In addition, the *paramFrame* of soup icons should have these slots:

`ownerApp`	Optional. The `appSymbol` of the application that owns these soups. This is needed for the soupervisor mechanism.
`soupNames`	An array of strings that are the names of the soups combined under this icon.

The *paramFrame* of script icons should have these additional slots:

`tapAction`	A function object that is called when the icon is tapped. It is passed no parameters. This object is stored in a soup, so you should keep it as small as possible.
`icon`	A bitmap frame containing the icon to be displayed; it should be 32x32 pixels.

pkgName A string specifying the package this icon should be associated with. For soup icons, this must be different from your application's package name. Script icons may want to use the same package name. Never pass `nil` for this argument.

store The store on which to keep the new icon. A value of `nil` specifies the default store.

GetExtraIcons

extrasDrawer:`GetExtraIcons(`*iconType*`,` *pkgName*`,` *store*`)`

Returns an array of all icons added by `AddExtraIcon` (page 16-88), of type *iconType*, that are owned by the package *pkgName*, and are on the store *store*. Do not rely on the format of the array elements returned; this may change in future versions of the system software.

iconType	A symbol; either `'scriptEntry` or `'soupEntry`.
pkgName	The package name used in the call to `AddExtraIcon`.
store	The store to look on.

GetPartCursor

extrasDrawer:`GetPartCursor(`*packageName*`,` *store*`,` *folderSym*`) // platform file function`

Returns a cursor for entries corresponding to parts (icons) displayed in the Extras Drawer.

IMPORTANT

This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

`call kGetPartCursorFunc with (`*packageName*`,` *store*`,` *folderSym*`);`

▲

packageName	Specify a string naming a package, or `nil`. If you specify a package name, the cursor returns only parts from that package. To return parts from all packages, specify `nil`.
store	Specify a store object or `nil`. If you specify a store object, the cursor returns parts from only that store. To return parts from all stores, specify `nil`.
folderSym	Specify a symbol identifying a folder, or `nil`. If you specify a folder symbol, the cursor returns only parts filed within that Extras Drawer folder. To return parts

from all folders, specify the symbol `'_all`. To return parts from the unfiled folder, specify `nil`.

The structure of the entries returned by the cursor is subject to change. Entries should be accessed only by using the functions `GetPartEntryData` (page 16-91), `LaunchPartEntry` (page 16-92), and `SetExtrasInfo` (page 16-92). Do not directly change the entries returned by `GetPartCursor`.

GetPartEntryData

extrasDrawer:`GetPartEntryData(`*entry*`) //platform file function`

Returns a frame containing information about an Extras Drawer part entry.

IMPORTANT

This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

`call kGetPartEntryDataFunc with (`*entry*`);`

▲

entry	An entry obtained from a part cursor; by using `GetPartCursor` (page 16-90).

The frame returned has the following slots:

Slot descriptions

`icon`	A bitmap frame (of the kind returned by `GetPictAsBits`) containing the bitmap for the part icon displayed in the Extras Drawer.
`text`	A string that is the text shown under the part icon.
`labels`	A symbol identifying the Extras Drawer folder in which the part is filed. For a list of these see "Folder symbols" (page 16-85).
`appSymbol`	A symbol identifying the application, if the part frame has an `app` slot.
`packageName`	A string that is the name of the package that contains the part.

LaunchPartEntry

extrasDrawer:`LaunchPartEntry(`*entry*`) //platform file function`

Launches the specified part. The operation is the equivalent of the user tapping the part icon in the Extras Drawer.

IMPORTANT

This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

`call kLaunchPartEntryFunc with (`*entry*`);`

▲

entry An entry obtained from a part cursor, by using `GetPartCursor` (page 16-90).

This function returns a non-`nil` value if the Extras Drawer would have closed itself after the icon was tapped. It returns `nil` if the Extras Drawer would have stayed open after the icon was tapped.

RemoveExtraIcon

extrasDrawer`:RemoveExtraIcon(`*extraIcon*`)`

Removes an icon added by `AddExtraIcon` (page 16-88).

extraIcon An element of the array returned by `GetExtraIcons` (page 16-90).

To get a reference to the Extras Drawer use this code:
`GetRoot().extrasDrawer:RemoveExtraIcon(`*extraIcon*`);`

SetExtrasInfo

extrasDrawer:`SetExtrasInfo(`*paramFrame*`, `*newInfo*`) //platform file function`

Changes the Extras Drawer information for the specified Extras Drawer icon. The return value of this function is the information frame that was in effect before this call. If the icon isn't found, this function returns `nil`.

IMPORTANT

This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

call kSetExtrasInfoFunc with (*paramFrame*, *newInfo*);

▲

paramFrame One of the following objects:

- A frame identifying the icon whose Extras Drawer information you want to change. This frame can have the following slots:

  `appSymbol`
  Required. A symbol identifying the application that the icon represents.

  `store`
  Optional. A store object identifying the store on which the icon resides.

  `packageName`
  Optional. A string naming the package to which the icon belongs.

- An entry obtained from a part cursor; by using `GetPartCursor` (page 16-90).
- Your `appSymbol`. Note that this allowed for compatability reasons, it may not be supported in future versions of the system software.

newInfo A new information frame for the icon represented by *paramFrame*. The slots in this frame are described below. If you don't specify a particular slot (or specify `nil`), the value of the slot is not changed.

You can read and modify the following slots in the *newInfo* frame:

`icon` A bitmap frame (of the kind returned by `GetPictAsBits`) containing the bitmap and mask for the part icon displayed in the Extras Drawer.

`text` A string that is the text shown under the part icon.

`labels` A symbol identifying the Extras Drawer folder in which to file the icon. See Table 16-8 “Folder symbols” (page 16-85).

`soupNames` An array of strings that are the names of soups to be associated with this icon. This slot applies to soup icons only.

`ownerApp` The `appSymbol` of the application that owns the soups. This slot applies to soup icons only.

Fax Soup Entries Reference

This section describes the `body` slot of an In/Out Box fax soup entry.

Body Slot of Fax Soup Entries

This slot contains a frame with the following slots:

Slot descriptions

`sender` A string for the sender’s phone number.

`pages` An array of page data. Each page is a frame with the following slots:

- `image` Required. This slot contains a NewtonScript shape object (see `protoImageView`).
- `annotations` Required. The default is `nil`. When not `nil`, this slot contains an array of shapes or paragraphs as in the Notes application.

	`thumbnail`	Optional. A shape placed here by the Fax Viewer as a cached object. If you don't put one in, it will be added. (See `protoThumbnail`.)
`info`	A frame with the following slots.	
	`dataRate`	The transmission rate of the fax in bits per second, default 2400.
	`endTime`	The point at which the fax completed in the number of minutes passed since midnight, January 1, 1904, The default is 0.
	`pageCount`	The number of pages in the fax. The default is 0.
	`pixelWidth`	The default width for all pages in pixels, default 0.
	`pixelHeight`	The default height for all pages in pixels, default 0.
	`resolution`	The image in dots per inch (dpi). Like a pensize, the value of the resolution slot may be an array or a single value. If this value is an array, the two elements of the array specify the x and y values in dpi. If this slot holds a single value, the pixels are square and have the same value for x and y.
	`startTime`	The point at which the fax started in the number of minutes passed since midnight, January 1, 1904. The default is 0.

▲ **WARNING**

The `info` slot contains internal information and is subject to change. ▲

Prefs and Formulas Rolls Reference

This section describes the methods used to add (and remove) a panel from the Prefs and Formulas rolls. Also included is the proto on which Prefs panels should be based.

Proto

This section describes the proto used to create Prefs items.

protoPrefsRollItem

This proto is used to add an item to the Prefs roll.

Slot descriptions

`overview`	Required. A string displayed in the Prefs overview.
`icon`	Required. A small icon displayed in the Prefs overview and as the title of the panel when it is picked. You may set this to `nil` to not use an icon.
`viewBounds`	Required. A bounds frame.
`height`	Required. The height of the panel.
`viewFlags`	The default setting is `vVisible`.
`viewJustify`	The default setting is `vjParentFullH`.
`viewFormat`	The default setting is `vfNone`.

Prefs and Formulas Functions

This section describes the registry functions which add (or remove) Prefs and Formulas items.

RegFormulas

`RegFormulas(`*appSymbol*`,` *formulasTemplate*`)`

Registers with the system a template used to add a view to the Formulas roll.

appSymbol A unique symbol identifying the application adding this item to the Preferences roll; normally, the value of this parameter is the application symbol, which includes your registered signature.

formulasTemplate A template for the view to be added to the Formulas roll. There is no particular proto on which this template should be based. Instead, this template should

- use a `protoFloatNGo` as the a base, with formula elements added to it
- include a slot named `overview` which contains a string displayed in the Formulas overview
- `viewBounds.bottom` must be equal to the height of the panel
- include a `protoTitle` whose `title` slot is the name of the Formulas panel

This function's return value is unspecified.

UnRegFormulas

`UnRegFormulas(`*appSymbol*`)`

Unregisters the specified Formulas application item.

appSymbol A unique symbol identifying the application adding this item to the Preferences roll. Normally, the value of this parameter is your application symbol, which includes your registered signature.

This function's return value is unspecified.

RegPrefs

`RegPrefs(`*appSymbol*`,` *prefsTemplate*`)`

Registers with the system a template used to add an item to the Preferences roll in the Extras Drawer. The template must be based on the `protoPrefsRollItem` system prototype. Note that items added to the Preferences roll must specify system-wide preferences rather than application-specific ones.

appSymbol A unique symbol identifying the application adding this item to the Preferences roll. Normally, the value of this parameter is your application symbol, which includes your registered signature.

prefsTemplate A view template based on the `protoPrefsRollItem` system prototype; it describes the view to add to the Preferences roll. Items in the Preferences roll must be used for settings that are global in nature, not for application-specific settings.

This function's return value is unspecified.

UnRegPrefs

`UnRegPrefs(`*appSymbol*`)`

Unregisters the specified application's Preference roll items.

appSymbol A unique symbol identifying the application adding this item to the Preferences roll. Normally, the value of this parameter is your application symbol, which includes your registered signature, or some variation on it.

This function's return value is unspecified.

Auxiliary Button Reference

These methods let you to add buttons to the status bars of the Notes, Names, and the background application, and let your applications allow themselves to be extended.

Auxiliary Buttons Functions and Methods

This section describes the functions used to install an auxiliary button, and the application-defined methods needed to support this mechanism.

AddAuxButton

app`:AddAuxButton(`*buttonFrame*`)`

The `AddAuxButton` message is sent to your application when someone calls `RegAuxButton`, specifying your application in the `destApp` slot. It is also sent when your application is the backdrop application and `RegAuxButton` is called with the `destApp` slot set to `nil`.

buttonFrame A frame that contains the `butt` slot. This slot holds the template for the button that was just added.

This method is optional; you are not required to implement it.

GetAuxButtons

`GetAuxButtons(`*appSymbol*`)`

Returns an array that contains the buttons specific to your application and, if yours is the backdrop application, any other buttons designated for the backdrop application. Each array element is a frame with a `butt` slot, which holds the template for the button.

Other slots may exist in this frame, but are undefined and are subject to change.

appSymbol Your unique application symbol with your signature.

RegAuxButton

`RegAuxButton(`*buttonSymbol*`,` *template*`)`

Adds a button to the auxiliary button registry. The return value of `RegAuxButton` is currently undefined.

buttonSymbol The unique symbol for the button. This symbol should include your developer signature.

template A view template for the button to be added, with one extra slot, `destApp`, which you should set to the symbol of the application that you want to add the button to. (For example, use the symbol `'paperroll` or `'cardfile` to add the button to the Notes or the Names application, respectively.) If `destApp` is `nil`, this button is added to the background application, if it can support it.

RemoveAuxButton

app`:RemoveAuxButton(`*buttonSymbol*`)`

The `RemoveAuxButton` message is sent to your application when someone calls `UnRegAuxButton` for a button that is specific to your application. It is also sent when your application is the backdrop application and `UnRegAuxButton` is called for a button whose `destApp` slot is set to `nil`.

buttonSymbol The symbol passed in to `RegAuxButton`, when the button was registered.

This method is optional; you are not required to implement it.

UnRegAuxButton

`UnRegAuxButton(`*buttonSymbol*`)`

Removes the button with the given symbol from the auxiliary button registry. The return value of `UnRegAuxButton` is currently undefined.

buttonSymbol	The symbol passed in to `RegAuxButton`.

System Data Reference

This section describes system stored user configuration data, and the functions used to interact with it. Also included are the functions used to read or write slots in the built-in application's soups.

User Configuration Variables

This section describes those user configuration variables in the System soup that are available to your applications. Certain of these variables are closely associated with text and shape recognition; these are described in the section "Using recConfig Frames" beginning on page 10-8.

Note that you should always use the functions `GetUserConfig` (page 16-107) and `SetUserConfig` (page 16-108) to access and change any of these variables.

`address`	A string or rich string for the first line of the address of the current persona.
`cityZip`	A string or rich string for the second line of the address of the current persona.
`company`	A string or rich string for the company name of the current persona.
`country`	A string or rich string for the name of the country of the current persona.

`countrySlot` A symbol representing the country specified by the user as their Country in the Locale panel in Prefs.

`currentAreaCode` A string for the area code of the current emporium.

`currentCountry` A symbol representing the country of the current emporium.

`currentEmporium` An alias to the `'worksite` Names soup entry designated by the user as the current emporium, or `nil` if the user has picked Other City from, for example, a routing slip.

Note that this is an alias, and thus needs to be resolved before use, as in the following code:

```
ResolveEntryAlias(GetUserConfig('currentEmporium))
```

For more information on entry aliases and the `ResolveEntryAlias` function, see Chapter 11, “Data Storage and Retrieval,” in *Newton Programmer's Guide*.

Call the `UseCurrentEmporium` function (page 16-109) after setting this variable, to force the system to update other user configuration variables.

`currentPersona` An alias to the `'owner` Names soup entry that has been designated by the user as the current persona. Note that this is an alias, and thus needs to be resolved before use, as in the following code:

```
ResolveEntryAlias(GetUserConfig('currentPersona))
```

For more information on entry aliases and the `ResolveEntryAlias` function, see Chapter 11, “Data Storage and Retrieval,” in *Newton Programmer's Guide*.

Call the `UseCurrentPersona` function (page 16-110) after setting this variable to force the system to update other user configuration variables.

`currentPrinter` A frame describing the last printer selected for use in a Print slip.

`dialingPrefix` A string for the dialing prefix of the current emporium.

`doAutoAdd` The default value `true` specifies that words are automatically added to the user dictionary and the autoAdd dictionary.

`doInkWordRecognition`
The value `true` causes the recognizer to convert strokes to ink words rather than sketch ink. This may occur when the text recognizer cannot recognize the input successfully or when text and shape recognition is disabled.

`doTextRecognition`
The value `true` enables word recognition. The system sets the value of this variable to `true` when the user turns on text recognition from the `protoRecToggle` view.

`doShapeRecognition`.
The value `true` enables shape recognition. The system sets the value of this variable to `true` when the user turns on shape recognition from the `protoRecToggle` view.

`emailPassword` A string for the current persona's email password.

`faxPhone` A string or rich string for the current persona's fax phone number.

`homePhone` A string or rich string for the current persona's fax phone number.

`leftHanded` This variable provides a single place for developers to look for a user's handedness preference. A non-`nil` value indicates the user is left handed. You may consider placing your views differently for left-handed users; for example, buttons that would appear on the right edge of the screen might instead be placed on the left edge.

`learningEnabledOption`
The value `nil` specifies that correcting misrecognized words in this view does not modify the system-defined handwriting model. Conversely, the default value `true` specifies that the system records learning data as the user corrects misrecognized words. Because the printed

recognizer does not record learning data, it ignores this value. For more information, see the description of this variable in "protoRecConfig" beginning on page 8-36.

`lettersCursiveOption`
: The default value `true` enables letter-by-letter recognition for certain views in the built-in Names and Dates applications. When this variable holds the value `true`, the cursive recognizer uses letter-by-letter recognition for `protoLabelInputLine` and notes views in the built-in Names and Dates applications. (The printed recognizer always provides letter-by-letter recognition.) The user can set this variable to `true` by checking the "Letter-by-letter in notes" box in the Handwriting Settings preferences slip.

`letterInFieldsOption`
: The value `true` specifies that the cursive recognizer uses letter-by-letter recognition in `protoLabelInputLine` views. (The printed recognizer always provides letter-by-letter recognition.) The user can set this variable to `true` by checking the "Letter-by-letter in fields" box in the Handwriting Settings preferences slip.

`letterSetSelection`
: Sets the recognizer currently in use. This value may be either of the constants `kStandardCharSetInfo` (cursive recognizer) or `kUCBlockCharSetInfo` (printed recognizer). Although the recognizers built into Newton platforms through version 2.0 of system software support these values, not all recognizers are guaranteed to support them. This value may be set by the user from the Handwriting Recognition preferences slip or set programmatically from a `recConfig` frame. For more information, see the description of this variable in "protoRecConfig" beginning on page 8-36.

`letterSpaceCursiveOption`
: The value of this variable affects the amount of space required to consider sets of strokes as belonging to separate letters or words. This value may be set by the

user from the Handwriting Recognition preferences slip or set programmatically from a `recConfig` frame. For more information, see the description of this variable in "protoRecConfig" beginning on page 8-36.

`location` — A frame that holds information about the current city, such as its name, the area code, and the airport designation.

`mailAccount` — A string or rich string for the current persona's e-mail account name.

`mailNetwork` — A symbol representing the e-mail network for the current emporium.

`mailPhone` — A string or rich string for the current persona's phone access number for their e-mail account.

`name` — A string or rich string for the current persona's name, first and last.

`paperSize` — The paper size currently selected for all print jobs. It contains a frame that is an element of the array in the user configuration variable `paperSizes`. That is, the `paperSizes` array is an array of `paperSize` frames. This variable contains the currently selected `paperSize` frame.

The only documented slot in this frame is the read-only `title` slot, which is a user-visible string representing the page size.

▲ **WARNING**

The value of the `paperSizes` array must not be modified and the individual `paperSize` frames must not be modified (they are read only). New size or custom size `paperSize` frames may not be created nor added to the `paperSize` or `paperSizes` variables. ▲

You can set the value of this slot (using `SetUserConfig`) to a frame which is already an element of the `paperSizes` array (using `GetUserConfig`).

Do not change this value without letting the user confirm this change, as it applies to all print jobs. Page size is used as a system global and cannot be overridden for individual print jobs; you cannot, for example, print a set of envelopes and a set of letters without changing the paper size globally.

`paperSizes` The paper sizes currently installed in the system. They appear as choices for the Paper Size item in the Locale Preferences form. This array contains frames such as the user configuration variable `paperSize`.

The dimensions here are independent of the current driver and tend to be somewhat less than the full printable area a driver can support. This leeway allows for printing to different drivers with the same word wrap and other format niceties.

`phone` A string or rich string for the current persona's office phone.

`signature` The signature of the current persona. It is an ink frame.

`speedCursiveOption`

The amount of time the cursive recognizer spends recognizing input. For more information see the description of this variable that begins on (page 8-39).

`timeoutCursiveOption`

This value affects the amount of time the recognizer waits from the completion of a stroke for subsequent strokes that might belong to the same character or word. For more information, see the description of this variable in "protoRecConfig" beginning on page 8-36.

`userFont` A font specification; this can be either a frame or a packed integer. The font to be used for drawing text. For more information on font specifications see "Using Fonts for Text and Ink Display" in Chapter 8, "Text and Ink Input and Display," in *Newton Programmer's Guide*. You should not set this variable.

System Data and Utility Functions

This section includes a description of some global functions applicable to the built-in applications and system data.

GetSysEntryData

`GetSysEntryData(`*entry*`,` *path*`)`

Returns the value of a slot from a built-in soup entry.

entry	The soup entry from which you want to read a slot.
path	A path expression specifying the data to read in the entry.

Use this function whenever you want to read the value of a slot in an entry from one of the built-in soups.

GetUserConfig

`GetUserConfig(`*configSym*`)`

Retrieves the value of a user configuration variable. The user configuration variables are listed in "User Configuration Variables" (page 16-101).

configSym	A symbol naming a user configuration variable.

This function returns the value of the requested user configuration variable.

Here is an example of how to use this function:

```
savedPrinter := GetUserConfig('currentPrinter);
```

RegUserConfigChange

`RegUserConfigChange(`*callbackID*`,`*callBackFn*`)`

Registers a function object to be called each time a user configuration variable changes. Note that it is up to the application that changed one of these variables to broadcast the change. This is not something that you need to worry about, since the `SetUserConfig` function will always broadcast the change. Also note that the system may change, and broadcast the change of, certain undocumented user configuration variables; you should ignore these symbols.

callbackID A unique symbol identifying the function object to be registered; normally, the value of this parameter is the application symbol, which includes your registered signature, or some variation on it.

callBackFn A function object called when a user configuration variable changes. It is passed one parameter, which is a symbol for the user configuration variable that changed.

The value returned by the *callBackFn* function is ignored.

IMPORTANT

This callback function must not call the `RegUserConfigChange` or `UnRegUserConfigChange` functions. ▲

This function's return value is unspecified.

SetSysEntryData

`SetSysEntryData(`*entry*`, `*path*`, `*value*`)`

Sets the value of a slot in a built-in soup entry.

entry The soup entry in which you want to set a slot.

path A path expression specifying the location to set in *entry*.

value The value you want to set in *path*.

Use this function whenever you want to set the value of a slot in an entry from one of the built-in soups, unless there is a specific API function for this.

SetUserConfig

`SetUserConfig(`*configSym*`, `*theValue*`)`

Sets the value of a user configuration variable and writes it to the system soup.

configSym A symbol naming a user configuration variable.

theValue The new value of the user configuration variable identified by *configSym*.

This function returns *theValue*.

Here is an example of how to use this function:

```
SetUserConfig('currentPrinter, savedPrinter);
```

UnRegUserConfigChange

`UnRegUserConfigChange(`*appSymbol*`)`

Unregisters a function object registered by the `RegUserConfigChange` function.

appSymbol A unique symbol identifying the function object to be unregistered. Normally, the value of this parameter is the application symbol, which includes your registered signature, or some variation on it.

▲ **WARNING**
This function's return value is unspecified and may change in the future; do not rely on values returned by this function. ▲

UseCurrentEmporium

`UseCurrentEmporium()`

Forces the system to adjust the values of other user configuration variables to reflect the value of the `currentEmporium` variable. You should call this function after changing the value of `currentEmporium`. Remember to use `SetUserConfig` to change any user configuration variables. Note that this function does not broadcast the change of any user configuration variables.

UseCurrentPersona

`UseCurrentPersona()`

Forces the system to adjust the values of other user configuration variables to reflect the value of the `currentPersona` variable. You should call this function after changing the value of `currentPersona`. Remember to use `SetUserConfig` to change any user configuration variables. Note that this function does not broadcast the change of any user configuration variables.

Localizing Newton Applications Reference

This chapter describes the facilities the Newton system includes for localizing applications.

Constants and Data Structures

Localization information is stored in a specialized data structure called a locale bundle. The slots of a locale bundle are described in the next section.

The Newton system provides constants that specify the format of date and time strings. These constants are described in “Date and Time Format Specifications” (page 17-11).

Contents of a Locale Bundle

This section shows the slots of a locale bundle that you can access. Slots not described here are for internal use; you should not change their values or write your application to depend on their contents or existence.

String Slots

This section describes slots in the locale bundle that are used to display locale-specific strings. The strings stored in these slots vary according to the locale.

Your application can reference these slots directly to use the strings stored here as labels for text or fields. (See "GetLocale" (page 17-19) for how to get the current locale bundle.) You should never set the values of these slots directly except in your own custom locale bundle.

`postalCodeLabel`
: A string used to label postal codes. For example, the built-in Names application uses this slot to display the string `"Zip Code"` in the U.S. locale and the string `"Post Code"` in the Australia locale.

`longOrdinals`
: An array of strings intended to label items ordinally; for example, `"First"`, `"Second"` and so on. This is not currently used by the system.

`shortOrdinals`
: An array of short strings intended to label items ordinally; for example, `"1st"`, `"2nd"` and so on. This is not currently used by the system.

`distanceLabel`
: The string used to label distances. For example, this value is `"miles"` in the U.S. locale and `"kilometers"` in the Canadian locale.

`distanceLabelShort`
: The short version of the string used to label distances. For example, this value is `"km"`, as opposed to the corresponding `distanceLabelLong` value of `"kilometers"`.

`title`
: A string that identifies this bundle in the Country pop-up menu in the Locale preference panel. You can also pass this string to locale functions such as `SetLocale`. This may change in the future, however so you should use the locale symbol to identify locale bundles.

`regionLabel` The string used to label regions. For example, this value is `"State"` in the U.S. locale and `"Province"` in the Canada locale.

`cityLabel` The string used to label cities. For example, this value is `"City"` in the U.S. locale and `"Town"` in the United Kingdom locale.

Date Strings

These slots contain strings used by the system in the textual representation of various date values. The strings stored in these slots vary according to the locale.

Rather than using these values directly, you generally would employ the appropriate `dateTimeStrSpec` to format the output of functions that return date information. However, your application can reference these slots directly to use the strings stored here as labels for text or fields in your application. You should never set the values of these slots directly except in your own custom locale bundle.

`longDateFormat` The frame containing strings for the textual representation of the elements of the long (verbose) date format. This frame contains the slots shown in Table 17-1 (page 17-4).

`shortDateFormat`

The frame containing strings for the textual representation of the elements of all date strings other than those in the `longDateFormat` format. This frame contains the slots shown in Table 17-2 (page 17-6).

Table 17-1 LongDateFormat slots

Slot Name	Description
longDofWeek	An array of strings representing full names of the days of the week; for example, "Sunday", "Monday", "Tuesday", and so on.
abbrDofWeek	An array of strings representing abbreviated names of the days of the week; for example, "Sun", "Mon", "Tue", and so on.
terseDofWeek	An array of strings representing shorter abbreviations of names of days than are specified in the abbrDofWeek slot; for example, "Su", "Mo", "Tu", and so on.
shortDofWeek	An array of strings representing single-letter abbreviations of names of the days of the week; for example, "S", "M", "T", and so on.
longMonth	An array of strings representing full-text names of the months of the year; for example, "January", "February", "March", and so on.
abbrMonth	An array of strings representing abbreviated names of the months of the year; for example, "Jan", "Feb", "Mar" and so on.
longDateOrder	A dateTimeStrSpec value describing the order in which the elements of a long date are to appear; for example, month/day/year ("January 31, 1994") as opposed to day/month/year ("31 January 1994").

Table 17-1 `LongDateFormat` slots (continued)

Slot Name	Description
`longDateDelim`	An array of strings that represents the character separating the elements in the textual representation of a long date string; for example, the string `", "` (comma-space) used in the long date string `"January 31, 1994"`. The system automatically selects a delimiter from this array according to the relative positions of the string elements to be separated. The 0th array element precedes the date string; the first element specifies the delimiter to be placed between the first and second elements in the date string; the second element in the array specifies the delimiter used to separate the second and third elements in the date string; and so on.

Table 17-2 `ShortDateFormat` slots

Slot Name	Description
`shortDateOrder`	A `dateTimeStrSpec` value describing the order in which the elements of a short date are to appear; for example, day/month/year (`"31/1/94"`) as opposed to month/day/year (`"1/31/94"`).
`shortDateDelim`	An array of strings that represents the character separating the elements in the textual representation of a short date string; for example, the string `"/"` (forward slash) used in the short date string `"1/31/94"`. The system automatically selects a delimiter from this array according to the relative positions of the string elements to be separated. The 0th array element precedes the date string; the first element specifies the delimiter to be placed between the first and second elements in the date string; the second element in the array specifies the delimiter used to separate the second and third elements in the date string; and so on.
`dayLeadingZ`	A value of `kLeadZero` specifies that a leading zero is to be prefixed to representations of single-digit day values in short dates; for example, the `0` in `11/01/94`. A value of `kNoLeadZero` suppresses the use of the leading zero prefix.
`monthLeadingZ`	A value of `kLeadZero` specifies that a leading zero is to be prefixed to representations of single-digit month values in short dates; for example, the `0` in `01/11/94`. A value of `kNoLeadZero` suppresses the use of the leading zero prefix.
`yearLeadingZ`	A value of `kLeadZero` specifies that a leading zero is to be prefixed to representations of single-digit year values in short dates; for example, the `0` in `12/12/04`. A value of `kNoLeadZero` suppresses the use of the leading zero prefix.

Time Strings

These slots contain strings used by the system in the textual representation of various time values. The strings stored in these slots vary according to the locale.

Rather than using these values directly, you generally would use the appropriate `dateTimeStrSpec` to format the output of functions that return time information.

You should never set the values of these slots directly except in your own custom locale bundle.

`timeFormat` The frame containing strings for the textual representation of the elements of time formats.

This frame contains the slots shown in Table 17-3.

Table 17-3 TimeFormat Slots

Slot Name	Description
`timeSepStr1`	The string that represents the character separating the first and second elements in the textual representation of a time string; for example, the string `"."` (period) used in the time string `"23.59:59"`.
`timeSepStr2`	The string that represents the character separating the second and third elements in the textual representation of a time string; for example, the string `":"` (colon) used in the time string `"23.59:59"`.
`morningStr`	The string for annotating times from midnight to just before noon for a 12-hour clock cycle; for example, the string `" am"` (space-`am`) used in the time string `"8:00 am"`.
`eveningStr`	The string for annotating times from noon to just before midnight for a 12-hour clock cycle; for example, the string `" pm"` (space-`pm`) used in the time string "`8:00 pm`".

Table 17-3 TimeFormat Slots (continued)

Slot Name	Description
suffixStr	The string used as a suffix for textual representations of times for a 24-hour clock cycle. This slot is not currently used by the MessagePad; however, it might contain, for example, the string `" GMT"` to indicate the use of Greenwich Mean Time.
hourLeadingZ	A value of `kLeadZero` specifies that a leading zero is to be prefixed to representations of single-digit hour values. A value of `kNoLeadZero` suppresses the use of the leading zero prefix.
minuteLeadingZ	A value of `kLeadZero` specifies that a zero is to be prefixed to representations of single-digit minute values. A value of `kNoLeadZero` suppresses the use of the leading zero prefix.
secondLeadingZ	A value of `kLeadZero` specifies that a zero is to be prefixed to representations of single-digit second values. A value of `kNoLeadZero` suppresses the use of the leading zero prefix.
timeCycle	A value of `kCycle12` specifies the use of a twelve-hour clock cycle; the value `kCycle24` specifies the use of a 24-hour clock cycle.
midNightForm	The value `kUseHourZero` specifies the representation of midnight as the numeric string `"00:00"`. The value `kUseHour12` specifies the representation of midnight as the numeric string `"12:00"`. The value `kUseHour24` specifies the representation of midnight as the numeric string `"24:00"`.
noonForm	The only valid value is `kUseHour12`, which specifies the representation of noon as the numeric string `"12:00".`

Numeric Strings

These slots contain strings used by the system in the formatting the text of various numeric values. The strings stored in these slots vary according to the locale.

You should never set the values of these slots directly except in your own custom locale bundle.

`numberformat` The frame containing strings for the textual representation of the elements of numeric formats. This frame contains the slots shown in Table 17-4.

Table 17-4 NumberFormat Slots

Slot Name	Description
`decimalpoint`	The string that represents the decimal character in the textual representation of a numeric string; for example, the string `"."` (period) used in the currency string `"$123.45"`.
`groupSepStr`	The string that represents the character separating the groupings of numbers in the textual representation of a numeric string; for example, the string `","` (comma) used in the numeric string `"1,234"`.
`groupWidth`	The number of characters in each grouping of numeric characters separated by the `groupSepStr` character; for example, the string `"123,456,789"` is separated in groups of three.
`minusPrefix`	The string used as a prefix for textual representations of negative numbers; for example, the minus sign (`-`) in the string `"-123"`.
`minusSuffix`	The string used as a suffix for textual representations of negative numbers; for example, the minus sign (`-`) in the string `"123-"`.

Table 17-4 NumberFormat Slots (continued)

Slot Name	Description
`currencyPrefix`	The string used as a prefix for textual representations of currency values; for example, the dollar sign (`$`) in the string `"$123"`.
`currencySuffix`	The string used as a suffix for textual representations of currency values; for example, the dollar sign (`$`) in the string `"123$"`, as used for the locale Canada (French locale).
`decimalLeadingZ`	A value of `kLeadZero` specifies that a zero is to be prefixed to representations of single-digit decimal values; for example, the `0` in the string `"$0.12"`. A value of `kNoLeadZero` suppresses the use of the leading zero prefix.

Other Slots in Locale Bundles

The following slots are also contained in locale bundles:

`_proto` The locale bundle from which this frame inherits default attributes. Not all built-in locale bundles have this slot, but all locale bundles that you define must have it.

`distanceMeasure`

The unit of measure for distances. For example, this value is `miles` in the U.S. locale and `kilometers` in the Canadian locale. The built-in Time Zones application employs this value to display the distance between two cities in a format appropriate to the user's locale; your application can use it also.

`localeSym` A symbol that uniquely identifies the locale. You must have this slot in any locale bundle that you define. Use this symbol to identify this locale bundle in calls to locale functions such as `SetLocale`.

`firstDayofWeek` This integer value, ranging from 0 to 6, specifies the starting day of the week: 0 represents Sunday, 1 represents Monday, and so on. This information is used by, for example, `clMonthView`.

`postalCodeNumeric`
: Has the value `true` if postal code field accepts numeric input only; for example, in the U.S. locale, postal codes do not include alphabetic characters.

`wordBreakTable`
: For internal use; the word-selection table used to find word boundaries when selecting words.

`lineBreakTable`
: For internal use; the word-selection table used to find word boundaries when breaking lines of text.

`defaultPaperSize`
: Specifies the default paper size to used when formatting pages for printing and faxing; valid values are `'eightByEleven` and `'a4`.

`keycodeMapping`
: Specifies the Macintosh-style `'kchr'` resource that maps keys on the floating keyboard to appropriate keycodes used for input via keyboard to map codes to characters.

Date and Time Format Specifications

Many of the functions in “Formatted Date/Time Functions” (page 17-22) take a format specification as a parameter. You can use the pre-defined format specifications included in the system (described in “System-Defined Format Specifications” (page 17-11) or create your own using the constants described in “Constants to Create Your Own Specification” (page 17-13) and the function `GetDateStringSpec`.

Note that the locale changes the separators and order of elements for dates and times even when you specify a format.

System-Defined Format Specifications

The system provides a set of date and time format specifications you can use. They are stored in the `ROM_dateTimeStrSpecs` global. Table 17-5 shows the slots of this global.

Table 17-5 Format specifications in `ROM_dateTimeStrSpecs` global

Slot	Note	Example of format
`longDateStrSpec`	*	Wednesday, July 22, 1992
`abbrDateStrSpec`	1	Wed, Jul 22, 1992
`yearMonthDayStrSpec`	1	July 22, 1992
`yearMonthStrSpec`	1	July 1992
`dayStrSpec`	1	Wed, Jul 22
`monthDayStrSpec`	1	July 22
`numericDateStrSpec`	†	7/22/92
`numericMDStrSpec`	2	7/22
`numericYearStrSpec`	1,2	1992
`longMonthStrSpec`	1	July
`abbrMonthStrSpec`	1	Jul
`numericDayStrSpec`	1,2	22
`longDayOfWeekStrSpec`	1	Wednesday
`abbrDayOfWeekStrSpec`	1	Wed
`longTimeStrSpec`	‡	10:40:59 AM
`shortTimeStrSpec`	3	10:40 AM
`shortestTimeStrSpec`	3	10:40
`hourStrSpec`	3	10
`minuteStrSpec`	3	40
`secondStrSpec`	3	59

* Argument to `LongDateStr` function
† Argument to `ShortDateStr` function
‡ Argument to `TimeStr` function

Constants to Create Your Own Specification

This section shows the system-supplied constants available for specifying the elements and formats of date and time strings.

The system-supplied constants for specifying the elements of date and time strings are listed in Table 17-6. These are paired with the format constants in Table 17-7 (page 17-15).

Table 17-6 Elements of date strings

Constant	Notes
`kElementNothing`	Prints nothing.
`kElementDayOfWeek`	Gives the day of the week, using the strings given in the `LongDateFormat` slot of the active locale bundle. The slot used depends on the value paired with it: If paired with `kformatshort`, it uses `LongDateFormat.shortdofweek`. If `LongDateFormat.shortdofweek` is undefined then `LongDateFormat.tersedofweek` is used. If paired with `kformatterse`, it uses `LongDateFormat.tersedofweek`. If `LongDateFormat.tersedofweek` is undefined then `LongDateFormat.abbrdofweek` is used. If paired with `kformataddr`, it uses `LongDateFormat.abbrdofweek`. If `LongDateFormat.abbrdofweek` is undefined then `LongDateFormat.longdofweek` is used. If paired with `kformatlong`, it uses `LongDateFormat.longdofweek`.

Table 17-6 Elements of date strings (continued)

Constant	Notes
`kElementDay`	Gives the date of the month. Can only be paired with `kFormatNumeric`. If less than 10, the system checks in the active locale for `longdateformat.leadingZ` or `shortdateformat.leadingZ` to see if there should be a leading zero.
`kElementMonth`	Gives the month, using the strings given in the `LongDateFormat` slot of the active locale bundle. The slot used depends on the value paired with it: If paired with `kformatshort`, it uses `LongDateFormat.shortmonth`. Currently in U.S. locale, this has the same behavior as `kformatabbr` because `shortmonth` is not defined in the U.S. locale bundle. If paired with `kformatterse`, it uses `LongDateFormat.tersemonth`. Currently in the U.S. locale this has the same behavior as `kformatabbr` because the terse month not defined in the U.S. locale bundle. If paired with `kformatabbr`, it uses `LongDateFormat.abbrmonth`. If paired with `kformatlong`, it uses `LongDateFormat.longmonth`. If paired with `kformatnumeric`, it uses the numbers 1 through 12. If the month number is less than 10, the system checks `ShortDateFormat.monthLeadingZ` to see if a leading zero should be used.

Table 17-6 Elements of date strings (continued)

Constant	Notes
`kElementYear`	Gives the year. The format paired with this is ignored. If used used with `LongDateStr`, the full four digits representing the year are included in the resulting string. If used with `ShortDateStr`, then `LocaleBundle.ShortDateFormat.yearleading` is checked to see if the first two digits should be dropped.
`kElementHour`	Gives the hour. The format paired with this is ignored.
`kElementMinute`	Gives the minute. The format paired with this is ignored.
`kElementSecond`	Gives the second. The format paired with this is ignored.
`kElementAMPM`	Specifies “AM”, “PM”
`kElementSuffix`	Gives a 24-hour clock indicator, such as “GMT”
`kIncludeAllElements`	Includes the day name, month, day, and year, such as Wednesday July 24, 1996.

Table 17-7 Formats for date and time string elements

Constant	Element of string	Example of display
`kFormatLong`	full-length	“Wednesday”
`kFormatAbbr`	abbreviated	“Wed”
`kFormatTerse`	shortened abbreviation	“We”
`kFormatShort`	single-letter	“W”
`kFormatNumeric`	numeral	“1994”
`kFormatDefault`	Default from active locale bundle	

Localization Function Reference

These functions are used for localizing Newton applications.

Compile-Time Functions

These functions allow you to build an application for various language environments. "GetDateStringSpec" (page 17-29), which is listed with the utility functions, is also a compile-time function.

LocObj

`LocObj(`*obj*`,` *pathexpr*`)`

Returns the object specified by the *obj* parameter or, if the Language setting in the Project Settings dialog box is something other than English, the object specified by the pathname in the *pathexpr* parameter. It gets that object from a localization frame that you define. See "Defining a Localization Frame" (page 20-4) in *Newton Programmer's Guide* for details of defining such a frame.

obj	The object. This must be a constant.
pathexpr	The path to an alternative localized version of the object. This also must be a constant.

You can reference `LocObj` from within a function executed at run time, because `LocObj` is evaluated at compile time and replaced with the string or other object appropriate to the language setting.

This is a compile-time function. Because `LocObj` is evaluated at compile time, its parameters must be constants, not references to local variables that are created at run time.

See "Using LocObj to Reference Localized Objects" (page 20-4) in *Newton Programmer's Guide* for more information.

MeasureString

`MeasureString(`*str*`,` *fontSpec*`)`

Measures the length of a text string in a specified font. This is a compile-time function; if you want to measure a string at run time, use `StrFontWidth`.

str	The text to measure.
fontSpec	The font in which the text appears. You can specify the font using any constant or combinations of constants described in Chapter 7, “Text and Ink Input and Display Reference.” If you’re using your own font, you can pass a font frame.

The `MeasureString` function returns the length, in pixels, of the *str* parameter in the font specified by the *fontSpec* parameter.

This is a compile-time function. Because `MeasureString` is evaluated at compile time, its parameters must be constants, not references to local variables created at run time. You can also use the `LocObj` function as a parameter, since it is evaluated at compile time.

See “Measuring String Widths at Compile Time” (page 20-6) in *Newton Programmer's Guide* for an example of using this function.

SetLocalizationFrame

`SetLocalizationFrame(`*frame*`)`

Establishes the language frame for `LocObj` to use when the Language setting for a build is anything other than English.

frame	The language frame; that is, the hierarchy of objects that maps to the pathnames used by the `LocObj` function. At its first level, the language frame contains one or more slots, whose names are the Language codes that can be specified through the Project Settings dialog box. Each language slot contains all objects established for that language.

If you call `SetLocalizationFrame` more than once, the most recent language frame replaces the previous language frame.

This is a compile-time function. Because `SetLocalizationFrame` is evaluated at compile time, its parameters must be constants, not references to local variables that are created at run time.

See "Defining a Localization Frame" (page 20-4) in *Newton Programmer's Guide* for more information on using this function.

Locale Functions

These functions manipulate locale bundles:

AddLocale

`AddLocale(`*theLocaleBundle*`) // platform file function`

Adds the specified frame to the available locales.

IMPORTANT
This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

`call kAddLocaleFunc with (`*theLocaleBundle*`);`
▲

theLocaleBundle The locale bundle to install into the system.

FindLocale

`FindLocale(`*locSymbol*`) // platform file function`

Returns, from the available locales, the frame that has the specified symbol in its `localeSym` slot . If the symbol is not found, this function returns `nil`. Use this function to get a frame to be referenced by your custom locale bundle's `_proto` slot.

IMPORTANT
This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

`call kFindLocaleFunc with (`*locSymbol*`);` ▲

locSymbol The symbol of the locale bundle to retrieve, as specified by the symbol in the bundle's `localeSym` slot.

GetLocale

```
GetLocale()
```

Returns the current locale frame.

For more information, see the sections "Examining the Active Locale Bundle" (page 20-6) in *Newton Programmer's Guide* and "Contents of a Locale Bundle" (page 17-1) in this reference guide.

RemoveLocale

`RemoveLocale(`*locSymbol*`) // platform file function`

Removes the specified locale bundle from the available locales.

IMPORTANT

This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

`call kFindLocaleFunc with (`*locSymbol*`);`

▲

locSymbol The symbol of the locale bundle to retrieve, as specified by the symbol in the bundle's `localeSym` slot.

SetLocale

`SetLocale(`*locSymbol*`)`

Searches the system for the specified locale bundle and, if it is found, makes it the active locale bundle. The new active locale is returned; if no locale is found, `nil` is returned.

locSymbol The symbol of the locale bundle you want as the active locale, as specified by the symbol in the bundle's `localeSym` slot.

Date and Time Functions

These functions are grouped into three categories

- Those that deal with system clock values directly
- Those that format system clock values into strings or strings into system clock values
- Those that format system clock values into date frames or date frames into system clock values

System Clock Functions

These functions use the system clock value, which is either the number of minutes since midnight, January 1, 1904 or the number of seconds since midnight, January 1, 1993. They perform the following functions:

- Getting the current system clock value
- Setting the system clock
- Incrementing a system clock value on a monthly basis
- Converting a seconds system clock value to a minute system clock value, or the other way around
- Giving a value that lets you measure durations in increments of sixtieths of a second

IncrementMonth

`IncrementMonth(`*time*`,` *numMonths*`)`

Returns a time that is offset from the date of the original time by the number of months indicated by the second parameter. The return value is the number of minutes since midnight, January 1, 1904.

time	An integer giving a number of minutes since midnight, January 1, 1904.
numMonths	An integer that specifies a number of months.

For example, suppose you wanted to get the day and date for two months past February 14, 1996. You can use this code line:

```
ShortDate(IncrementMonth(stringtodate("2/14/96 3PM"), 2))
```

In the U.S. locale, this returns the string `"Sun 4/14"`.

SetTime

`SetTime(`*time*`)`

Sets the time of the system clock. This function always returns `nil`.

time	The time to which to set the system clock, specified as the number of minutes elapsed since midnight, January 1, 1904.

SetTimeInSeconds

`SetTimeInSeconds(`*time*`)`

Sets the time of the system clock. This function always returns `nil`.

time	The time to which to set the system clock, specified as the number of seconds elapsed since midnight, January 1, 1993.

Ticks

`Ticks()`

Returns a number of ticks; a tick is one-sixtieth of a second. There is no defined starting time for ticks; they are used to measure durations of time. Typically, you would call `Ticks`, do something or wait, then call `Ticks` again and compare the values to see how much time has passed.

Time

`Time()`

Returns an integer that indicates the time as the number of minutes since midnight, January 1, 1904.

TimeInSeconds

`TimeInSeconds()`

Returns the time in seconds as an integer. This is the number of seconds since midnight, January 1, 1993.

TimeInSecondsToTime

`TimeInSecondsToTime(`*secondsSince1993*`)`

Returns a time in minutes since midnight, January 1, 1904 based on a time in seconds since midnight, January 1, 1993.

secondsSince1993	A number of minutes since January 1, 1993.

TimeToTimeInSeconds

`TimeToTimeInSeconds(`*minutesSince1904*`,` *extraSeconds*`)`

Returns a time in seconds since midnight, January 1, 1993 based on a time in minutes since midnight, January 1, 1904.

minutesSince1904	A number of seconds since January 1, 1904.
extraSeconds	Any extra seconds that should be added to the *minutesSince1904* value to increase accuracy.

Formatted Date/Time Functions

These functions return formatted date and/or time strings. Some of the functions in this list format the string according to the active locale bundle; others take a format specification supplied as one of their arguments. See "Date and Time Format Specifications" (page 17-11) for details of format specifications; see "Functions that Take Format Specifications" (page 20-11) in *Newton Programmer's Guide* for information on how to use those format specifications.

DateNTime

`DateNTime(`*time*`)`

Returns the specified time as a string with the format such as *MM*/*DD*/*YYYY HH*:*MM*; for example, `10/23/1993 12:45`. The formats used for individual elements and delimiters in the returned string are determined by values in the active locale bundle.

time	The time in minutes since midnight, January 1, 1904, as returned by the `Time` function.

HourMinute

`HourMinute(`*time*`)`

Returns the value of the *time* argument as a string in the format *HH*:*MM*; for example, `12:45`. The formats for individual elements and delimiters in the returned string are determined by values in the active locale bundle.

time	The time in minutes since midnight, January 1, 1904, as returned by the `Time` function.

LongDateStr

`LongDateStr(`*time*`,`*dateStrSpec*`)`

Returns the date as a string in the specified format. For example:

With the U.S. locale:

```
LongDateStr(time(),
ROM_dateTimeStrSpecs.yearMonthDayStrSpec);
```

returns `"April 22, 1996"`

With the Canada (French) locale:

```
LongDateStr(time(),
ROM_dateTimeStrSpecs.yearMonthDayStrSpec);
```

returns `"22. Avril 1996"`

The active locale determines certain features of the returned string, specifically the order of elements and the separators used.

time The time in minutes since midnight, January 1, 1904, as returned by the `Time` function.

dateStrSpec A format specification returned by the `GetDateStringSpec` function or one of the format specifications found in `ROM_dateTimeStrSpecs`; see Table 17-5 (page 17-12) for those specifications.

ShortDate

`ShortDate(`*time*`)`

Returns the date as a string in the short format specified by the active locale. For example, `in the` U.S. locale `"Fri 12/25"`; in the German locale `"Mo 22.4"`. The formats used for individual elements and delimiters in the returned string are determined by values in the active locale bundle.

time The time in minutes since midnight, January 1, 1904, as returned by the `Time` function.

ShortDateStr

`ShortDateStr(`*time*`,` *dateStrSpec*`)`

Returns the date as a string in the format specified in `dateStrSpec`; for example, `"5/8/93"`.

The active locale bundle determines certain features of the returned string, specifically the order of elements and the separators used.

time The time in minutes since midnight, January 1, 1904, as returned by the `Time` function.

dateStrSpec A format specification returned by the `GetDateStringSpec` function or one of the format specifications found in `ROM_dateTimeStrSpecs`; see Table 17-5 (page 17-12) for those specifications.

StringToDate

`StringToDate(`*dateString*`)`

Parses a string for date or time information and returns the result as the number of minutes passed since midnight, January 1, 1904. The formats used for individual elements and delimiters in the input string are determined by values in the active locale bundle.

dateString The string to parse. If the year is omitted from the string, the current year is assumed. The following types of date/time strings can be parsed (the case of letters is not significant):

```
"12:05 a.m. sun, jan 2, 1992"
"jan 2, 1992"
"12:05 1/2/92"
"1/2/92"
"12:05 mon,1/2"
"1/2"
```

StringToDateFrame

`StringToDateFrame(`*str*`)`

Returns the input string as a date frame. See Table 17-8 (page 17-27) for details of a date frame.

This function is similar to `StringToDate`, with two significant differences:

- The `StringToDateFrame` function returns a date frame instead of the number of minutes since midnight, January 1, 1904. For example, the `StringToDateFrame` function returns the following frame when passed the string `"June 2"` as its argument:

```
{
year: nil,
month: 6,
date: 2,
dayOfWeek: nil,
hour: nil,
```

```
minute: nil,
second: nil,
daysInMonth: nil,
status: 0
}
```

- The `StringToDateFrame` function does not supply date or time elements missing from the input string. In the previous example, the `year`, `dayOfWeek`, `hour`, `minute`, and `second` slots are set to `nil` because the input string does not include these values.

This behavior can be useful for determining what's really in the input string. If you want to make certain that you have all slots filled, you can use `StringToDate` to convert the string to the number of minutes since midnight, January 1, 1904 and `Date` to convert that value to a date frame. For example:

```
Date(StringToDate("12:01a.m. 1/1/96")
```

StringToTime

`StringToTime(`*timeString*`)`

This function is similar `StringToDate`, except that it ignores any date information that may be given in the parameter, *timeString*, and uses, instead, the current date. That is, it returns the number of minutes from midnight 1/1/1904 until a given time on the day that the call is executed. For example, all of the following return the same value, assuming the current date doesn't change in between the calls:

```
StringToTime("12:01a.m. 1/1/96")
StringToTime("12:01a.m. 5/15/46")
StringToTime("12:01a.m.")
```

The formats for individual elements and delimiters in the input string are determined by values in the active locale bundle.

timeString	The string to parse for time information; any date information in this string is ignored.

TimeStr

`TimeStr(`*time*`,` *timeStrSpec*`)`

Returns the specified time as a string in the specified format. The seconds field is always 00.

The active locale bundle determines certain features of the returned string, specifically the order of elements and the separators used.

time	The time in minutes since midnight, January 1, 1904, as returned by the `Time` function.
timeStrSpec	A format specification returned by the `GetDateStringSpec` function or one of the format specifications found in `ROM_dateTimeStrSpecs`; see Table 17-5 (page 17-12) for those specifications.

Date Frame Functions

These functions use or produce a date frame with the format shown in Table 17-8.

Table 17-8 Date frame slots and values

Slot name	Example value
`year`	1993
`month`	1
`date`	24
`dayofweek`	0 (Sunday=0, Saturday=6, and so on)
`hour`	15
`minute`	38
`second`	30
`daysInMonth`	31

Date

`Date(`*time*`)`

Returns the specified time as a date frame. The second slot of the returned frame has a random value and should not be used.

time — The time expressed as a number of minutes since midnight, January 1, 1904, such as that returned by the `Time` function.

DateFromSeconds

`DateFromSeconds(`*timeInSeconds*`)`

Returns the specified time as a date frame.

timeInSeconds — The time expressed as a number of seconds since midnight, January 1, 1993, such as that returned by the `TimeInSeconds` function.

TotalMinutes

`TotalMinutes(`*dateFrame*`)`

Returns the time in minutes since midnight, January 1, 1904, when passed a date frame. You must pass in a date frame, or this function returns an error.

TotalSeconds

`TotalSeconds(`*dateFrame*`)`

Returns the time in seconds since midnight, January 1, 1993, when passed a date frame. You must pass in a date frame, or this function returns an error.

Utility Functions

These functions perform tasks related to the presentation of data in regionalized formats.

GetDateStringSpec

`GetDateStringSpec(`*formatArray*`)`

Returns a date or time format specification that can be passed in place of a predefined format from `ROM_datetimeStrSpecs` to one of the following built-in functions:

- `LongDateStr`
- `ShortDateStr`
- `TimeStr`

Because the `GetDateStrSpec` function is available at compile time only, its return value must be stored in a compile-time variable used to initialize an evaluate slot at run time. The slot value is then passed to date and time functions requiring the format spec at run time.

The order in which elements of a date or time string appear is not specified by the format specification, but by values stored in the active locale bundle. The delimiters that separate the various elements of the date or time string are also not specified in the format spec, but are also retrieved from the active locale bundle.

formatArray An array of two-element arrays. Each two-element array lists a single date or time element and a corresponding format to use to display that element. For example:

```
[[kElementMonth,kFormatAbbr],
[kElementDay, kFormatNumeric]]
```

The two-element subarrays can appear in any order; the order in which elements of the date or time string appear is defined in the active locale bundle, not by the format spec.

See the section "Constants to Create Your Own Specification" (page 17-13) for a complete listing of the values to use for the date or time element in each subarray, and an example of each as returned by one of the built-in date or time functions.

Note

This function is available in the Newton Toolkit development environment only at compile time; it is not available at run time. ◆

GetLanguageEnvironment

```
GetLanguageEnvironment()// platform file function
```

Returns a value indicating the language for which the ROM in the current Newton device is implemented. These values are summarized in Table 17-9.

IMPORTANT

This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

```
call kGetLanguageEnvironmentFunc with ();
```

▲

Table 17-9 ROM language codes

Language	Value
English	`0`
French	`1`
German	`2`

IsValidDate

`IsValidDate(`*date*`)`

Returns `TRUE` if the object passed is a valid date. Otherwise, returns `NIL`.

date Either a string or a date frame. The date is considered valid if it contains both a legal day or month. It checks for leap years. If the year is missing, February can have 29 days.

SetCountryClass

`SetCountryClass(`*countryName*`)`

Sets the class of a country name so that it can be automatically translated if it is placed in a soup that is used on a Newton with a different country ROM.

countryName A string that is the name of a country. This must be a name that exists in the current ROM. For example, if you are using a Newton device with a U.S. ROM, and you use `"Deutchland"` as a country name, this function will do nothing. You may want to check that the user has entered a valid country name. You can do that with this code:

```
if length ( GetCountryEntry(countryName) ) > 0
            then SetCountryClass(countryName)
```

Calling this function makes the country name universally recognizable to the Newton system. If you store the country name in a soup and the resulting soup entry is read on a Newton with a different ROM, the country can still be identified when `GetCountryEntry` is called. You should always call this function on a country name that you are going to store in a soup.

Routing Interface Reference

This chapter 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 "Item Frame" (page 22-2) in *Newton Programmer's Guide*.

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 sets the `appSymbol` slot in the received item to this value, and the original value of the `appSymbol` slot is stored in the `fromAppSymbol` slot in the received item frame.

`body` Required. This slot contains a NewtonScript object representing the data to send. For fax and print transports, this object should be referenced by the print format that will draw the page. Print formats should access this data using the expression `target` (not `fields.body`).

All application-specific data and information must be contained in the `body` slot. Do not add application-specific slots to the item frame.

`title` Optional. A string to be shown in the In/Out Box's view as the item's title. Note that you should not make this string so long that it wraps to the next line in the Out Box. 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 need to supply this slot. Note that for e-mail, this string is also shown as the message subject when the mail is viewed.

`toRef` Required for some transports (of the built-in transports, 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" (page 21-27) in *Newton Programmer's Guide*.

`cc` — Used by e-mail transports. 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 e-mail transports. 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 frame 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" (page 22-13) in *Newton Programmer's Guide*. If the format needs to get the sender name, it can do so from this name reference. If you specify this slot, it overrides the one provided by the transport.

`currentFormat` — Optional. A symbol representing the routing format to use 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 hides 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 in your application, the `hidden` slot is removed from the item frame by the `Send` function. ▲

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

`completionScript`

Optional. This slot is a Boolean. If set to `true`, the application is 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. 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.

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

`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 is used. This is obtained from the `currentPrinter` variable of the user configuration data. For more information on how to specify a printer, see the section "Specifying a Printer" (page 21-28) in *Newton Programmer's Guide*.

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 is printed. If `nil`, no cover page is printed. If this slot is omitted, the user preference setting is observed. Don't rely on an exact number of extra pages being printed as a result of setting this slot.

`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, `'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
appSymbol: symbol,      // used if defined in a view def
RouteScript: symbol or function,// function called if this
                              // action is chosen
GetTitle: function      // supplied instead of title slot
...
}
```

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 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, `nil`, or the symbol `'pickSeparator`. If this method returns `nil`, the action does not show up in the picker. If this method returns the symbol `'pickSeparator`, it includes a separator line in the list of actions. The `GetTitle` method allows you to return different titles, depending on the target item.

Note that your application can override the `GetTargetInfo` method (page 12-11) to return custom data.

Protos

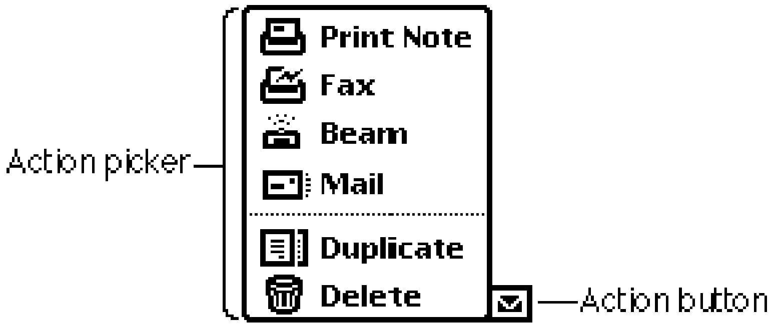
This section describes protos used in the Routing interface.

protoActionButton

This proto is used to include the Action button in a view. The context in which the Action button is placed establishes the context for routing actions.

When the user taps the Action button, a picker is dynamically created and displayed. The picker lists 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:



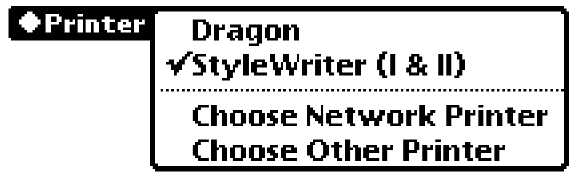
Slot descriptions

`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.

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.

protoPrinterChooserButton

This proto is used to include the printer chooser button in a view. When the user taps 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 selects 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.
`viewJustify`	Optional. The default setting is `vjLeftH + oneLineOnly`.

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

Routing Format Protos

The three routing format protos, `protoRoutingFormat`, `protoPrintFormat`, and `protoFrameFormat`, are used to create routing formats. They 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 `'routeFormat`. You shouldn't change it. (Note that some ROM versions use the symbol `'printFormat`.)
`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 include `'view`, `'frame`, `'text`, and `'binary`. For more information about these types, see Table 21-1 (page 21-7) in *Newton Programmer's Guide*. 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 for gathering 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. 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; 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), 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 is 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 (created by `CreateTargetCursor`) 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 that 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, a cursor will not be used, even if `storeCursors` is set to `true`. Instead, 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 use 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. You must specify only non-negative values, to

	make sure that you don't print off the page. The default is `{left:0, top:0, right:0, bottom:0}`.
`viewFlags`	Optional. The default setting is `vVisible + vReadOnly`.
`viewBounds`	Do not modify this slot.
`viewJustify`	Do not modify this slot.
`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 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*`)`

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 19-28). For more information about the item frame, see the section "Item Frame" (page 22-2) in *Newton Programmer's Guide*.
targetInfoFrame	The target information frame returned by the method `GetTargetInfo` (page 12-11).

The routing format protos provide a default `SetupItem` method that 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" (page 21-12) in *Newton Programmer's Guide*.

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, 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. 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*`, `*reserved*`)`

Returns a textual representation of the data to be routed. This method is typically called by transports that handle text-type data, such as e-mail transports.

item	The item frame. The data being routed is stored in the `body` slot of this frame. Because the `body` slot might contain an alias, in order to access it you should always call the `ResolveBody` format method on *item*. `ResolveBody` returns the data in the `body` slot whether or not it is referenced by an alias.
reserved	Ignore this parameter.

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. If no text is found by any of these methods, a string is returned that says no text is available for the item.

You can override this behavior by providing your own `TextScript` method.

Note that the `TextScript` method is not guaranteed to be called by a transport if the transport does not support the `'text` data type or if *item*`.body` is a subclass of `'string` or `'text`. (If *item*`.body` is a subclass of `'string` or `'text`, the transport may use the string data in *item*`.body` directly, rather than using the `TextScript` method to obtain it.)

TargetSize

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

Returns the size of the target object. You must override this method 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 uses the `EntrySize` function to determine the size of the object.

MakeBodyAlias

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

Returns an alias for the target object. You must override this method 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`.

The alias object that you return must have two slots:

- a `class` slot whose value is the class of the target object
- an `_ioalias` slot whose value is the alias you've constructed

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

ResolveBody

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

Resolves an alias created by `MakeBodyAlias`. You must override this method 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.

If the body slot contains an alias that cannot be resolved, `ResolveBody` returns `nil`.

ViewSetupChildrenScript

format:`ViewSetupChildrenScript()`

Sets 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`, the `ViewSetupChildrenScript` method is called for each subsequent page as well.

At the beginning of this method, don't forget to call the inherited method (`inherited:?ViewSetupChildrenScript`) so that the proto behavior is preserved before your own code is executed.

PrintNextPageScript

format:`PrintNextPageScript()`

Sets up the print view for the next page of data. 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()` shows the view.

Typically, you do this by keeping track of what data has been routed so far. 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 2, "Views Reference."

Instead of setting up a new group of child views and calling `RedoChildren`, you might want to change the contents in the existing views. Use the `SetValue` function to change the values in individual views.

While there is more data to route, `PrintNextPageScript` 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" (page 21-19) in *Newton Programmer's Guide*.

GetCursorFormat

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

Returns 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*`,` *reserved*`)`

Allows the print format to perform initialization operations. You can supply this method in your print format to perform any lengthy initialization operations that you want to do before a connection is made. This method is guaranteed to be called before a connection is made. This is helpful for reducing the chance of a fax timeout.

item — The Out Box item frame. The data being routed is stored in the `body` slot of this frame. Because the `body` slot might contain an alias, in order to access it you should always call the `ResolveBody` method on *item*. `ResolveBody` returns the data in the `body` slot whether or not it is referenced by an alias.

reserved — Ignore this parameter.

When the `FormatInitScript` message is sent, the message receiver is not the format frame itself. The pseudo-variable `self` references a temporary frame based on your registered format. The print format view is based on this temporary frame that is based on your registered format frame. Your `FormatInitScript` method can store data in `self` for use in the `ViewSetupFormScript` method or other view methods. Your format's view methods will be able to access those slots using prototype inheritance.

For more information on using this method and faxing, see the section "Printing and Faxing" (page 21-19) in *Newton Programmer's Guide*.

CountPages

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

Returns the number of pages. If possible, you should override this method of `protoPrintFormat` to return the number of pages in the fax (not including the cover page, if present).

item — The Out 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 on faxing, see the section "Printing and Faxing" (page 21-19) in *Newton Programmer's Guide*.

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 (for sending, by the user) 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 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" (page 21-26) in *Newton Programmer's Guide*. 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
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 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?
// transport-specific slots
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?
}
```

Some of the slots in the item frame shown here are transport-specific. Other transports may define additional slots. For more details, see "Item Frame" (page 18-1).

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` method (page 12-11).

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

GetDefaultFormat

app:`GetDefaultFormat(`*transportSym*`, `*target*`)`

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

transportSym A symbol identifying a transport.

target A frame, which is the `target` slot from the target information frame returned by the `GetTargetInfo` method (page 12-11) and verified by the `VerifyRoutingInfo` method.

This method is used to get the default format symbol for a transport and target item. It should return a format symbol or `nil`, if none is 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 *transportSym*. If the transport is found, it returns the format symbol 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 in your application base view, 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` evaluates to the application base view of the application that sent the item.

SetDefaultFormat

app`:SetDefaultFormat(`*transportSym*`, `*target*`, `*formatSym*`)`

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

transportSym	A symbol identifying a transport.
target	A frame, which is the `target` slot from the target information frame returned by the `GetTargetInfo` method (page 12-11) and verified by the `VerifyRoutingInfo` method.
formatSym	A routing format symbol. This is the value of the `symbol` slot in the 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 *formatSym* in this slot in `self` (the application base view):

`lastFormats.`*transportSym*

If you implement this method in your application base view, 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` evaluates 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 (page 19-28).

targetInfo This parameter must be a frame, containing `target` and `targetView` slots, as returned by the `GetTargetInfo` method (page 12-11).

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 can 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);
```

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. For more information about handling multiple-item target objects, see “Using the Built-in Overview Data Class” (page 21-14) in *Newton Programmer’s Guide*.

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 is automatically 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 just skips over the unresolved item. Subsequent calls to the cursor methods `Next` and `Prev` skip over the unresolved item.

The cursor object returned by this function is not 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 cursor methods `Entry` (page 9-61), `Next` (page 9-63), and `Prev` (page 9-64), described in Chapter 9, "Data Storage and Retrieval Reference." The use of other cursor methods is not supported.

TargetIsCursor

`TargetIsCursor(`*target*`)`

Returns `true` if the target object to be routed is a multiple-item target object, or returns `nil` if it's not.

target The target object to be routed.

You can use the function `GetTargetCursor` to obtain the cursor, even if `TargetIsCursor` returns `nil`.

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 and wants to receive any messages that arrived for it before it was installed. If your application uses an `AutoPutAway` method, you can call the `AppInstalled` function from your application part `InstallScript` function to let the system know that the application is present.

appSymbol A symbol identifying your application.

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.

Note that you must call the `AppInstalled` function using a deferred action, like this:

```
AddDeferredCall(GetGlobalFn('AppInstalled),[kAppSymbol]);
```

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

app`:GetActiveView()`

Returns the view to which the `GetTargetInfo` message (page 12-11) should be 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. This is useful if there is more than one routing slip displayed, or if the `GetTargetInfo` method must be executed in a specific context.

The default `GetActiveView` method implemented in the root view returns the current receiver (`self`), which is the view to which this message is sent. If this is not appropriate for your application, 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(`*targetInfoFrame*`)`

Returns the value of the `routeScripts` slot, using full proto and parent inheritance and starting in the context of `self`.

targetInfoFrame The target information frame returned by the method `GetTargetInfo` (page 12-11).

The system calls this method to build the list of application-specific routing actions to show in the Action picker. When this method is called, `self` is the Action button view.

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" (page 21-22) in *Newton Programmer's Guide*.

RegAppClasses

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

Registers an application to accept data of the specified classes.

appSymbol	A symbol identifying your application or transport that 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 the applications that have registered to handle items with that data class. The user can choose which application the item should be put away to. If the user chooses your application, it is sent the `PutAwayScript` message, with the item to 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, it is compared with the `title` slot in the incoming item. If the string in the `title` slot begins with the *test* string, 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.

TransportNotify

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

Sends a message to a transport or to all transports.

transportSym — A symbol identifying the transport to which you want to send a message. You can specify a transport group symbol to send the message to the current (last-used for sending, by the user) 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 *transportSym* is not the symbol `'_all` and the method does not exist in the transport, the symbol `'NoMethod` is returned.

If *transportSym* 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 three messages that the system currently sends to transports by using `TransportNotify`. They are `AppOpened` (page 19-10), `AppClosed` (page 19-8), and `AppInFront` (page 19-9).

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.

▲ **WARNING**
This function unregisters all data classes registered for the application—even those that may have been registered by other formats. ▲

It is recommended that you use `UnRegTheseAppClasses` instead of this function in most cases.

UnRegInboxApp

`UnRegInboxApp(`*appSymbol*`)`

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

appSymbol	A symbol identifying your application.

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 also unregisters 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 defined in an application to implement particular features.

AutoPutAway

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

Lets an application automatically put away a received item.

When an item is received by the In Box, and the In Box can identify an application that can receive the item, the In Box sends the base view of the 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, depending on the user's preference.

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

PutAwayScript

app:`PutAwayScript(`*item*`)`

Lets an application put away a received item as a result of the user choosing to put it away.

When the user is viewing an In Box item and taps the Put Away button, and the In Box can identify one or more applications that can receive the item, the user is allowed to choose the application to which it is sent. The In Box sends the base view of that application the `PutAwayScript` message. This gives an application the opportunity to do something with the item.

item A frame that is the In Box 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 saved or deleted from the In Box, depending on the user's preference as set in the Put Away slip.

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

If your application defines this method, it must 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.

Some transports send multiple-item target objects. 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.

ItemCompletionScript

app`:ItemCompletionScript(`*item*`)`

Alerts an application when an item's state changes or when errors occur while the item is being sent.

item The In/Out Box item whose state changed.

This message is sent to the base view of an application. It is sent only if the `completionScript` slot in the item frame is set to `true`. To take advantage of this callback mechanism, you must set the `completionScript` slot.

VerifyRoutingInfo

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

The system sends this message to the base view of your application when the routing slip is opened after the user chooses an action. 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` method (page 12-11).

item An item frame, as obtained from the transport method `NewItem` (page 19-28). 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" (page 22-2) in *Newton Programmer's Guide*.

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`.

Transport Interface Reference

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

Constants

This section describes constants.

Icon Constants

The magic pointer constants listed here reference icons that can be used for the `icon` slot in a `routeScripts` frame or the `groupIcon` slot of a transport.

```
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

This section describes transport-related protos.

protoTransport

This is the basic transport object.

Slot descriptions

`appSymbol` Required. A symbol that identifies this transport in In/Out Box soup items and to the system. The symbol must be unique, so it is recommended that you append your registered developer signature.

`title` Required. A string that is the name that identifies this transport to the user.

`dataTypes` Optional. 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. If you do create a custom data type, be sure to append your developer signature to make it unique. 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 group the transport belongs to, if it belongs to one. 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, you can include a unique icon for your transport in the `icon` slot. The following magic pointer constants reference built-in bitmaps to use with the built-in transport groups: `ROM_RouteMailIcon`, `ROM_RoutePrintIcon`, `ROM_RouteFaxIcon`, `ROM_RouteBeamIcon`.
`routingSlip`	Optional. Used for transports that send data. A template for the routing slip. See “Providing a Routing Slip Template” (page 22-26) in *Newton Programmer’s Guide*. 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 get the default template that shows the item title, transport, and item size. See "Providing a Routing Slip Template" (page 22-26) in *Newton Programmer's Guide*.

`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 "Providing a Preferences Template" (page 22-33) in *Newton Programmer's Guide*.

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

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

`modalStatus`
: Optional, a Boolean. `True` means that you want modal status dialogs that don'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. You must keep these same slots, but you can change the strings. 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 or change the existing slot names.) 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 "Setting the Address Class" (page 22-6) in *Newton Programmer's Guide*.

`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 19-29).

`allowBodyCursors`
: A Boolean value that indicates if the transport can handle a multiple-item target object in the `body` slot of an item in a send request. If the transport can parse and handle a multiple-item target object, set this slot to `true`. If this slot is set to `nil`, the In/Out Box never sends the transport a multiple-item target object in the body slot; it always parses the object ahead of time and sends the transport multiple send requests—one for each item. For more information about storing multiple-item target objects, see “Storing Multiple Items” (page 21-14) in *Newton Programmer’s Guide*.

`ownerApp`
: The application that is managing the transport. This slot is set by the system, and typically refers to the In/Out Box.

Note

In Newton OS version 2.0, the `ownerApp` slot must be set up by using the NTK platform file function `kSetupOwnerAppFunc` in the transport `InstallScript` method. ◆

`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 19-1. To override this frame, you must construct an identical one with different values, though you can add your own additional slots also. You can’t use a `_proto` slot in the default frame since this slot is stored in a soup and `_proto` slots 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 19-1 Preferences slots

Slot	Description
autoStatus	A Boolean. True means that you want the protoTransport to open and close the status slip 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.
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 (page 19-23).
inboxFiling	A symbol indicating the In Box folder in which to file an item 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 (this is essentially the same effect as nil). Note that filing doesn't occur until after the In/Out Box is closed. The default is the symbol 'same.
outboxFiling	A symbol indicating the Out Box folder in which to file an item 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 occur 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 always to send items immediately (corresponds to the "Send now" preferences choice). Specify the symbol 'later to force the button always to 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(`*senderSym*`)`

Notifies the transport that an application has closed. The transport owner application sends this message to all transports when it closes.

senderSym A symbol identifying the application that closed. The sender application is usually the transport owner (In/Out Box).

You should respond to this message only if the *senderSym* parameter identifies the owner of your transport. Use code like the following to check this:

```
If transport.ownerApp <> GetRoot().(senderSym) then
   return;
```

Note
In Newton OS version 2.0, the `ownerApp` slot must first be set up by using the NTK platform file function `kSetupOwnerAppFunc` in the transport `InstallScript` method. ◆

The `AppClosed` method is not defined by default in `protoTransport` since 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 "Application Messages" (page 22-19) in *Newton Programmer's Guide*.

AppInFront

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

Notifies the transport that an application is or is not the frontmost application. The transport owner sends this message to all transports when it becomes frontmost or is no longer frontmost (but not when it is opened or closed). You must use `AppOpened` and `AppClosed` to catch when the transport owner opens and closes.

inFront	A Boolean. This value is set to `true` if the application is now the frontmost application, or to `nil` if the application is no longer frontmost.
senderSym	A symbol identifying the application whose frontmost status has changed. The sender application is usually the transport owner (In/Out Box).

You should respond to this message only if the *senderSym* parameter identifies the owner of your transport. Use code like the following to check this:

```
If transport.ownerApp <> GetRoot().(senderSym) then
   return;
```

Note

In Newton OS version 2.0, the `ownerApp` slot must first be set up by using the NTK platform file function `kSetupOwnerAppFunc` in the transport `InstallScript` method. ◆

The `AppInFront` 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 "Application Messages" (page 22-19) in *Newton Programmer's Guide*.

AppOpened

transport:`AppOpened(`*senderSym*`)`

Notifies the transport that an application has opened and is interested in data from the transport. The transport owner application sends this message to all transports when it opens.

senderSym A symbol identifying the application that opened. The sender application is usually the transport owner (In/Out Box).

You should respond to this message only if the *senderSym* parameter identifies the owner of your transport. Use code like the following to check this:

```
If transport.ownerApp <> GetRoot().(senderSym) then
   return;
```

Note

In Newton OS version 2.0, the `ownerApp` slot must first be set up by using the NTK platform file function `kSetupOwnerAppFunc` in the transport `InstallScript` method. ◆

The `AppOpened` method is not defined by default in `protoTransport` since it's transport-specific. If you want to respond to the `AppOpened` message, you must define this method in your transport.

For more information about using the `AppOpened` method, see "Application Messages" (page 22-19) in *Newton Programmer's Guide*.

CancelRequest

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

Requests the transport to cancel the current send or receive operation. This method must be defined by all transports.

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 but the power was lost and the shutdown was not handled cleanly.

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

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

This method should return a non-`nil` value if it is OK to turn off power immediately after the method returns, or `nil` if it is not. 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 "Canceling an Operation" (page 22-13) in *Newton Programmer's Guide*.

CanPutAway

transport`:CanPutAway(`*item*`)`

Allows your transport to add a put away option for the 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). You don't need to implement this method.

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`, the Put Away choice is not included in the In/Out Box Tag picker. If there is one option, the Put Away choice appears and, if selected, the single option appears in the Put Away slip. If there are multiple options, 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, the method adds no options.

For an application not previously registered to handle data of the item's class, to put the item away, return the `appSymbol` of that application from this method. This adds the application as a put away option for the user to choose.

For the item to be put away by a particular application (or even by your transport), and to display a different name to the user in the Put Away picker, 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 lets your transport put the item away to itself and do some special handling (it must define the `PutAwayScript` method), 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 another put away alternative to those already available to the user for the item.

CheckOutbox

transport`:CheckOutbox()`

Causes 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`.

The return value of this method is unspecified. Do not rely on it.

Do not override this method.

Note that if there are no items to send, the system displays a slip explaining that to the user. You can use the function `QuietSendAll` (page 19-50) to do the same thing but avoid the user alert if there is nothing to send.

CloseStatusDialog

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

Closes the status slip.

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` reopens the status slip. If the user closes the status slip, it remains closed for the remainder of the current communication transaction.

ConnectionDetect

transport`:ConnectionDetect()`

Lets the transport control the operation of the Send button in the routing slip. This message is sent to a transport when the routing slip is displayed.

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 to send 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 for choosing between now or later.

To force the Send button to send now or later, you can implement the `ConnectionDetect` method. 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 19-7) from the transport configuration frame, obeying the user preference setting. You can override this method to force a different behavior.

GetConfig

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

Returns a value from the transport preferences.

prefName A symbol identifying a transport preferences item.

GetDefaultOwnerStore

transport:`GetDefaultOwnerStore()`

Returns 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*`)`

Returns the name of the folder where an item should be filed. 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 occurs after an item is sent or put away.

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*`)`

Returns a string representing the sender of the item.

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

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 let you 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`, 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`.

Supply this method only if the default behavior doesn't suit your needs.

GetItemInfo

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

Returns 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, 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. It calls

`GetItemTime` and `GetToText` (for Out Box items) or `GetFromText` (for In Box items) to let your transport customize the sender or recipient information.

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

GetItemStateString

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

Returns a status string based on the state of the specified item.

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

This method first checks if the item has an error, then it checks if the item is a remote item, and finally it checks the *item*`.state` slot. It fetches the appropriate string from the `itemStateMsgs` frame, which is `itemStateMsgs.error` if the item has an error, `itemStateMsgs.remote`, if the item is remote, and otherwise is based on the value of the *item*`.state` slot.

GetItemTime

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

Returns a string containing time and date information for the item.

Define this method 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 extracts the time and date from the `timeStamp` slot in the item frame. Supply this method only if the default behavior doesn't suit your needs.

This method is called by `GetItemInfo`.

GetItemTitle

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

Returns a string that is the title of the item. This method gets the string by calling the `StringExtract` method of 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*`)`

Returns 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, 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()`

Returns the status string based on the current status. This method fetches the string from the `dialogStatusMsgs` frame.

GetTitleInfoShape

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

Returns 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, 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.

If you want to change the contents of the header, it's recommended that you use the methods `GetItemTitle`, `GetItemInfo`, and `GetItemStateString`. You can override `GetTitleInfoShape` to do something different, such as add special graphics to the header, but this will change the user interface in a nonstandard way.

If you override `GetTitleInfoShape`, it's not recommended that you return a shape that looks radically different from the existing design of the item header. Your header should conform as closely as possible to the existing item header to keep the user experience similar.

GetToText

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

Returns a string representing the recipient's name, address, and/or other information.

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

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 let you supply 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`, 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`.

Provide this method only if the default behavior doesn't suit your needs.

GetTransportScripts

transport:`GetTransportScripts(`*target*`)`

Lets your transport add items to the In/Out Box Tag picker.

This message is sent to your transport when the user selects an In/Out Box item and taps the Tag button in the In/Out Box (the button that looks like a tag). You don't need to implement this method.

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 function `GetTargetCursor` (page 18-25) to return a cursor to *target* in case it is a multiple-item target object.

The `GetTransportScripts` method must return an array of frames that describe new items to add to the In/Out Box Tag picker. The array is exactly the same as the `routeScripts` array (page 18-6) that adds 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 function that is called if this action is selected by the user. 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.

Alternatively, in this slot you can specify a symbol identifying a transport method. If you do this, you must also include an `appSymbol` slot in this frame that contains your transport symbol.

`appSymbol` Your transport symbol. This slot is needed only if you specify a symbol in the `routeScript` slot.

`GetTitle` 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, `nil`, or the symbol `'pickSeparator`. If this method returns `nil`, the action does not show up in the picker. If this method returns the symbol `'pickSeparator`, it includes a separator line in the list of actions. The `GetTitle` method allows you to return different titles, depending on the target item.

For more detailed information about the items in the array, see “Providing Application-Specific Routing Actions” (page 21-22) in *Newton Programmer’s Guide*.

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()`

Catches exceptions on standard transport methods. This method is the default exception handler for transports. 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, 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 to, you can override the `HandleThrow` method to implement a different way of handling exceptions.

Also, `HandleThrow` calls other functions that you can override to modify its functionality, including `IgnoreError` and `HandleError`.

IgnoreError

transport:`IgnoreError(`*error*`)`

Lets your transport 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 it 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*`)`

Performs transport initialization operations that you define.

symbol The transport `appSymbol` that was passed to `RegTransport`.

This message is sent to the transport when it is registered in the system by `RegTransport`. The `InstallScript` method lets the transport perform any necessary initialization operations.

If you define this method, within it you must call the inherited method, like this:

```
inherited:?InstallScript(kAppSymbol);
```

IOBoxExtensions

transport:`IOBoxExtensions(`*item*`,` *target*`,` *viewDefs*`,` *reserved*`)`

Lets your transport add functionality to items in the In/Out Box by modifying the list of view definitions available for an item. This message is sent to your transport when an item belonging to the transport is displayed in the In/Out Box.

item	A frame that is the In/Out Box entry.
target	The target frame within *item* (usually the `body` slot).
viewDefs	A frame containing view definitions found by the system for the current data definition.
reserved	Ignore the data passed in this parameter.

Your transport can add to or delete from the *viewDefs* frame. Do not replace this frame.

If you want to change the view definition the item uses, 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` if it is in the Out Box. If the item is not an In/Out Box entry, the return value is undefined.

item	An item frame.

IsLogItem

transport`:IsLogItem(`*item*`)`

Returns `true` if the *item* has been logged, or `nil` otherwise.

item	An item frame.

ItemCompleted

transport`:ItemCompleted(`*item*`, `*state*`, `*error*`)`

Processes an item after the transport finishes with it.

Send this message after the transport finishes operating on an item, whether sending or receiving, with or without errors. Use this method 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 19-5). 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.

This method returns the item if *state* is `'received`. In all other cases, 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 18-33) 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 *error* is not `nil` and `IgnoreError` returns `nil`, `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, turns the item into a log entry (calls `MakeLogEntry`), or deletes the item from the Out Box, depending on the error conditions and 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 reply to an item, change the status to `'pending` and call `ItemCompleted`; the item is 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*`)`

Alerts a transport that an item is about to be deleted. 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 is always a single item, not a multiple-item target object.

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*`)`

Alerts a transport that an item has been duplicated. 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 is always a single item, not a multiple-item target object. 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*`)`

Alerts the transport that an item has been put away by an application.

This message is sent to a transport by the In/Out Box right after an item has been put away by an application. It provides an opportunity for the transport to take some action.

item A frame that is the item put away.

ItemRequest

transport:`ItemRequest(`*request*`)`

Gets an item, or the next item in the queue, from the In/Out Box.

You call this method from `SendRequest` or `ReceiveRequest` to get an item. If there is an item frame to be sent or a remote item to be received, the item 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 18-26) and `GetTargetCursor` (page 18-25) 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. After resolving each item but before sending it, you should send the message `SetupItem` to the item's format.

If your transport cannot handle body data that consists of multiple items, you must set the `allowBodyCursors` slot to `nil`.

If the `body` slot of an item originally contained an alias to a single item, the alias is automatically resolved by `ItemRequest`. That is, single items returned by `ItemRequest` always contain a `body` slot that is not an alias.

Note that you can save aliases to entries returned by `ItemRequest`. Later, when using them, make sure that `ResolveEntryAlias` returns a non-`nil` value, and that the item `state` slot is set as expected.

MakeLogEntry

transport`:MakeLogEntry(`*logItem*`, `*item*`)`

Lets your transport make a log entry for an 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(`*reserved*`)`

Notifies the user that there is nothing to send. 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.

reserved Ignore this parameter.

NewFromItem

transport:`NewFromItem(`*item*`)`

Returns a new item frame 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. It 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 these 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, you can set the target application differently from the originating application.

For more information about using the `NewFromItem` method, see "Obtaining an Item Frame" (page 22-13) in *Newton Programmer's Guide*.

NewItem

transport:`NewItem(`*context*`)`

Returns 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, or `nil`. During a send operation, the In/Out Box sets this argument to the application base view of the sending application, or to `nil`. If *context* is not `nil`, `NewItem` sets the `item.appSymbol` slot to the `appSymbol` found in *context*.

For more information about using the `NewItem` method, see "Obtaining an Item Frame" (page 22-13) in *Newton Programmer's Guide*.

If you override this method, be sure to call the inherited method first, in your version.

NormalizeAddress

transport:`NormalizeAddress(`*nameFrame*`)`

Converts 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, 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. The translation can consist either of appending a string to the given address or of passing it to a function object that 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 19-2 lists the built-in e-mail classes and the kind of translation done for each. If a string is listed as the translation, that string is appended to the given e-mail address, if it is not already part of that address.

Table 19-2 E-mail address translations

E-mail class	Translation done
`string`	nothing done
`|string.email|`	nothing done
`|string.email.internet|`	nothing done
`|string.email.aol|`	"@aol.com"
`|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 the string "@compuserve.com" is appended to the address if it is not already part of it.
`|string.email.eworld|`	"@eworld.com"

PowerOffCheck

transport:`PowerOffCheck(`*why*`)`

Displays an alert to the user that the system wants to power off. 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 it returns `nil`, the power-off sequence is canceled.

For more information, see “Power-Off Handling” (page 22-20) in *Newton Programmer’s Guide*.

QueueRequest

transport:`QueueRequest(`*doWhat*`,` *newRequest*`)`

Queues a send or receive request 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 calls when the state of the transport returns to idle. It passes *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 eventually returns 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 "Handling Requests When the Transport Is Active" (page 22-12) in *Newton Programmer's Guide*.

ReceiveRequest

transport`:ReceiveRequest(`*request*`)`

Requests the transport to receive items. The In Box sends this message to the transport. Define this method if receiving is supported by the transport.

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 a user request 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 "Receiving Data" (page 22-9) in *Newton Programmer's Guide*.

SendRequest

transport:`SendRequest(`*request*`)`

Requests the transport to send items. The Out Box sends this message to the transport. Define this method if sending is supported by the transport.

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 19-3.

Table 19-3 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 19-26) to get the item (or next item) to send. In your `SendRequest` method, keep calling `ItemRequest` until it returns `nil`, signaling 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 “Sending Data” (page 22-8) in *Newton Programmer’s Guide*.

SetConfig

transport`:SetConfig(`*prefName*`, `*value*`)`

Sets a value for the transport preferences.

prefName	A symbol identifying a transport preferences item.
value	A value to set in the *prefName* slot.

SetStatusDialog

transport`:SetStatusDialog(`*newStatus*`, `*name*`, `*values*`)`

Sets the current state of the transport and displays a status view to the user.

newStatus — Can be any symbol, such as `'Disconnected`, `'Connecting`, `'Connected`, `'Sending`, `'Receiving`, `'Disconnecting`, or `'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 “Providing a Status Template” (page 22-21) in *Newton Programmer’s Guide*. If you specify `nil`, the last symbol used is assumed; 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 "Providing a Status Template" (page 22-21) in *Newton Programmer's Guide*.

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*`)`

Lets your transport 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*`)`

Lets the transport modify each item within a multiple-item target object, during a send operation.

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.

item	An item submitted for sending, after it has been passed to the `SetupItem` method of the routing format. This is always 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, which contains a multiple-item target object in its `body` slot.
entry	A resolved entry from *multiItem,* 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.

When only a single item (not a multiple-item target object) is submitted to the Out Box, `VerifyRoutingInfo` is not called. In this case, if you need to modify the item before it is sent, you can do this in the routing slip method `PrepareToSend`.

protoTransportHeader

This proto provides a template for the routing information view. For more information about creating a routing information view, see “Providing a Routing Information Template” (page 22-25) in *Newton Programmer’s Guide*.

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.
`addedHeight`	Optional. An integer representing the additional height you are adding to the view, in pixels. The default is 0. You must specify this slot if you are adding additional child views to the routing information view. Note that you can specify this slot dynamically, in the `BuildText` method or before the inherited `ViewSetupFormScript` method is called.
`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()`

Adds lines of text to the header view.

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*`)`

Constructs a line of text to add to the header view. 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.

The return value of the `AddText` method is unspecified. Do not rely on it.

InfoChanged

context:`InfoChanged(`*changed*`)`

Notifies another view when the routing information view is closed. 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 is `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 “Providing a Routing Slip Template” (page 22-26) in *Newton Programmer’s Guide*.

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 eventually becomes 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.

IMPORTANT

The `target` object is always a frame; it is never an alias that needs to be resolved. In some cases, this object may be supplied by the In/Out Box, and not by an application. In those cases, it may have been processed somehow by the In/Out Box. Do not modify the `target` frame; it is for read-only use. ▲

`activeFormat`
: The currently selected routing format.

`transport`
: The transport object.

`editing`
: A Boolean. This slot is set to `true` if the slip is opened for editing. This occurs when the user readdresses an item in the Out Box or when the slip is reopened after a "send now" operation fails with an error. In the editing state, the format picker and Send button are hidden, and the user cannot select a different transport from the transport picker in the routing slip (if there are multiple transports in the same group).

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. This value is platform-specific and may vary according to the current screen orientation.

`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. This value is platform-specific and may vary according to the current screen orientation.

`bottomIndent` — Optional. An integer that is the space below the envelope image, in pixels. The default is 28, or 49 if there is more than one routing format (then the routing format picker is included). 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` . If you need to override one of these methods, be sure to call the inherited method also.

The following methods of `protoFullRouteSlip` are of interest.

BottomOfSlip

routingSlip`:BottomOfSlip()`

Returns the vertical 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*`)`

Notifies the routing slip view that the user chose 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, setting the `currentFormat` slot in the item, calling the routing format's `SetupItem` method, opening an auxiliary view, if one is defined in the routing format, and saving the chosen routing format in the application base view.

If the `FormatChanged` method returns `true`, the default code stops. The assumption in the latter case is that you've done all the necessary processing in your `FormatChanged` method.

OwnerInfoChanged

routingSlip`:OwnerInfoChanged()`

Notifies the routing slip view that the sender pop-up view changed, so you can catch any changes. The sender pop-up view is the sender's name and worksite location, 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.

In the `OwnerInfoChanged` method, you can obtain any changes by checking slots in which you are interested in the user configuration frame, using the `GetUserConfig` function. For example, the area code at the user's location can be found in the `currentAreaCode` slot. For a list of slots in the user configuration frame, see "User Configuration Variables" (page 16-101).

PrepareToSend

routingSlip`:PrepareToSend(`*when*`)`

Notifies the routing slip view that the user selected Now or Later from the Send 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 sending it.

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*`)`

Continues the process of submitting an item to the Out Box. Send this message to the routing slip view from your `PrepareToSend` method if you want to continue with the process of submitting the item. 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*`)`

Notifies 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 lets you take such necessary action as alerting the user that addressing 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 it returns `nil`, the transport is changed and the routing slip is closed and reopened.

You don't need to supply this method.

protoAddressPicker

This proto provides a picker list to use in the routing slip for choosing an address from the Names file. 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*.

Slot descriptions

`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`, to select none 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 use in creating the list, and determines the information displayed in each column 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 other 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`.

protoTransportPrefs

This proto provides a template for a preferences view for your transport. It is based on the `protoFloater`. For more information about creating a preferences view, see "Providing a Preferences Template" (page 22-33) in *Newton Programmer's Guide*.

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, displayed as part of the title at the top of the preferences view, if you include it.

`appSymbol` — Required. The transport `appSymbol`.

`transport` — This slot is set at run-time with the transport to which this preferences view belongs.

`silentPrefs` A frame defining the text of the checkbox that controls whether or not to show status dialogs. This frame has as its default value the slots described in Table 19-4 (page 19-46). 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 19-5 (page 19-46). 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 19-6 (page 19-47). 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 19-7 (page 19-47). 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 methods that support the Info button, see the description of `protoInfoButton` (page 6-10).

The following four tables describe the default frames for the `silentPrefs`, `sendPrefs`, `outboxPrefs`, and `inboxPrefs` slots. To override any of the default slots in a frame, you must specify a new frame with all the slots shown.

Table 19-4 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 19-5 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 to use for the picker that lists choices. The default array is a localized version of this: `["Send now", "Send later", "Specify when"]`.
`testMethod`	A symbol identifying a sending method for which to test in the transport object. If this method is not found in the transport object, the "When sending" view element is not automatically displayed in the preferences view. The default value is `'SendRequest`. In other words, if the transport does not support sending, the "When sending" view element won't be included.

Table 19-6 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 to use for the picker that lists logging choices. The default array is a localized version of this: `["File", "Log", "Delete"]`.
`testMethod`	A symbol identifying a sending method for which to test in the transport object. If this method is not found in the transport object, the view element controlling Out Box logging is not automatically displayed in the preferences view. The default value is `'SendRequest`. In other words, if the transport does not support sending, this view element won't be included.

Table 19-7 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."
`testMethod`	A symbol identifying a receiving method for which to test in the transport object. If this method is not found in the transport object, the view element controlling In Box logging is not automatically displayed in the preferences view. The default value is `'ReceiveRequest`. In other words, if the transport does not support receiving, this view element won't be included.

Functions and Methods

Utility Functions

This section describes utility functions used in the Transport interface.

RegTransport

`RegTransport(`*symbol*`,` *transport*`)`

Registers a new transport in the system. Call `RegTransport` from the `InstallScript` function in your transport part.

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 a hook that allows the transport to do other initialization when it is installed. Note that the `InstallScript` method of the transport is not related to the `InstallScript` function in the part frame.

UnRegTransport

`UnRegTransport(`*symbol*`)`

Unregisters a transport from the system. Usually you would call this function from the `RemoveScript` function in your transport part.

symbol The transport `appSymbol` passed to `RegTransport`.

The return value of this function is undefined.

DeleteTransport

`DeleteTransport(`*symbol*`)`

Removes 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 in your transport part.

symbol The transport `appSymbol` passed to `RegTransport`.

The return value of this function is undefined.

Note that the `RemoveScript` function in the transport part is also called, following the `DeletionScript` function.

GetCurrentFormat

`GetCurrentFormat(`*item*`)`

Returns the routing format frame (not the format symbol) for an item from the In or 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 can be found in the group, it returns `nil`.

groupSymbol A symbol identifying a transport group. The following group symbols are defined: `'print`, `'fax`, `'beam`, and `'mail`.

QuietSendAll

`QuietSendAll(`*transportSym*`) // platform file function`

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

IMPORTANT

This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

`call kQuietSendAllFunc with (`*transportSym*`);`

▲

transportSym A symbol identifying the transport that is to send queued items.

If the specified transport is not found, this function returns the symbol `'noTransport`. If this function completes normally, the return value is unspecified.

Note that if there are no items to send, the system does not display an alert, as with `CheckOutbox`.

Refresh

ownerApp`:Refresh()`

Causes the application that manages transports (typically the In/Out Box) to refresh the view of the in box.

You must send the `Refresh` message to the `ownerApp` slot in the transport.

Note

In Newton OS version 2.0, the `ownerApp` slot must first be set up by using the NTK platform file function `kSetupOwnerAppFunc` in the transport `InstallScript` method. ◆

You use `Refresh` to refresh the in box view after remote items are fully retrieved and after remote items that are not fully retrieved are deleted. For more information about handling remote items, see "Deferring Reception of the Item Data" (page 22-10) in *Newton Programmer's Guide*.

RemoveTempItems

ownerApp`:RemoveTempItems(`*transportSym*`)`

Causes the application that manages transports (typically the In/Out Box) to delete all remote items from the specified transport that have not been fully downloaded.

transportSym	The transport symbol (from the `appSymbol` slot of the transport).

After this message, you must send the `Refresh` message to the `ownerApp` slot in the transport.

Note
In Newton OS version 2.0, the `ownerApp` slot must first be set up by using the NTK platform file function `kSetupOwnerAppFunc` in the transport `InstallScript` method. ◆

Typically, you use the `RemoveTempItems` method after the transport disconnects, to remove all remote items that the user chose not to retrieve fully. For more information about handling remote items, see "Deferring Reception of the Item Data" (page 22-10) in *Newton Programmer's Guide*.

Endpoint Interface Reference

This chapter provides reference information for all the symbols, constants, data structures, protos, methods, and global functions available for working with the Endpoint interface.

Constants and Symbols

This section describes constants and symbols that your application uses to interact with the Endpoint interface.

Data Form Symbols

The symbols in Table 20-1 specify how data is interpreted and handled by the Endpoint interface.

Table 20-1 Data form symbols

Data form	Description
`'char`	Data is converted to or from Unicode, using the endpoint `encoding` slot.
`'number`	For sending or receiving data, or getting or setting endpoint options, the data is interpreted as a NewtonScript 30-bit integer using 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, only the low-order bytes are used (the data is truncated).
`'binary`	No conversion done.
`'template`	Used to exchange data with a service that expects C-type data, such as when setting communication tool options.
`'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.

Data Type Symbols

The symbols in Table 20-2 specify the data types that can be specified in the `typelist` array when using data of the `'template` form.

Table 20-2 `Typelist` data type symbols

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

Option Opcode Constants

The constants described in Table 20-3 are used in the `opCode` slot of an endpoint option frame to specify how the option is to be set or gotten.

Table 20-3 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

Endpoint Error Code Constants

Endpoint error code constants are described in Table 20-4.

Table 20-4 Endpoint error code constants

Constant	Value	Description
`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
`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

Table 20-4 Endpoint error code constants (continued)

Constant	Value	Description
`kCommScriptInputSpecAlreadyActive`	–54012	Tried to set an input spec when one was already active
`kCommScriptInvalidProxy`	–54013	Invalid value in filter proxy 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 proxy of the input spec is more than a single byte

Option Error Code Constants

Option error code constants are described in Table 20-5.

Table 20-5 Option error code constants

Constant	Value	Description
`kCommOptionFailure`	–54021	Option 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	Invalid option `opcode`
`kCommOptionNotFound`	–54026	Option not found
`kCommOptionTruncated`	–54027	One or more requested options missing

Endpoint State Constants

The constants described in Table 20-6 are the possible return values of the endpoint `State` method.

Table 20-6 Endpoint state constants

Constant	**Value**	**Description**
`kUninit`	0	Uninitialized
`kUnbnd`	1	Unbound
`kIdle`	2	Idle
`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

Other Endpoint Constants

Additional constants used in the Endpoint interface are described in Table 20-7.

Table 20-7 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)

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 24, "Built-in Communications 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. You don't need to specify this slot because the default value `'template` applies to all options.

`result` A result code, set on return from the endpoint method. This slot is ignored if you set it; it is documented here only because it is added to the option frame returned by endpoint methods that can set options. The possible result codes are listed in Table 20-5 (page 20-5).

`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 23-5 in *Newton Programmer's Guide*.

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 you use this slot, specify an integer greater than 30. 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

callbackSpec`:CompletionScript(`*endpoint*`, `*options*`, `*result*`)`

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 23-11 in *Newton Programmer's Guide*.)

`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 23-4 in *Newton Programmer's Guide*.

`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 23-20 in *Newton Programmer's Guide*.

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 23-4 in *Newton Programmer's Guide*.

`target` A frame defining additional information pertaining to `'template` and `'binary` data forms. This frame is described in the section "Input Spec Target Frame" (page 20-15).

`termination` A frame defining input termination conditions. This frame is described in the section "Input Spec Termination Frame" (page 20-16).

`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. If you use this slot, specify an integer greater than 30.

`filter` A frame defining how incoming data is to be processed. This frame is described in the section “Input Spec Filter Frame” (page 20-17).

`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
- `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

inputSpec:`InputScript(`*endpoint*, *data*, *terminator*, *options*`)`

Sent to the input spec frame when one of the data termination conditions has been met, or when the `Input` method is called.

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 of the input spec, *data* would be the `target` frame's data 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. If this argument is `nil`, it indicates that the `InputScript` message was sent as a result of invoking the `Input` method. 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 23-17 in *Newton Programmer’s Guide*.

The return value of the `InputScript` method is ignored by the system.

In the input spec, the `InputScript` slot is evaluated when the `SetInputSpec` message is sent to the endpoint, and every time the `InputScript` message is sent to the input spec.

If the *terminator* argument is `nil`, it indicates that the `InputScript` message was sent as a result of invoking the `Input` method. In this case, the input spec is still active and you cannot set another one by calling `SetInputSpec`. If you want to cancel the current input spec, you must use `Cancel` to do so.

The current input spec (and therefore, the current `InputScript` method) remains in effect after the `InputScript` method returns, unless you call `SetInputSpec` to change the input spec. This feature maximizes performance if the same input spec can be used for each receive.

PartialScript

inputSpec`:PartialScript(`*endpoint*`, `*data*`)`

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.

This method can be used only for data formatted with the input data forms `'string` or `'bytes`.

CompletionScript

inputSpec:`CompletionScript(`*endpoint*`,` *options*`,` *result*`)`

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 currently 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.

`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 an array of one or more 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`.

Note

An input spec filter can be used with all data forms except `'binary`. ◆

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*`)`

Instantiates 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 23-7 in *Newton Programmer's Guide*.

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.

This method is synchronous.

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. After the binding is made or fails, the system calls your callback method.

options An array of one or more option frames.

bindCallback A callback spec frame 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. For details on callback spec frames, see "Callback Spec Frame" (page 20-9).

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. After the unbinding is complete or fails, the system calls your callback method.

unbindCallback A callback spec frame 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. For details on callback spec frames, see "Callback Spec Frame" (page 20-9).

You must disconnect the endpoint before sending this message to unbind it.

Dispose

endpoint:`Dispose()`

Closes the endpoint and deallocates the underlying endpoint structures. This method is synchronous.

You must disconnect and unbind the endpoint before sending this message to dispose it.

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. After the connection is made or fails, the system calls your callback method.

options — An array of one or more option frames.

connectCallback — A callback spec frame 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. For details on callback spec frames, see "Callback Spec Frame" (page 20-9).

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` after a connection has been established (either by `Connect` or `Accept`).

Listen

endpoint:`Listen(`*options*, *listenCallback*`)`

Listens for a connection request from the remote system.

After the connection request is received, you must call the `Accept` or `Disconnect` method to accept or reject the connection. When used synchronously, this method waits for the connection request to be received before returning. When used asynchronously, this method posts the listen request and then returns. After the connection request is received or this method fails, the system calls your callback method.

options — An array of one or more option frames.

listenCallback — A callback spec frame 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. For details on callback spec frames, see “Callback Spec Frame” (page 20-9).

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 request was received by the `Listen` method.

When used synchronously, this method waits for the connection to be established before returning. When used asynchronously, this method posts a request to establish a connection and then returns. After the connection is established or this fails, the system calls your callback method.

options — An array of one or more option frames.

acceptCallback A callback spec frame 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. For details on callback spec frames, see “Callback Spec Frame” (page 20-9).

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` after you have accepted the connection.

Disconnect

endpoint`:Disconnect(`*cancelPending*`, `*disconnectCallback*`)`

Disconnects a connection.

When used synchronously, this method waits for the connection to be disconnected before returning. When used asynchronously, this method posts a request to disconnect a connection and then returns. After the connection is disconnected or this fails, the system calls your callback method.

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 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 details on callback spec frames, see "Callback Spec Frame" (page 20-9).

For more discussion on canceling, see the section "Canceling Operations" beginning on page 23-21 in *Newton Programmer's Guide*.

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 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. For details on output spec frames, see "Output Spec Frame" (page 20-10).

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 "Input Spec Frame" (page 20-11).

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.

▲ **WARNING**
The `Cancel` method aborts all pending asynchronous operations on the endpoint. Use it with caution. ▲

Input

endpoint:`Input()`

Causes the `InputScript` message to be sent to the current input spec. All data in the input buffer is formatted and passed to the `InputScript` method, and the input buffer is cleared. Note that the input spec is not terminated.

You use this method only when receiving data of the forms `'string` and `'bytes`.

An input spec must be active at the time this method is called, or the method throws an exception with the error `kCommScriptNoActiveInputSpec`.

IMPORTANT

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

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 to clear the input buffer.

You use this method only when receiving data of the forms `'string` and `'bytes`.

An input spec must be active at the time this method is called, or the method throws an exception with the error `kCommScriptNoActiveInputSpec`.

IMPORTANT

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

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).

Cancel

endpoint`:Cancel(`*cancelCallback*`)`

Cancels all pending requests, synchronous or asynchronous.

cancelCallback A callback spec frame 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. For details on callback spec frames, see "Callback Spec Frame" (page 20-9).

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.

If there is nothing to cancel, the `Cancel` method has no effect.

IMPORTANT
For more information on canceling asynchronous operations, see the section "Canceling Operations" beginning on page 23-21 in *Newton Programmer's Guide*. ▲

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 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. For details on callback spec frames, see "Callback Spec Frame" (page 20-9).

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.

It is possible to 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. This is generally considered inappropriate and should be avoided, if possible. For more information on option processing, see the section "Setting Endpoint Options" beginning on page 23-8 in *Newton Programmer's Guide*.

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 20-6 (page 20-6).

Note

The endpoint `State` method returns information about the state of the NewtonScript endpoint, but does not necessarily indicate the true state of the communication tool(s) in use by the endpoint. Do not rely on the value returned by the `State` method to determine a course of action for the endpoint; it is intended for debugging only. ◆

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

streamingEndpoint`:StreamIn(`*streamSpec*`)`

Posts a receive request to the communication tool. As data arrives, it is unflattened into a frame and collected in memory.

This synchronous 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. If you use this slot, specify an integer greater than 30.

`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 23-15 in *Newton Programmer's Guide*.

`target` — Optional. If you are receiving a frame containing embedded virtual binary objects, this slot specifies on which store to place the objects. This slot must contain a frame with a single slot, `store`. The `store` slot must contain a reference to the store on which virtual binary objects are to be created.

The `ProgressScript` method (page 20-32) can also be defined in the *streamSpec* frame.

Virtual binary objects embedded in a received frame are not copied into the NewtonScript heap along with the rest of the frame, if you specify a store to receive them in the `target` slot of the *streamSpec* frame. Use this technique to avoid overflowing the NewtonScript heap when receiving such objects.

Note
The `StreamIn` method cannot receive version 1 flattened frames. ◆

StreamOut

streamingEndpoint:`StreamOut(`*data*`,` *streamSpec*`)`

Takes a frame and sends it in chunks while flattening it.

This synchronous 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. If you use this slot, specify an integer greater than 30.
`sendFlags`	Optional. This slot can contain protocol flags provided for certain communication tools. For more details, see the section "Sending Data" beginning on page 23-11 in *Newton Programmer's Guide*.

The `ProgressScript` method, described next, can also be defined in the *streamSpec* frame.

Note
The `StreamOut` method sends data using version 2 (or later) of the flattened frame format. ◆

ProgressScript

streamSpec:`ProgressScript(`*bytes*`,` *totalBytes*`)`

Informs your application of `StreamOut` or `StreamIn` progress.

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.

Depending on the size of the data being sent or received, and the amount of time used to perform the operation, the `ProgressScript` message may never be sent if the operation completes before it can be triggered.

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 of the form `"`*name*`:`*type*@*zone*`"`.

MakeModemOption

`MakeModemOption()`

Returns an option frame of the `'template` 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*`)`

Translates data using the specified translation engine. This function returns 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 20-8 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. Specify `nil` if this parameter does not apply to a particular translation (for example flattened frame to frame).

progressScript A function object that may be called periodically during the translation process to inform your application of progress. This function is passed two parameters: the first is the number of bytes that have been translated thus 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 function 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 feature is not implemented by either of the two existing translators, so this parameter is currently ignored.

Table 20-8 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.

Built-in Communications Tools Reference

This chapter provides reference information for the constants, options, methods, and functions that you use with the Newton built-in communications tools.

When you use a built-in communications tool, you need to include a service option in your options array. The service option must specify the service identifier for the tool that you are using. For example, to use the built-in serial tool with MNP, you include an option like the following:

```
myOptions := [
   { label:   kCMSMNPID,
     type:    'service,
     opCode:  opSetRequired } ];
```

Table 21-1 shows the service option label for each built-in communications tool.

Table 21-1 Built-in communications tool service option labels

Service option label	Value	Built-in communication tool
`kCMSAsyncSerial`	`"aser"`	asynchronous serial
`kCMSMNPID`	`"mnps"`	serial with MNP
`kCMSModemID`	`"mods"`	modem
`kCMSSlowIR`	`"slir"`	infrared
`kCMSFramedAsyncSerial`	`"fser"`	framed, asynchronous, serial
`kCMSAppleTalkID`	`"atlk"`	AppleTalk

Use of the built-in communications tools is described in "Built-in Communications Tools" (page 24-1) in *Newton Programmer's Guide*.

Options for the Standard Asynchronous Serial Tool

This section describes the options you can use to configure the serial communication tool. Table 21-2 summarizes the standard serial options.

Table 21-2 Summary of serial options

Label	Value	Use When	Description
`kCMOSerialHWChipLoc`	"schp"	Before or at binding	Sets which serial hardware to use.
`kCMOSerialChipSpec`	"sers"	Before or at binding	Sets which serial hardware to use and returns information about the serial hardware.
`kCMOSerialCircuitControl`	"sctl"	After connecting	Controls usage of the serial interface lines.
`kCMOSerialBuffers`	"sbuf"	Before or at binding	Sets the size of the input and output buffers.
`kCMOSerialIOParms`	"siop"	Any time	Sets the bps rate, stop bits, data bits, and parity options.
`kCMOSerialBitRate`	"sbps "	Any time	Changes the bps rate.
`kCMOOutputFlowControlParms`	"oflc"	Any time	Sets output flow control parameters.
`kCMOInputFlowControlParms`	"iflc"	Any time	Sets input flow control parameters.
`kCMOSerialBreak`	"sbrk"	After connecting	Sends a break.
`kCMOSerialDiscard`	"sdsc"	After connecting	Discards data in input and/or output buffer.
`kCMOSerialEventEnables`	"sevt"	Any time	Configures the serial tool to complete an endpoint event on particular state changes.

Table 21-2 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

The serial chip location option, with label `kCMOSerialHWChipLoc`, specifies which serial hardware in the system to use for the endpoint. This option must be set during or after binding of the endpoint; however, you can use this option at any time to retrieve the serial chip location information.

The following example shows the use 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 21-3. Note that these locations are hardware platform dependent.

Table 21-3 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.

The external serial port is typically the default setting; however, this can vary since the default is established by the communications tool.

The service ID field within the `data` slot specifies a four-character string identifying a communications tool. If the location label slot is `nil`, the default serial chip location for the specified communications 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

The serial chip specification option, with label `kCMOSerialChipSpec`, 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 an 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, you must use it before or in your call to the endpoint `Bind` method. You can, however, use this option at any time to retrieve serial chip information.

Note

The serial chip specification option is considered an "expert" option. It is only useful in special circumstances. ◆

The serial chip specification option also returns information about the serial hardware. You can use this option to retrieve information at any time.

The following example shows the use of this option 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        // 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
      ]
   }
};
```

Table 21-4 shows the fields in the serial chip specification option.

Table 21-4 Serial chip specification option fields

Option Field	Description
fHWLoc	Specifies the serial chip location. The default value is 0.
fSerFeatures	Features supported by this chip. The default value is 0.
fSerOutSupported	Output signals supported by this chip. The default value is 0.
fSerInSupported	Input signals supported by this chip. The default value is 0.
fParitySupport	Parity supported by this chip. See Table 21-5 for the constants you can specify. The default value is 0.
fDataStopBitSupport	Number of data and stop bits supported by this chip. See Table 21-5 for the constants you can specify. The default value is 0.
fUARTType	Type of serial chip. See Table 21-5 for the constants you can specify. The default value is 0.
fChipNotInUse	A Boolean specifying whether or not the chip is in use (The default value is `true`, which means the chip is not in use).
fCIS_ManFID	Application card Card Information Structure (CIS) manufacturer ID. The default value is 0.
fCIS_ManFIDInfo	Application card CIS manufacturer ID information. The default value is 0.

The constants you can use to specify various field values in the serial chip specifications option are listed in Table 21-5.

Table 21-5 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
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 Universal Asynchronous Receiver/ Transmitter (UART)
kSerialChip16450	0x01	16450 UART

Table 21-5 Serial chip specification option constants (continued)

Constant	Value	Description
`kSerialChip16550`	0x02	16550 UART
`kSerialChip8530`	0x20	8530 UART (SCC chip)
	0x21	Reserved for future use
	0x22	Reserved for future use
	0x23	Reserved for future use
`kSerialChipUnknown`	0x00	Unknown type of UART

Serial Circuit Control Option

The serial circuit control option, with label `kCMOSerialCircuitControl`, controls usage of the serial control lines. You can only use this option after the endpoint is connected. When you do set it, this option returns the current state.

Note
The serial circuit control option is considered an "expert" option. It is only useful in special circumstances. ◆

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. ▲

The following example shows the use 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: [
         'struct,
         'byte,         // fSerOutToSet
         'byte,         // fSerOutToClear
         'byte,         // fSerOutState
         'byte          // fSerInState
      ]
   }
};
```

The fields in the serial circuit control option frame are described in Table 21-6.

Table 21-6 Serial circuit control option fields

Option field	Description
`fSerOutToSet`	Output lines to assert. Combine the values from Table 21-7 for each output line you want to assert. The default value is 0.
`fSerOutToClear`	Output lines to negate. Combine the values from Table 21-7 for each output line you want to negate. The default value 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 21-7.

Table 21-7 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	A "break" condition

When the `kSerInBreak` bit is on, 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" with the serial event configuration option, which is described in “Serial Event Configuration Option” (page 21-21). 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 serial send break option, which is described in "Serial Send Break Option" (page 21-20).

Serial Buffer Size Option

Th serial buffer size option, with label `kCMOSerialBuffers`, lets you increase the size of the buffers used by the serial tool. Buffers larger than 4KB are not supported; an error results if you specify too large a buffer. Also note you can get an out-of-memory error at connect time if the serial tool cannot allocate the buffers.

This option is often useful because appropriate buffer size can increase performance and decrease overrun errors. For communications that use packet-oriented protocols, a good buffer size is one that is a few bytes larger than the typical packet size.

For streamed communications, output buffer size is not as important as input buffer size 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 received 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 between one and four bytes less than the buffer size, because of DMA boundary constraints and other considerations.

You can only set the serial buffer size option before or at connect time. You can, however, use this option at any time to retrieve the current buffer settings.

The following example shows the use of the serial buffer size 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,     // error characters to remember
      ]
   }
};
```

The default output buffer size is 512 bytes and the default input buffer size is 512 bytes.

Serial Configuration Option

The serial configuration option, with label `kCMOSerialIOParms`, specifies the bps rate, stop bits, data bits, and parity options for the serial tool. You typically set this option at endpoint `open` or `connect` time. You can also use this option to change the data format after a connection has been established; however, this might require resetting some serial chips, which can cause problems with serial inputs.

This option returns the actual values that were set. You can compare those values with what you requested to see if you actually got the connection configuration that you requested.

The following example 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
`k2400bps`	2400
`k4800bps`	4800
`k7200bps`	7200

Constant	Value
`k9600bps`	9600 (default)
`k12000bps`	12000
`k14400bps`	14400
`k19200bps`	19200
`k38400bps`	38400
`k57600bps`	57600
`k115200bps`	115200
`k230400bps`	230400

Serial Data Rate Option

The serial data rate option, with label `kCMOSerialBitRate`, is used for changing the bps rate after a connection has already been made. If the data bit, stop bit, and parity values don't have to be changed, the serial date rate option is a reliable way of changing the data rate.

You can use this option at any time. It returns the actual bit rate that was set. You can compare this value with what you requested to see if you actually got the speed that you requested.

The following example shows the use of this option:

```
local option := {
   type:    'option,
   label:   kCMOSerialBitRate,
   opCode:  opSetRequired,
   form:    'number,
   result:  nil,          // not needed; returned
   data:    k19200bps,    // change to 19200
};
```

You can specify the same values in the `data` slot as for the bps field in the serial configuration option, given on (page 21-16). The default value 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 input serial flow control option has the label `kCMOInputFlowControlParms.` The output serial flow control option has the label `kCMOOutputFlowControlParms`. Both of these options can be used at any time.

The following example shows the use of 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 21-8.

Table 21-8 Serial flow control option fields

Option field	Description
XON character	Specifies the XON character to use for software flow control (the default value is DC1, `0x11`).
XOFF character	Specifies the XOFF character to use for software flow control (the default value 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

You use the serial send break option, with label `kCMOSerialBreak`, to send a break (string of start bits) for the amount of time specified. No synchronization is done with output.

Use this option after the endpoint is connected. Note that you can only set this option; you cannot read the current setting, since the option performs an action rather than setting some kind of parameter.

The following example 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 100 ms break
};
```

Specify the length of time for the break (in the `data` slot) in milliseconds by specifying an integer multiplied by the constant `kMilliseconds`. The default value is 75 milliseconds.

Serial Discard Data Option

You use the serial discard data option, with label `kCMOSerialDiscard`, to discard data in the input or output buffers. Discarding is useful after error conditions, or before synchronization is achieved in serial communications.

Use this option after the endpoint is connected. Note that you can only set this option; you cannot read the current setting, since the option performs an action rather than setting some kind of parameter.

With modem endpoints, this option works only when Microcom Networking Protocol (MNP) is not being used.

The following example 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

You use the serial event configuration option, with label `kCMOSerialEventEnables`, to configure the serial tool to send an `EventHandler` message to your endpoint when particular state changes occur. You can use this option at any time.

The following example 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. Combine the constants, which are shown in Table 21-9, to specify more than one event. The default value is zero, for no events.

For the `kSerialEventDCDNegatedMask` event, you need to 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 value is zero.

Table 21-9 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.

The serial tool passes two parameters when it sends an `EventHandler` message to your endpoint. The first is the integer value `1`. The second parameter is an integer value that indicates which event occurred, using the same mask bits as shown in Table 21-9. Some higher-order bits may be set as well, so don't count on them being zero.

You must provide an `EventHandler` method in your endpoint to receive the message from the serial tool. See "EventHandler" (page 20-28) for more information about this method.

Serial Bytes Available Option

The serial bytes available option, with label `kCMOSerialBytesAvailable`, is a read-only option that returns the number of bytes waiting to be read from the receive buffer.

You can only use this option after the endpoint is connected.

The following example shows the use of this option:

```
local option := {
   type:    'option,
   label:   kCMOSerialBytesAvailable,
   opCode:  opGetCurrent,
   form:    'number,
   result:  nil,     // not needed; returned
   data:    0,       // returned
};
```

Serial Statistics Option

The serial statistics option, with label `kCMOSerialIOStats`, is a read-only option that returns various software and hardware statistics related to the serial tool.

You can only use this option after the endpoint is connected.

The following example 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 the serial statistics option frame are described in Table 21-10..

Table 21-10 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]

Table 21-10 Serial statistics option fields (continued)

Option field	Description
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 occur when receiving data. Hard overruns occur when the serial driver doesn't unload the data from the hardware before it is overwritten by subsequent data. Soft overruns occur 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

The serial external clock divide option, with label `kHMOSerExtClockDivide`, controls how the clock rate is divided when using an external clock. This option is not supported by all serial chips.

You can use this option after binding your endpoint. Any changes that you make with this option take effect at endpoint connect time. If you are already connected and you set the clock rate with this option, you must follow the setting of this option with the setting of the `kCMOSerialIOParms` (page 21-14) and `kCMOSerialBitRate` (page 21-17) options.

Note
The serial external clock divide option is considered an "expert" option. ◆

The following example 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,
};
```

You can use the following constants in the `data` slot:

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

Options for the Serial Tool with MNP Compression

This section describes the options you can use to configure the serial communication tool with MNP compression (the MNP tool). The MNP tool uses all of the standard serial tool options and two additional MNP options. One of the MNP options, the data compression type option, is described in "MNP Compression Option" (page 21-61). The other is described in this section.

Table 21-11 summarizes the MNP serial options.

Table 21-11 Summary of serial tool with MNP options

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

Serial MNP Data Rate Option

The serial MNP data rate option, with label `kCMOMNPDataRate`, is used by the MNP tool to configure its internal timers. This option is required because the serial port speed may be different than the end-to-end speed.

When using a serial MNP endpoint, you must set this option to the correct value for MNP to function correctly. You must set this option at or before calling your endpoint's `Connect` method.

The following example 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.

Options for the Framed Asynchronous Serial Tool

The framed asynchronous serial tool is a superset of the standard asynchronous serial tool that supports framed data. The framing is controlled by the two additional options used by this tool.

Table 21-12 summarizes the additional framed asynchronous serial tool options.

Table 21-12 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.

Serial Framing Configuration Option

The serial framing configuration option, with label `kCMOFramingParms`, configures data framing parameters. This option applies only to the framed asynchronous serial tool.

The following example 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 21-13.

Table 21-13 Serial framing configuration option fields

Option field	Description
escape character	Specifies the character to use for escape. The default value is DLE (`0x10`).
EOM character	Specifies the character to use for end of message. The default value 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 24-1 (page 24-5) in *Newton Programmer's Guide*.

Serial Framing Statistics Option

The serial framing statistics option, with label `kCMOFramedAsyncStats`, is a read-only option that 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.

The following example 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; returned (# of bytes)
};
```

Options for the Modem Tool

This section describes the options that you can use with the built-in modem tool. Table 21-14 summarizes the modem tool options.

Table 21-14 Summary of modem options

Label	Value	Use When	Description
`kCMOModemPrefs`	"mpre"	Any time	Configures the modem controller.
`kCMOModemProfile`	"mpro"	Any time	Override modem setup selected in preferences. Use when instatiating.
`kCMOModemECType`	"mecp"	Any time	Specifies the type of error control protocol to be used in the modem connection.
`kCMOModemDialing`	"mdo"	Any time	Controls the parameters associated with dialing.
`kCMOModemConnectType`	"mcto"	Any time	Configures the modem endpoint for the type of connection desired (voice, fax, data, or cellular data).
`kCMOModemConnectSpeed`	"mspd"	After connecting	Read-only option indicating modem to modem raw connection speed.
`kCMOModemFaxCapabilities`	"mfax"	After bind, before connecting	Read-only option indicating the fax service class capabilities and modem modulation capabilities.
`kCMOModemFaxEnabledCaps`	"mfec"	Any time	Determines or sets currently enabled fax service and modem modulation capabilities.
`kCMOModemVoiceSupport`	"mvso"	After bind, before connecting	Read-only option indicating if the modem supports line current sense (LCS).

Table 21-14 Summary of modem options (continued)

Label	Value	Use When	Description
`kCMOMNPSpeedNegotiation`	"mnpn"	Any time	Sets MNP data rate speed.
`kCMOMNPCompression`	"mnpc"	Before connecting	Sets the data compression type.
`kCMOMNPStatistics`	"mnps"	After connecting	Read-only option reporting performance statistics from the current MNP connection.

Modem Address Option

You use the modem address option, with label `kCMARouteLabel`, to specify the address to use during the connection phase. You can specify different routing types in this option, one of which is `kPhoneNumber`. You use the `kPhoneNumber` constant with the modem tool to specify that you are providing a phone number.

The following example shows the use of this option:

```
local option := {
      label:  kCMARouteLabel,
      type:   'address,
      opCode: opSetRequired,
      data: {
         arglist: [
            kPhoneNumber,        // type
            size,                // phone string length
            phoneStr,            // the phone number
         ],
         typelist: [
            'struct,
            'long,
            'uLong,
```

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

Alternatively, you can call the global function `MakeModemOption` to construct an address option. This function is described in "MakeModemOption" (page 20-33).

Modem Preferences Option

You use the modem preferences option, with label `kCMOModemPrefs`, to configure the modem controller. You can enable or disable certain features of the controller with this option, which must be set before you call your endpoint's `Bind` method.

The following example shows the use 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 the modem preferences option frame are described in Table 21-15.

Table 21-15 Modem preferences option fields

Option Field	Description
`fConnectInDirectMode`	If `true`, forces the 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 the modem profile indicates the modem can support speed buffering. The default value is `nil`.
`fIdModem`	If `true`, the modem tool executes the ID sequence in an attempt to identify which modem 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. The default value 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. The default value 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. The default value is `true`.
`fUseSoftwareCD`	Ignored.
`fUseConfigString`	If `true`, before initiating a connection, the modem tool sends the current configuration string to the modem (as determined by active modem profile and the connection type). If `nil`, no configuration string is sent. The default value is `true`.

Table 21-15 Modem preferences option fields (continued)

Option Field	Description
`fUseDialOptions`	If `true`, the modem tool sets the modem dialing configuration according to current option settings. This is done before initiating a connection and after the configuration string is sent to modem. If `nil`, the dial configuration string is not sent to modem. The default dial configuration string is `ATM1L2X4S7=060S8=001S6=003\n`. The default value is `true`.
`fHangUpAtDisconnect`	If `true`, the modem tool hangs up the modem using the hang-up sequence when the modem disconnects. If `nil`, when the modem disconnects, the modem tool does not send any commands to the modem. The default value is `true`.
`fEnablePassThru`	If `true`, the 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`, the modem tool controller is enabled for normal modem tool operation. The default value is `nil`.
`fEnableDialOutStream`	If `true`, enables dialing of the output stream. After connecting, all data output by modem tool client endpoint is sent to the modem as dial commands. This feature can be used for interactive dialing. If `nil`, the modem handles client endpoint output as normal data. The default value is `nil`.

Table 21-15 Modem preferences option fields (continued)

Option Field	Description
`fDirectModeSpeed`	The speed in bits per second (bps) at which modem tool begins direct mode connection. The default value is 19200 bps.
`fHWCDDelayLowSpeed`	The amount of time, in seconds, that the CD line must be deasserted before considering the line disconnected. This value is used for connection speeds less than 2400 bps. The default value is 3 seconds.
`fHWCDDelayHighSpeed`	The amount of time, in seconds, that the CD line must be deasserted before considering the line disconnected. This value is used for connection speeds greater than 2400 bps. The default value is 15 seconds.

Modem Profile Option

You use the modem profile option, with label `kCMOModemProfile`, to define the characteristics of a modem. The profile is used by the modem controller to configure and connect the modem.

The modem profile option specifies a number of modem characteristics, including the following:

- whether the modem supports asynchronous speed buffering, in which case CTS flow control must be supported
- whether the modem supports special cellular configuration (e.g., signal attenuation)
- error control types supported
- possible connection speeds
- the highest speed supported for the DTE-DCE interface
- the maximum command processing time
- the configuration strings used for various types of connections

Applications do not usually need to set this option. The modem profile is usually established by the modem setup, which enables a particular modem to work with all applications that use a modem endpoint. Users can choose the appropriate modem setup in the modem preferences. See Chapter 25, “Modem Setup Service,” for a description of how to write a modem setup package.

If your application needs to customize the active modem profile, you can set this option in the configuration options for your endpoint. You should disable the modem ID feature in your modem preferences option by setting the `fIdModem` slot in that option to nil.

The following example shows the use of this option:

```
local Stuffer := func(src, dst)
begin
   local count := StrLen(src);
   local index := Length(dst);
   SetLength(dst, index + count + 1);
   for i:=0 to count-1 do
      dst[index + i] := Ord(src[i]);
   dst[index + count] := 0;
   count + 1;
end;
local data := [];
local size :=
// modem id string
call Stuffer with ("pagemodem", data) +
// config string no EC
call Stuffer with ("ATE0&C1S12=12S11=60S23=5\n", data) +
// config string EC only
call Stuffer with ("AT\n", data) +
// config string EC and fallback
call Stuffer with ("AT\n", data) +
// config string EC cellular
call Stuffer with ("AT\n", data) +
// config string direct connect
call Stuffer with ("ATE0&C1S12=12S11=60S23=5\n", data);
```

```
local option := {
   type: 'option,
   label:kCMOModemProfile,
   opCode:opSetRequired,
   form: 'template,// not needed
   data : {
      arglist: [
         nil,        // supports Cellular
         nil,        // supports EC
         nil,        // supports LCS
         true,       // direct connect only
         255,        // connect speeds
         1200,       // config speed
         2000,       // command response timeout
         40,         // max characters per command line
         25,         // inter-command delay
         size,       // modem strings length
         data,       // the strings, packed&byte-aligned
      ],
      typelist: [
         'struct,
         'boolean,   // fSupportsCellular
         'boolean,   // fSupportsEC
         'boolean,   // fSupportsLCS
         'boolean,   // fDirectConnectOnly
         'ulong,     // fConnectSpeeds
         'ulong,     // fConfigSpeed
         'ulong,     // fCommandResponseTimeOut
         'ulong,     // fMaxCharsPerCmdLine
         'ulong,     // fInterCmdDelay
         'ulong,     // fModemStringsLen
         ['array, 'byte, size], // fModemStrings
```

```
      ]
   }
};
```

The fields in the modem profile option frame are described in Table 21-16.

Table 21-16 Modem profile option fields

Option field	Description
fSupportsCellular	If true, indicates that the 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. The default value is nil.
fSupportsEC	If true, indicates that the modem supports built-in error correction, and the profile contains configuration strings for error correction. The default value is nil.
fSupportsLCS	If true, indicates that the modem supports line current sense (LCS). 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). The default value is nil.
fDirectConnectOnly	If true, indicates that the modem only supports direct connect mode and can’t support speed buffering. In this case, the DTE speed must be adjusted to the modem speed after the carrier is established. If nil, indicates that the modem supports speed buffering and the use of CTS flow control. The default value is true.

Table 21-16 Modem profile option fields (continued)

Option field	Description
`fConnectSpeeds`	Indicates speeds (in bps) at which the modem can connect. This value does not affect the modem configuration. The intention is for the application to read this value to determine the modem capabilities. The default value is 255, which represents these speeds: 300, 1200, 2400, 4800, 7200, 9600, 12000, and 14400. Here are the bit flags, which are combined 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
`fConfigSpeed`	Indicates the speed at which to configure the modem, in bps. The default value 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. The default value is 2000.

Table 21-16 Modem profile option fields (continued)

Option field	Description
`fMaxCharsPerCmdLine`	Indicates the 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. The default value is 40.
`fInterCmdDelay`	Indicates the minimum amount of delay required between modem commands, in milliseconds. This is the time from last response received to next command sent. The default value is 25.
`fModemStringsLen`	Indicates the length of the modem strings in the `fModemStrings` field (packed together byte-aligned and null terminated). This value includes the termination characters.
`fModemStrings`	An array of bytes that contains the modem configuration strings shown in Table 21-17. You can create this array with the `Stuffer` function shown in the example.

Table 21-17 shows the modem configuration strings in the `fModemStrings` field of the modem profile option.

Table 21-17 Modem profile configuration strings

Configuration string	Explanation
Modem id string	Modem response to the `ATI4` command. If the modem responds with more than one result string, you can specify only one string. The default value is `unknown`.

Table 21-17 Modem profile configuration strings (continued)

Configuration string	Explanation
Config string no EC	Modem command string used to configure the modem for a non-error-corrected connection. Uses speed buffering. This string is used for FAX connections. The default value is `"ATE0&C1S12=12W2&K3&Q6\n"`.
Config string EC only	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. The default value is `nil`.
Config string EC and fallback	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. The default value is `nil`.
Config string EC cellular	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. The default value is `nil`.
Config string direct connect	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. The default value is `"ATE0&C1S12=12W2&K0&Q0\n"`

Modem Error Control Type Option

The modem error control type option, with label `kCMOModemECType`, specifies the type of error control protocol to use in the modem connection. If you specify more than one type of error control protocol, the modem tool uses precedence to determine which type of error control protocol to use for the connection.

The following pseudo-code shows how the modem tool determines which error control protocol to use when you specify more than one:

```
if (External EC is enabled) then
   begin
      if (No EC is enabled) then
         use fConfigStrECAndFallback
      else
         use fConfigStrECOnly
   end
else usefConfigStrNoEC;

// attempt MNP connection
if (MNP connection fails)
   begin
      if (No EC is enabled)
         fallback to normal connection
      else
         disconnect
   end
else
   connected with MNP
```

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, `fConfigStrCellular` is used even if external error control is enabled separately. The modem dialing option is described in “Modem Dialing Option” (page 21-47). ◆

You must set the modem error control option at instantiation time. The following example 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 21-18. Note that these values can be combined together to specify multiple error control types. The default is `kModemECProtocolMNP+kModemECProtocolNone.`

Table 21-18 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`	0x00000010	Connect with internal error control only; overrides other settings.

What happens when a reliable connection cannot be established depends on which error control types you specify. If you include `kModemECProtocolNone` in your specification, then the modem tool "falls back" to a connection with no error control. If you do not include `kModemECProtocolNone` in your specification and a reliable connection cannot be established, the connection fails.

If you specify `kModemECInternalOnly`, the Newton's internal error control is used. This setting takes precedence over the other error control types.

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 `kModemECProtocolMNP` or `kModemECProtocolExternal` specified, then which string is used depends on whether you have `kModemECProtocolNone` specified: if you do, the fallback string is used; if you don't, the error-control-only string is used.

Modem Dialing Option

You use the modem dialing option, with label `kCMOModemDialing`, to control the parameters associated with dialing. You must set this option in your call to the endpoint `Connect` method or after the endpoint is connected.

Rather than setting the modem dialing option manually, you should use the global function `MakeModemOption`, which is described in "MakeModemOption" (page 20-33). This method reads the user preferences and builds the modem dialing option frame for you.

The following example shows the use of this option:

```
local option := {
   type:    'option,
   label:   kCMOModemDialing,
   opCode:  opSetRequired,
   form:    'template,// not needed
   data : {
      arglist: [
         true,       // speaker on
```

```
         true,        // detect dial tone
         true,        // detect busy
         true,        // dtmf tone dialing
         nil,         // manual dial
         2,           // speaker volume
         2,           // wait for carrier in seconds
         2,           // wait for blind dial in seconds
         2,           // comma delay in seconds
         2,           // ring to answer after 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 21-19.

Table 21-19 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`). The default value is `true`.
`fDetectDialTone`	If `true`, the modem detects and requires a 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`). The default value 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`). The default value is `true`.
`fDTMFToneDialing`	If `true`, the modem uses DTMF dialing (`ATDT`…). If `nil`, the modem uses pulse dialing (`ATDP`…). The default value 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. The default value is `nil`.
`fSpeakerVolume`	Modem speaker level. This value is used in the `ATLn` command. Use one of the following: `kSpeakerVolumeLow` "1" `kSpeakerVolumeMedium` "2" `kSpeakerVolumeHigh` "3" Note that these are one-character strings. The default value is `kSpeakerVolumeMedium`.

Table 21-19 Modem dialing option fields (continued)

Option Field	Description
fWaitForCarrier	The amount of time, in seconds, that the modem waits to establish carrier before going off-hook. This value sets modem register S7. The default value is 55.
fWaitBeforeBlindDial	The amount of time, in seconds, that the modem waits after going off-hook until dialing when dial tone is not required (when fDetectDialTone is nil). This value sets modem register S6. The default value is 3.
fCommaDelay	The amount of time, in seconds, that the modem pauses in dialing when a comma is encountered in the dial string. This value sets modem register S8. The default value is 1.
fRingToAnswerAfter	The number of rings after which to answer when waiting for an incoming call. This value sets modem register S0. The default value is 2.
fCountryId	The current location of the user, derived from the Time Zones setting. The following values are defined, based on the country codes: kUSACountryId 1 kCanadaCountryId 10 kJapanCountryId 81 The default value is kUSACountryId.
fCellularConnection	Indicates that the fConfigStrCellular string from the Modem Profile Option should be used.

▲ **WARNING**

Some modem setups are written exclusively for cellular modems, which means that you must only set the value of `fCellularConnection` to `true` if you are also specifying your own modem profile that includes an `fCellularConnection` string. ▲

Modem Connection Type Option

You use the modem connection type option, with label `kCMOModemConnectType`, to configure 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; Class One commands are then 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).

If more than one type of connection is enabled, the modem tool initiates the connection type with the highest precedence. The connection precedence order is voice (highest), fax, and 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.

You must set this option before or in the call to your endpoint's `Connect` method.

The following example shows the use of the modem connection type 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 the modem connection type option frame are described in Table 21-20.

Table 21-20 Modem connection type option fields

Option Field	Description
`fVoiceEnable`	If `true`, enables voice connection (auto-dial with modem). The default value is `nil`.
`fFaxEnable`	If `true`, enables fax connection. The default value is `nil`.
`fDataEnable`	If `true`, enables data connection. The default value is `true`.
`fImmediate`	If `true`, go off-hook immediately after configuring the modem. The dialing step (or when listening, the waiting for ring step) is skipped. The default value is `nil`.

Modem Connection Speed Option

You use the modem connect speed option, with label `kCMOModemConnectSpeed`, to retrieve the 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. You can only use this option when the endpoint is in the connected state.

The following example shows the use of this option:

```
local option := {
   type:    'option,
   label:   kCMOModemConnectSpeed,
   opCode:  opGetCurrent,
   form:    'number,
   data :   0,
};
```

Modem Fax Capabilities Option

You use the modem fax capabilities option, with label `kCMOModemFaxCapabilities`, to determine the fax service class capabilities and modem modulation capabilities. You can only use this option after the endpoint `Bind` call. This is a read-only option that returns the values for the modem.

The following example shows the use of the modem fax capabilities 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 21-21.

Table 21-21 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 / Class 1 fax kModemFaxClass2 / 0x00000004 / Class 2 fax kModemFaxClass2_0 / 0x00000008 / Class 2.0 fax
fTransmitDataMod	Indicates transmit modulations supported by the AT+FTM=x command. See Table 21-22 (page 21-56) for possible return values. The array of possible values needs to be combined together.
fTransmitHDLCDataMod	Indicates transmit HDLC modulations supported by the AT+FTH=x command. See Table 21-22 for possible return values. The array of possible values needs to be combined together.
fReceiveDataMod	Indicates receive modulations supported by the AT+FRM=x command. See Table 21-22 for possible return values. The array of possible values needs to be combined together.
fReceiveHDLCDataMod	Indicates receive HDLC modulations supported by the AT+FRM=x command. See Table 21-22 for possible return values. The array of possible values needs to be combined together.

The fax modulation return values are shown in Table 21-22.

Table 21-22 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 Fax Enabled Capabilities Option

You use the modem fax enabled capabilities option, with label `kCMOModemFaxEnabledCaps`, to determine or set which fax service class capabilities and modem modulation capabilities are enabled. You can use this option at any time.

Note
This option is available only with System Software version 2.1 or later. ◆

The slots in the modem fax enabled capabilities option are the same as the slots in the modem fax capabilities option described in the previous section (page 21-53). You use this option to constrain which hardware capabilities you want used for your application or to determine which capabilities are currently enabled.

You don't normally 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. For information about writing modem setup packages, see "Defining a Modem Setup" (page 25-5) in *Newton Programmer's Guide*.

The following example shows the use of the modem fax capabilities option:

```
local option := {
   type:    'option,
   label:   kCMOModemFaxEnabledCaps,
   opCode:  opSetCurrent,
   form:    'template,// not needed
   data : {
      arglist: [
         nil,
         0,
         kModemFaxClass0 + kModemFaxClass1
            + kModemFaxClass2 + kModemFaxClass2_0,
         kV17st_14Mod + kV17_14Mod + kV17st_12Mod
            + kV17_12Mod + kV17st_96Mod + kV17_96Mod
            + kV17st_72Mod + kV17_72Mod + kV29_96Mod
            + kV29_72Mod + kV27Ter48Mod + kV27Ter24Mod,
         kV21Ch2Mod,
         kV17st_14Mod + kV17_14Mod + kV17st_12Mod
            + kV17_12Mod + kV17st_96Mod + kV17_96Mod
            + kV17st_72Mod + kV17_72Mod + kV29_96Mod
            + kV29_72Mod + kV27Ter48Mod + kV27Ter24Mod,
         kV21Ch2Mod,
```

```
      ],
      typelist: [
         'struct,
         'ulong, // fServiceId
         'ulong, // fExtendedResult
         'ulong, // fServiceClass
         'ulong, // fTransmitDataMod
         'ulong, // fTransmitHDLCDataMod
         'ulong, // fReceiveDataMod
         'ulong, // fReceiveHDLCDataMod
      ]
   }
};
```

Modem Voice Support Option

You use the modem voice support option, with label `kCMOModemVoiceSupport`, 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 that 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.

The following example shows the use 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

You use the MNP speed negotiation option, with label `kCMOMNPSpeedNegotiation`, to control 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 serial MNP data rate option, which is described in "Serial MNP Data Rate Option" (page 21-28).

Note
You can only use this option if you are using the Newton's built-in MNP software, which means that you must be using the `kModemECInternalOnly` error control type. ◆

You typically use the serial configuration option (page 21-14) to establish the connection parameters and then use the serial data rate option (page 21-17) to change speeds during later negotiations. If you use both options in the same call, 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 serial configuration or serial data rate options). When you use this MNP speed negotiation option, it negotiates the MNP data rate speed, and the serial port speed is set to this value. ◆

The following example shows the use of the MNP speed negotiation 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

You use the MNP compression option, with label `kCMOMNPCompression`, is 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 scheme can be used, the connection can be made with no compression. This option must be set at or before the endpoint `Connect` call.

Note

You can only use this option if you are using the Newton's built-in MNP software, which means that you must be using the `kModemECInternalOnly` error control type. ◆

The following example 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 21-23. Note that these values can be combined to specify multiple compression types. The default value is all three values combined together.

Table 21-23 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

You use the MNP data statistics option, with label `kCMOMNPStatistics`, to retrieve performance statistics from the current MNP connection. This is a read-only option. You can use this option after your endpoint is connected.

Note
You can only use this option if you are using the Newton's built-in MNP software, which means that you must be using the `kModemECInternalOnly` error control type. ◆

The following example 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 the MNP data statistics option frame are described in Table 21-24.

Table 21-24 MNP data statistics option fields

Option field	Description
fAdaptValue	Maximum size data packet when the connection supports adaptive packet sizing (Class 4). The default value is 196.
fLTRetransCount	Number of times current data packet (LT) has been retransmitted. The default value is 0.
fLRRetransCount	Retransmission count for connect packet (LR - link request). The default value is 0.
fRetransTotal	Total number of LT frame retransmissions during connection. The default value is 0.
fRcvBrokenTotal	Total number of broken frames received during connection. The default value is 0.
fForceAckTotal	Total number of forced acknowledgments during connection. The default value is 0.
fRcvAsyncErrTotal	Total number of serial driver async errors (overruns) received during connection. The default value is 0.
fFramesRcvd	Total number of frames received during connection. The default value is 0.
fFramesXmited	Total number of frames transmitted during connection. The default value is 0.
fBytesRcvd	Total number of data bytes received during connection. Includes packet header/tail. The default value is 0.

Table 21-24 MNP data statistics option fields (continued)

Option field	Description
`fBytesXmited`	Total number of data bytes transmitted during connection. Includes packet header/tail. The default value is 0.
`fWriteBytesIn`	Total number of user data bytes transmitted during connection (before compression). The default value is 0.
`fWriteBytesOut`	Total number of user data bytes transmitted during connection (after compression). The default value is 0.
`fReadBytesIn`	Total number of user data bytes received during connection (before decompression). The default value is 0.
`fReadBytesOut`	Total number of user data bytes received during connection (after decompression). The default value is 0.
`fWriteFlushCount`	Number of flush calls to V.42bis compressor during connection. The default value is 0.

Options for the Infrared Tool

This section describes the options that you can use with the infrared (IR) tool. Table 21-25 summarizes the infrared tool options.

Table 21-25 Summary of infrared options

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

Infrared Connection Option

The infrared connection option, with label `kCMOSlowIRConnect`, controls how the infrared connection is made. You can set it in the `Instantiate`, `Bind`, or `Connect` methods.

The following example 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.

You 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, with label `kCMOSlowIRPRotocolType`, is a read-only option that reports the protocol and speed of the current infrared connection. You can use this option after the endpoint is connected or accepted.

The following example shows the use 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, Apple recommends setting `sendFlags` to `kPacket+kEOP` every time you send data. 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 `sendFlags` see "Output Spec Frame" (page 20-10).

For more information on `useEOP` and `byteCount`, see "Input Spec Termination Frame" (page 20-16).

Infrared Statistics Option

The infrared statistics option, with label `kCMOSlowIRStats`, is a read-only option that reports various statistics on the current infrared connection. You can use this option after the endpoint is connected or accepted.

The following example 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 21-26.

Table 21-26 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.

Table 21-26 Infrared statistics option fields (continued)

Option field	Description
`falseStarts`	Not used.
`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.

Options for the AppleTalk Tool

This section describes the options that you can use with the AppleTalk tool. Table 21-27 summarizes the AppleTalk tool options.

Table 21-27 Summary of AppleTalk options

Label	Value	Use When	Description
`kCMARouteLabel`	“rout”	When connecting or listening	Sets an AppleTalk NBP address.
`kCMOAppleTalkBuffer`	“bsiz”	When connecting, listening, or accepting	Sets the size of the send, receive, and attention buffers.

Table 21-27 Summary of AppleTalk options (continued)

Label	Value	Use When	Description
`kCMOSerialBytesAvailable`	"sbav"	After connecting	Read-only option returns the number of bytes available in the receive buffer.
`kCMSAppleTalkID`	"atlk"	For instantiation	Specifies AppleTalk tool type.
`kCMOEndpointName`	"endp"	For instantiation	Specifies AppleTalk endpoint. Must be used as above.

AppleTalk Address Option

The AppleTalk address option, with label `kCMARouteLabel`, specifies the AppleTalk NBP address. You must specify this option when connecting or listening.

The following example 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],
      ]
   }
};
```

You need to set the size and the NBP string. The value of size is the length in bytes of the NBP string that you are passing.

You must pass the `kCMARouteLabel` option to the ADSP's `Connect` and `Listen` methods. For `Connect`, the option value specifies to whom you are connecting. For Listen, the option value specifies who you are.

Alternatively, you can construct an address option by calling the `MakeAppletalkOption` function, which is described in “MakeAppleTalkOption” (page 20-33).

AppleTalk Buffer Size Option

The AppleTalk buffer size option, with label `kCMOAppleTalkBuffer`, specifies the sizes of the send, receive, and attention buffers. You must specify a separate option for each buffer type. You can set this option in conjunction with the `Connect`, `Listen`, or `Accept` methods.

The buffer types are identified by integers, as follows:

Buffer type	Identifier	Default size
Send	`kSndbuffer`	511
Receive	`kRevBuffer`	511
Attention	`kAtnBuffer`	0

The following example shows the use of this option:

```
local opt := {
   label:   kCMOAppleTalkBuffer,
   type:    'option,
   opCode:  opSetRequired,
   data: {
      arglist: [
         kBufferType,   // kSndbuffer, kRevBuffer,
                        //  or kAtnBuffer
         kSize,         // buffer size in bytes
      ],
      typelist: [
         'struct,
         'ulong,
         'long,
      ]
   }
};
```

AppleTalk Bytes Available Option

The AppleTalk bytes available option, with label `kCMOSerialBytesAvailable`, is a read-only option that the number of bytes waiting to be read from the receive buffer. You can use this option after the endpoint is connected.

The following example 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 Tool Type Option

The AppleTalk tool type option, with label `kCMSAppleTalkID`, specifies which AppleTalk tool to use. At the current time you must use the ADSP tool. You must specify this option at instantiation time and you must specify exactly as shown in the following example:

```
local option := {
   type:    'option,
   label:   kCMSAppleTalkID,
   type:    'option,
   opCode:  opSetRequired,
   form:    'template,
   data:     {
      arglist: ["adsp"],      // or KCMOAppleTalkADSP
               typelist:[
                  'struct
                  ['array, 'char, 4]
                  ]
   },
};
```

▲ **WARNING**

You must specify the AppleTalk tool type option in your `Instantiate` call for the AppleTalk tool. You must specify this option exactly as shown above for your connection to work. ▲

AppleTalk Endpoint Name Option

The AppleTalk endpoint name option, with label `kCMOEndpointName`, specifies which built-in endpoint the AppleTalk tool is to use. You must specify this option at instantiation time and you must specify exactly as shown in the following example:

```
local option := {
   type:    'option,
   label:   kCMOEndpointName,
   opCode:  opSetRequired,
   form:    'template,
   data:    {
      arglist:  [kADSPEndpoint],
      typelist: [
               'struct
               ['array, 'char, 0]
               ]
      }
};
```

▲ **WARNING**
You must specify the AppleTalk endpoint name option in your `Instantiate` call for the AppleTalk tool. You must specify this option exactly as shown above for your connection to work. ▲

AppleTalk Functions

This section describes the global functions you can use to obtain the addresses of other devices on an AppleTalk network.

AppleTalk Driver Functions

This section describes the functions you can use to open and close the AppleTalk drivers. For more information about these drivers, see "AppleTalk Functions" (page 24-12) in *Newton Programmer's Guide*.

OpenAppleTalk

`OpenAppleTalk()`

Opens the AppleTalk drivers and returns zero if successful. You can call `OpenAppleTalk` as many times are you like. Be sure to call `CloseAppleTalk` at least as many times as you call `OpenAppleTalk`.

If AppleTalk is already open, the `OpenAppleTalk` function increments a counter and returns `0`.

You do not need to call `OpenAppleTalk` to access zone information. The zone information function open and close AppleTalk if necessary.

CloseAppleTalk

`CloseAppleTalk()`

Closes the AppleTalk drivers and returns 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.

You do not need to call `CloseAppleTalk` to access zone information. The zone information function open and close AppleTalk if necessary.

AppleTalkOpenCount

`AppleTalkOpenCount()`

Returns the open count for the AppleTalk drivers. A return value of `0` means the drivers are closed.

Functions for Obtaining AppleTalk Zone Information

This section describes the global functions you can use to obtain zone information.

HaveZones

```
HaveZones()
```

Returns `true` if a connection exists and zones are available. Returns `nil` if there are no zones available.

If the AppleTalk drivers are not opened, this function automatically opens and then closes them.

GetMyZone

```
GetMyZone()
```

Returns a string naming the current AppleTalk zone. A return value of `"*"` identifies the default zone, which usually means that no AppleTalk router was found.

If the AppleTalk drivers are not opened, this function automatically opens and then closes them.

GetZoneList

```
GetZoneList()
```

Returns an array containing strings of all the existing zone names, or returns `nil` if no zones are available, which usually means that no AppleTalk router was found.

If the AppleTalk drivers are not opened, this function automatically opens and then closes them.

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.

The following example shows the use of this function:

```
#4415501  "Idiot Savante:LaserWriter@RD1/
NewHaven-LocalTalk"

GetNames(GetUserConfig('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`.

The following example shows the use of this function:

```
GetZoneFromName(
            GetUserConfig('currentPrinter).printerName)
#44184A9  "RD1/NewHaven-LocalTalk"
```

NBPStart

`NBPStart(`*entity*`)`

Begins a lookup of network entities, as specified by the *entity* parameter.

If the AppleTalk drivers are not opened, this function automatically opens and then closes them.

The `NBPStart` function returns a lookup ID that is used with the other NBP functions described in this section. This function returns `nil` if the lookup cannot 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 the following code:

```
lookupID := NBPStart("=:LaserWriter@*");
NBPGetNames(lookupID);
```

NBPGetCount

`NBPGetCount(`*lookupID*`)`

Returns the number of entities found by the currently running NBP lookup.

lookupID The lookup ID returned by the `NBPStart` function used to start this lookup.

The following example shows the use of 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 `0` 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 Methods

This section describes the methods you can use with the Net Chooser. For more information on using the Net Chooser, see "The Net Chooser" (page 24-13) in *Newton Programmer's Guide*.

OpenNetChooser

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

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.

The following example shows the use of this method:

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

NetworkChooserDone

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

To obtain the user's selection, you need to provide a method called `NetworkChooserDone`. This method must have the above format.

currentSelection Returns the selected entity

currentZone Returns the currently selected AppleTalk zone.

Options for Resource Arbitration

This section describes the resource arbitration options. For more information about using these options, see "Resource Arbitration Options" (page 24-10)

in *Newton Programmer's Guide*. Table 21-28 summarizes the resource arbitration options.

Table 21-28 Summary of 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

Passive Claim Option

The resource-passive claim option, with label `kCMOPassiveClaim`, specifies whether or not a communications tool is claiming its resources passively.

The following example shows the use of this option:

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

If the `data` value is `true`, the communications tool is claiming its resources passively. If the `data` value is `nil`, the tool is claiming its resources actively, which means that another application cannot claim those resources.

Passive State Option

The resource-passive state option, with label `kCMOPassiveClaim`, specifies whether or not your communications tool is currently in a state in which it can release resources.

The following example shows the use of this option:

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

If the `data` value is `true`, the communications tool is willing to relinquish use of its passively claimed resources. If the `data` value is `nil`, the communications tool is not willing to relinquish use of its passively claimed resources.

Modem Setup Service Reference

This chapter describes the constants that you use with the modem setup service. For more information about the modem setup service, see “Modem Setup Service” (page 25-1) in *Newton Programmer’s Guide*.

Modem Setup General Information Constants

The following constants specify general information about the modem setup.

Table 22-1 Constants for modem setup general information

Constant	Description
`kModemName`	The string name that identifies this modem. This string is shown as the modem name in the Modem Preferences picker..
`kVersion`	The integer version number of this modem setup package. The Newton system software 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 Preference Constants

The following constants specify the modem setup preferences for configuring the modem controller. For more information about the modem preference option, see “Modem Preferences Option” (page 21-34)

Table 22-2 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.

Table 22-2 Constants for modem setup preferences (continued)

Constant	Description
`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 22-3 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`. 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 applications that want to determine the modem capabilities.
`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.

Table 22-3 Constants for the modem setup profile (continued)

Constant	Description
`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.
`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 shown in "The No Error Control Configuration String" (page 22-7).
`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`. See "The Error Control Configuration String" (page 22-8) for an example of an error control configuration string.
`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`. See "The Error Control with Fallback Configuration String" (page 22-9) for an example of a configuration string for error control with fallback.
`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 shown in "The Direct Connect Configuration String" (page 22-9).

The No Error Control Configuration String

The following is the default value for the `kConfigStrNoEC` configuration string:

`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.

The Error Control Configuration String

The following is an example of a `kConfigStrECOnly` configuration string:

`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.

The Error Control with Fallback Configuration String

The following is an example of a `kConfigStrECAndFallback` configuration string:

`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.

The Direct Connect Configuration String

The following is the default value for the `kConfigStrDirectConnect` configuration string:

`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 Constants

The following constants specify the fax setup preferences for configuring the modem controller.

Table 22-4 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. The available speeds are shown in Table 22-5 (page 22-11).
`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. The available speeds are shown in Table 22-5 (page 22-11).

Table 22-4 Constants for the fax profile (continued)

Constant	Description
`kServiceClass`	Specifies which fax protocols are supported. The available service classes are shown in Table 22-6 (page 22-12). You can only set the service class (use the `kServiceClass` constant) for versions of the software that support the Class 2 fax protocol. Newton System Software version 2.1 and the German version of Newton System Software version 2.0 support the Class 2 fax protocol.

The speeds at which faxes are sent and received are specified by a bit table. The individual on bits in the value indicate the available fax speeds. For example:

```
kV21Ch2Mod + KV27Ter24Mod + kV27Ter48Mod.
```

Table 22-5 lists the strings available for these two constants.

Table 22-5 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

Table 22-5 Available fax speeds (continued)

Configuration string	Value	Bits per second
`kV17_12Mod`	0x00000200	12000
`kV17st_12Mod`	0x00000400	12000
`kV17st_14Mod`	0x00001000	14400

Table 22-6 lists the fax service classes.

Table 22-6 Available fax service classes

Configuration string	Value	Fax protocol
`kModemFaxClass0`	0x00000001	no fax service
`kModemFaxClass1`	0x00000002	Class 1 fax
`kModemFaxClass2`	0x00000004	Class 2 fax
`kModemFaxClass2_0`	0x00000008	Class 2.0 fax

Utility Functions Reference

This chapter describes a number of utility functions. The following groups of functions are included here:

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

Object System Functions

The functions described in this section operate on NewtonScript objects. They perform operations such as getting and checking for slots, removing slots, cloning frames, and so forth.

ClassOf

`ClassOf(`*object*`)`

Returns the class of an object.

object The object whose class to return.

The return value is a symbol. Some common object classes are `'int`, `'char`, `'boolean`, `'string`, `'array`, `'frame`, `'function`, and `'symbol`. Note that this is not necessarily the same as the primitive class of an object. For binary, array, and frame objects, the class can be set differently from the primitive class.

Frames or arrays without an explicitly assigned class are of the primitive class `'frame` or `'array`, respectively. If a frame has a class slot, the value of the class slot is returned. Here are some examples:

```
f:={multiply:func(x,y) x*y};
classof(f);
#1294    Frame

f:={multiply:func(x,y) x*y, class:'Arithmetic};
classof(f);
#1294     Arithmetic

s:="India Joze";
classof(s);
#1237    String
```

See also `PrimClassOf`.

Clone

`Clone(`*object*`)`

Makes and returns a “shallow” copy of an object; that is, references within the object are copied, but the data pointed to by the references is not.

object The object to copy.

Here is an example:

```
SeaFrame :=  {Ocean: "Pacific", Size: "large" , Color: "blue"};
seaFrameCopy := clone(seaFrame);
seaFrameCopy.Deep := true;
seaFrame
#441896D  {Ocean: "Pacific", size: "large", Color: "blue"}
seaFrameCopy
#4418B0D  {Ocean: "Pacific", size: "large", Color: "blue",
   Deep: TRUE}
```

See Table 26-1 (page 26-2) in *Newton Programmer’s Guide* for a comparison with other object copying functions.

DeepClone

`DeepClone(`*object*`)`

Makes and returns a “deep” copy of an object; that is, all of the data referenced within the object is copied, including that referenced by magic pointers (pointers to ROM objects).

object The object to copy.

It is not guaranteed that every part of the data structure is in RAM. (Certain information, such as the symbols naming frame slots, may be shared with the original object.)

Contrast this function with `Clone`, which only makes a “shallow” copy, and the functions `TotalClone` and `EnsureInternal`, which ensure that the object exists entirely in internal RAM. See Table 26-1 (page 26-2) in *Newton Programmer’s Guide* for a comparison with other object copying functions.

EnsureInternal

`EnsureInternal(`*obj*`)`

Ensures that the object exists entirely in internal RAM or ROM. This function may copy all, some, or none of the object to ensure that it exists in RAM. Note that magic pointers are not followed; that is, objects referenced through magic pointers are not copied.

obj The object to ensure exists in internal RAM.

This function returns an object, which may or may not be a copy of the original object.

See Table 26-1 (page 26-2) in *Newton Programmer's Guide* for a comparison with other object copying functions.

GetFunctionArgCount

`GetFunctionArgCount(`*function*`)`

Returns the number of arguments expected by a function.

function The function whose number of arguments to get.

GetSlot

`GetSlot(`*frame*`,` *slotSymbol*`)`

Returns the value of a slot in a frame. Only the frame specified is searched.

frame A reference to the frame in which to look for the slot.

slotSymbol A symbol naming the slot whose value to get.

If the slot doesn't exist, this function returns `nil`.

Unlike `GetVariable`, `GetSlot` searches for a slot only in the indicated frame. Inheritance is not used to find the slot.

The use of the NewtonScript dot operator is similar to the `GetSlot` function in that it also returns the value of a frame slot. For example, the expression `frame.slot` returns the value of the specified slot. However, when using the dot operator, if the slot is not found in the specified frame, proto frames are also searched for the slot (but not parent frames).

GetVariable

`GetVariable(`*frame*`,` *slotSymbol*`)`

Returns the value of a slot in a frame. If the slot is not found, returns `nil`.

frame	A reference to the frame in which to begin the search for the slot.
slotSymbol	A symbol naming the slot whose value to get.

This function begins its search for the slot in the specified frame and makes use of the full proto and parent inheritance.

HasSlot

`HasSlot(`*frame*`,` *slotSymbol*`)`

Returns non-`nil` if the slot exists in the frame, otherwise returns `nil`. Inheritance is not used to find the slot.

frame	The name of the frame in which to look for the slot.
slotSymbol	A symbol naming the slot whose existence to check.

HasVariable

`HasVariable(`*frame*`,` *slotSymbol*`)`

Returns non-`nil` if the slot exists in the frame, otherwise returns `nil`. This function searches proto and parent frames of the specified frame if the slot is not found there.

frame	The name of the frame in which to begin the search for the slot.
slotSymbol	A symbol naming the slot whose existence to check. You must use a single quote before the slot name because it is a symbol.

Intern

`Intern(` *string* `)`

May or may not create and return a symbol whose name is given as the string parameter *string*. If a symbol with that name already exists, the preexisting symbol is returned.

string — The name of the symbol.

IsArray

`IsArray(`*obj*`)`

Returns non-`nil` if *obj* is an array.

obj — The object to test.

IsBinary

`IsBinary(`*obj*`)`

Returns non-`nil` if *obj* is a binary object.

obj — The object to test.

IsCharacter

`IsCharacter(`*obj*`)`

Returns non-`nil` if *obj* is a character, and returns `nil` otherwise.

obj — The object to test.

IsFrame

`IsFrame(`*obj*`)`

Returns non-`nil` if *obj* is a frame.

obj — The object to test.

IsFunction

`IsFunction(`*obj*`)`

Returns non-`nil` if *obj* is a function, and returns `nil` otherwise.

obj	The object to test.

IsImmediate

`IsImmediate(`*obj*`)`

Returns non-`nil` if *obj* is an immediate.

obj	The object to test.

IsInstance

`IsInstance(`*obj*`,` *class*`)`

Returns non-`nil` if *obj*'s class symbol is the same as *class* or a subclass of *class*.

obj	The object to test.
class	A symbol specifying the class.

Note that this is equivalent to

`IsSubclass(ClassOf(`*obj*`),` *class*`)`

IsInteger

`IsInteger(`*obj*`)`

Returns non-`nil` if *obj* is an integer, and returns `nil` otherwise.

obj	The object to test.

IsPathexpr

`IsPathexpr(`*obj*`)`

Returns non-`nil` if *obj* is a valid expression, and returns `nil` otherwise.

obj	The object to test.

IsNumber

`IsNumber(`*obj*`)`

Returns non-`nil` if *obj* is a number (integer or real), and returns `nil` otherwise.

obj The object to test.

IsReadOnly

`IsReadOnly(`*obj*`)`

Returns non-`nil` if *obj* is read-only, and returns `nil` otherwise. You can use `IsReadOnly` to determine if an array, frame, or binary object is writable.

obj An array, frame, or binary object to test. (Immediate objects such as integers are never read-only.)

Here is an example:

```
if IsReadOnly(viewBounds) then
   viewBounds := Clone(viewBounds);
```

This function should not be used to determine the location of an object; that is, whether it is in the heap, in ROM, or in protected memory. The NewtonScript language permits read-only objects in the NewtonScript heap, or writable objects that exist in other locations.

IsReal

`IsReal(`*obj*`)`

Returns non-`nil` if *obj* is a real number, and returns `nil` otherwise.

obj The object to test.

IsString

`IsString(`*obj*`)`

Returns non-`nil` if *obj* is a string, and returns `nil` otherwise.

obj The object to test.

IsSubclass

`IsSubclass(`*sub, super*`)`

Checks if a class is a subclass of another class.

sub	A class symbol to test.
super	A class symbol.

This function returns non-`nil` if *sub* is a subclass of *super,* or is the same as *super.* Returns `nil` if *sub* is not a subclass of *super.* See also the related function `IsInstance` (page 23-7).

IsSymbol

`IsSymbol(`*obj*`)`

Returns non-`nil` if *obj* is a symbol, and returns `nil` otherwise.

obj	The object to test.

MakeBinary

`MakeBinary(`*length*`, `*class*`)`

Allocates a new binary object of the specified *length* and *class*.

length	The size of the binary object in bytes.
class	A symbol specifying the class.

Map

`Map(`*obj*`, `*function*`)`

Applies a function to the slot name and value of each element of an array or frame.

obj	An array or frame.
function	Returns `nil`. A function to apply to the elements or slots in *obj*. The function is passed two parameters: *slot* and *value*. The *slot* parameter contains an integer array

index if *obj* is an array, or a symbol naming a slot if *obj* is a frame. The *value* parameter contains the value of the array or frame slot referenced by the *slot* parameter.

This is equivalent to

```
for each slot,value in obj do call function with
(slot,value)
```

PrimClassOf

`PrimClassOf(`*obj*`)`

Returns the primitive class of an object.

obj The object whose primitive class to return.

Returns a symbol identifying the primitive data structure type of the object, either: `'immediate`, `'binary`, `'array`, or `'frame`.

See also `ClassOf`.

RemoveSlot

`RemoveSlot(`*obj*`,` *slot*`)`

Removes a slot from a frame or array.

obj The name of the frame or array from which to remove the slot.

slot A symbol naming the frame slot you want to remove, or the index of the array slot to remove. Note that no inheritance lookup is used to find this slot in *obj*.

This function returns the modified frame or array. If *slot* is not found, nothing is done and the unmodified frame or array is returned. Note that the system throws an exception if *obj* is read-only.

ReplaceObject

`ReplaceObject(`*originalObject*`,` *targetObject*`)`

Causes all references to an object to be redirected to another object.

originalObject The original object.

targetObject The object to which you want to redirect references to *originalObject*.

This function always returns `nil`.

Note that you cannot specify immediate objects as parameters to this function.

Here is an example:

```
x:={name:"Star"};
y:={name:"Moon"};
replaceobject(x,y);
x;
#469E69   {name: "Moon"}

y;
#46A1E9   {name: "Moon"}
```

SetClass

`SetClass(`*obj*`,` *classSymbol*`)`

Sets the class of an object.

obj The object whose class to set.

classSymbol A symbol naming the class to give to the object.

This function returns the object whose class was set.

You can set the class of the following kinds of objects: frames, arrays, and binary objects. Note that you cannot set the class of an immediate object.

When setting the class of a frame, if a `class` slot doesn't exist, one is created in the frame. For example:

```
x:={name: "Star"};
setclass(x, 'someClass);
#46ACC9   {name: "Star",
           class: someClass}
```

SetVariable

`SetVariable(`*frame*`, `*slotSymbol*`, `*value*`)`

Sets the value of a slot in a frame. The value is returned.

frame	A reference to the frame in which to begin the search for the slot.
slotSymbol	A symbol naming the slot whose value to set. If the slot is not found, it is created in *frame*.
value	The new value of the slot.

This function begins its search for the slot in the specified frame and makes use of the full proto and parent inheritance.

If the slot is found in the proto chain, it is not set there, but is created and set in *frame*, or in its parent chain, following the usual inheritance rules as they apply to setting a value.

SymbolCompareLex

`SymbolCompareLex(`*symbol1*`, `*symbol2*`)`

Compares symbols lexically. This function returns a negative number if symbol *symbol1* is less than symbol *symbol2*. Returns zero if the two symbols are equal. Returns a positive number if *symbol1* is greater than *symbol2*. Case is not significant (that is, `'Hello` and `'hello` are equal).

symbol1	A symbol.
symbol2	A symbol.

TotalClone

`TotalClone(`*obj*`)`

Makes and returns a “deep” copy of an object; that is, all of the data referenced within the object is copied.

obj The object to copy.

This function is similar to `DeepClone`, except that this function guarantees that the object returned exists entirely in internal RAM. Also, unlike `DeepClone`, `TotalClone` does not follow magic pointers, so that objects referenced through magic pointers are not copied. See Table 26-1 (page 26-2) in *Newton Programmer’s Guide* for a comparison with other object copying functions.

String Functions

These functions operate on and manipulate strings.

BeginsWith

`BeginsWith(` *string*`,` *substr* `)`

Returns non-`nil` if *string* begins with *substr,* or returns `nil` otherwise. This function is case and diacritical-mark insensitive. An empty *substr* matches any *string*.

string The string to test.

substr A string.

Capitalize

`Capitalize(` *string* `)`

Capitalizes the first character in *string* and returns the result. This function modifies *string*.

string The string to modify.

CapitalizeWords

`CapitalizeWords(` *string* `)`

Capitalizes the first character of each word in *string* and returns the result. This function modifies *string*.

string The string to modify.

CharPos

`CharPos(`*str*`,` *char*`,` *startpos*`)`

Returns the position of the next occurrence of character in the specified string, starting from the *startPos* (or `nil` if it's not found).

str The specified string.

char The specified character in the string.

startpos The starting position of the character to return.

Downcase

`Downcase(` *string* `)`

Changes each character in *string* to lowercase and returns the result. This function modifies *string*.

string A string or character (when used to interpret code).

EndsWith

`EndsWith(` *string*`,` *substr* `)`

Returns non-`nil` if *string* ends with *substr,* or returns `nil` otherwise. This function is case and diacritical-mark insensitive. An empty *substr* matches any *string*.

string The string to test.

substr A string.

EvalStringer

`EvalStringer(` *frame*`,` *array* `)`

Returns a string containing all of the elements in *array* concatenated. Any symbols in *array* are evaluated in the context of the specified *frame*.

frame	A frame used as the context for evaluating symbols in *array*.
array	An array.

Numbers, strings, characters, and symbols are converted to their natural string representation. For elements that are frames, arrays, and Booleans, this function converts them to an empty string.

FindStringInArray

`FindStringInArray(` *array*`,` *string* `)`

Finds a string in an array. This function compares the string to each element of the array. If the string matches the value of an array element, the index of that array element is returned. If the string is not found in the array, `nil` is returned.

array	An array to test.
string	A string.

The string comparison used to find a match is case sensitive. The string in the array must exactly match the string you specify in order for it to be found; a partial word will not be found.

FindStringInFrame

`FindStringInFrame(` *frame*`,` *stringArray*`,` *path* `)`

Finds one or more strings, specified by *stringArray,* in a frame.

frame	A frame to test.
stringArray	An array containing strings.
path	A Boolean indicating whether or not to return a description of the locations of successful searches.

This function compares the strings to each slot of the frame that contains a string. If all of the strings you specify in *stringArray* are found somewhere in the frame, this function returns non-`nil`. This function recursively searches arrays and frames referenced within the target frame for the strings. If all of the specified strings are not found within the target frame, including other frames and arrays referenced in it, `nil` is returned.

The string comparison used to find a match is not case sensitive (unlike `FindStringInArray`). Also, the search looks for word beginnings, so it will not find a string unless it begins a word. For example, in the string "blackboard", this function would find the strings "blackboard" or "black", but not "board".

If *path* is non-`nil`, and the strings are found in the frame, this function returns an array of entries describing where each occurrence of the strings was found in the frame. A group of three entries is added to the array for each occurrence of a found string.

The first entry in each group is the complete value of the slot where the string was found.

The second entry is the path to the slot where the string was found (array elements are indicated by their index). This second entry can be either a slot access expression; that is, `aSlot.anotherSlot.lastSlot`, or a path expression array; that is `[pathExpr: aSlot, 3, lastSlot]` if the path includes an array.

The third entry is the offset (in characters) of the string within the slot where it was found.

Here is an example:

```
myframe:={type: 'person,
             data: {name: "Christine Morrison",
                  employer: {company: "Apple",
                               years: 4,
                               boss: "John Morris"}}}
findstringinframe(myframe, ["Morris"], true)
```

```
#52185B1  ["Christine Morrison",
            data.name,
            10,
            "John Morris",
            data.employer.boss,
            5]
```

FormattedNumberStr

`FormattedNumberStr(`*number*`,` *formatString*`)`

Returns a formatted string representation of a real number.

number	A real number.
formatString	A string specifying how the number should be formatted.

This function works similar to (but not exactly like) the C function `sprintf`. The *formatString* parameter specifies how the real number should be formatted; that is, whether to use decimal or exponential notation, and how many places to include after the decimal point. You can specify the following *formatString* values:

%f	Use decimal notation (such as "123.456000").
%e	Use exponential notation (such as "1.234560e+02").
%E	Use exponential notation (such as "1.234560E+02").

You can also specify a period followed by a number, after the % symbol (for example,"%.2f") to indicate how many places to show following the decimal point.

Note

`FormattedNumberStr` uses the current values of `GetLocale().numberFormat` to get the separator and decimal characters and settings. The example strings shown above are for the US English locale. ◆

IsAlphaNumeric

`IsAlphaNumeric(`*char*`)`

Returns non-`nil` if *char* is a number or a letter; otherwise, returns `nil`.

char A character to test.

IsWhiteSpace

`IsWhiteSpace(`*char*`)`

Returns non-`nil` if *char* is a space ($\20), tab ($\09), linefeed ($\0A), or carriage return ($\0D) character; otherwise, returns `nil`.

char A character.

LatitudeToString

`LatitudeToString(`*latitude*`)`

Returns a string representation of the encoded latitude value.

latitude The latitude value.

LongitudeToString

`LongitudeToString(`*longitude*`)`

Returns a string representation of the encoded longitude value.

longitude The longitude value.

NumberStr

`NumberStr(` *number* `)`

Returns a string representation of the number passed in.

number An integer or real number to convert.

For example, if you pass in the value 1234.56, you get back: `"1234.56"`. If you pass in an integer, the string will contain an integer, and if you pass in a real number, the string will contain a real number.

ParamStr

`ParamStr(` *baseString*`,` *paramStrArray* `)`

Returns a new string that is the result after the substitution has been performed. The original *baseString* is not modified.

baseString	The base string containing substitution placeholders.
paramStrArray	An array of strings to substitute for the placeholders in the base string. You can also specify numbers, characters, or symbol data types to convert them to their natural string representation.

This function returns the base string after the substitutions have been made.

The substitution placeholders in the base string are the following character pairs: `"^0"`, `"^1"`, and so on up to `"^9"`. There can be a maximum of 10 placeholders specified in any order in the base string. However, no numbers can be skipped; that is, if the string contains ^2, it must also contain ^1 and ^0. The substitution is done by replacing placeholder ^0 with the first element from the string array. Then placeholder ^1 is replaced by the second element, and so on.

Placeholders can be nested up to three levels deep. This means that the substitution strings can themselves contain placeholders, which are replaced on subsequent passes up to two additional times after the initial replacement.

If you need to specify a caret (^) as part of a string, use two carets together (^^).

`ParamStr` also supports conditional substitution using this syntax:

`^?`*Xtrue*`|`*false*`|`

The value *X* is an integer from 0 through 9, representing a standard placeholder, as above. If the element in *paramStrArray* corresponding to this placeholder is non-`nil` and not the empty string, the *true* characters are interpreted. Otherwise, the *true* characters are skipped, and the *false* characters are interpreted. The vertical bars act to delimit the *true* and *false* portions of the string. Note that the *true* or *false* portions of the string may contain no characters.

Conditional operators can be nested, and any character can appear between the delimiters. If you need to use the vertical bar character as part of a *true* or *false* string, specify `^|`.

The conditional operator is useful for avoiding the insertion of unnecessary punctuation or spaces when building a string from elements that may include optional or potentially empty items.

Here are some examples. If your *baseString* is:

```
"^2 ^0 of each ^1."
```

and your *paramStrArray* is

```
["Monday", "week", "Every"]
```

then `ParamStr` returns this string:

```
"Every Monday of each week."
```

If your *baseString* is

```
"^?0^0, ||^?1^1, ||^2"  // false branches are empty
```

and your *paramStrArray* is:

```
["Sarah", "", "Smith"]
```

then `ParamStr` returns this string

```
"Sarah, Smith"
```

SPrintObject

`SPrintObject(` *obj* `)`

Returns a string of the object passed in. Numbers, strings, characters, and symbols are converted to their natural string representation. For frames, arrays, and Booleans, this function returns an empty string.

To convert the contents of a frame or array into strings, use the `Foreach` statement along with the `Stringer` function to iterate over each slot.

To convert a Boolean into a string, you must check for non-`nil` or `nil` and return the appropriate string.

Note
This function changes the number format depending on the current locale setting. Real numbers may be formatted unexpectedly. ◆

StrCompare

`StrCompare(` *a*, *b* `)`

Returns a negative number if string *a* is less than string *b*. Returns zero if string *a* and *b* are equal. Returns a positive number if string *a* is greater than string *b*. Case is not significant (that is, "Hello" and "hello" are equal).

a	A string.
b	A string.

Note that this is a content comparison of the two strings, not a pointer comparison.

Use `StrExactCompare` to do a case-sensitive comparison of strings.

StrConcat

`StrConcat(` *a*, *b* `)`

Concatenates string *b* onto string *a* and returns the result as a new string.

a	A string.
b	A string.

StrEqual

`StrEqual(` *a*, *b* `)`

Returns non-`nil` if the two strings, *a* and *b*, are equal.

a	A string.
b	A string.

Case is not significant. Note that this is a content comparison of the two strings, not a pointer comparison.

Use `StrExactCompare` to do a case-sensitive comparison of strings.

StrExactCompare

`StrExactCompare(` *a*`,` *b* `)`

Returns a negative number if string *a* is less than string *b*. Returns zero if string *a* and *b* are equal. Returns a positive number if string *a* is greater than string *b*. Case and diacritical marks are significant (that is, “Hello” and “hello” are not equal).

a	A string.
b	A string.

Note that this is a content comparison of the two strings, not a pointer comparison.

Use `StrCompare` or `StrEqual` to do a case-insensitive comparison of strings.

StrFilled

`StrFilled(`*string*`)`

Returns non-`nil` if the expression *string* evaluates to a string with a length greater than zero. This function returns `nil` if the expression *string* is `nil` or evaluates to an empty string.

string	An expression that evaluates to a string.

StrFontWidth

`StrFontWidth(` *string*`,` *fontSpec*`)`

Returns the width of the string in pixels, if drawn in the specified font.

string	A string.

fontSpec A frame having the following format:

`{family:`*familyName*`,face:`*faceName*`,size:`*pointSize*`}`

For more information about specifying fonts, see the section “Using Fonts for Text and Ink Display” (page 8-17) in *Newton Programmer’s Guide*.

Stringer

`Stringer(` *array* `)`

Returns a string containing all of the elements in the array concatenated.

array An array.

Numbers, strings, characters, and symbols are converted to their natural string representation. For elements that are frames, arrays, and Booleans, this function converts them to an empty string.

StringFilter

`Stringfilter(`*str*`,` *filter*`,` *instruction*`)`

Returns a string filtered according to the instruction.

str The string to filter.

filter A string containing characters to filter from the string.

instruction One of the symbols shown in Table 23-1.

Table 23-1 Instruction symbols for `StringFilter`

Instruction symbol	Meaning
`'passAll`	Returns any letter in *str* that is also in *filter.*
`'passBeginning`	Looks for any character in *filter,* and returns everything in *str* after and including that character.
`'passOne`	Passes only the first letter of a group in the filter and passes everything else. This is useful to collapse an arbitrary number of spaces to one.

Table 23-1 Instruction symbols for `StringFilter` (continued)

Instruction symbol	Meaning
`'rejectAll`	Returns any letter in *str* that is not in *filter*.
`'rejectBeginning`	Rejects any letter that is in *filter* until it reaches a letter that isn't in *filter*. It returns everything past that point.

StringToNumber

`StringToNumber(` *string* `)`

Parses a string representing a number and returns the real number value (never an integer).

string A string.

The format of the real number returned by this function is determined by values in the current locale bundle. The number of digits allowed on both sides of the decimal is 63. Instead of simply changing the constants, a more space-efficient way is to calculate the value. If the number of digits on either side of the decimal point exceeds 63, `StringToNumber` returns `nil`. For more information, see "Localizing Newton Applications" (page 20-1) in the *Newton Programmer's Guide*.

Strings with the following kinds of numbers can be parsed:

```
1
1.2
-12,345
(12,345.78)
```

StrLen

`StrLen(` *string* `)`

Returns the number of characters in a string, excluding the null terminator (if one exists).

string A string.

StrMunger

`StrMunger(` *dstString, dstStart, dstCount, srcString, srcStart, srcCount* `)`

Replaces characters in *dstString* with characters from *srcString* and returns the destination string after munging is complete. This function is destructive to *dstString*.

dstString	The destination string. The string must be writable, if you specify a string literal, or an exception is thrown. Use `Clone` (page 23-3) or a similar function to make a writable copy from a string literal.
dstStart	The starting position within *dstString*.
dstCount	The number of characters to replace in *dstString*. You can specify `nil` for *dstCount* to go to the end of the string.
srcString	A string. This can be `nil` to simply delete the characters.
srcStart	The starting position in *srcString* from which to begin taking characters to place into *dstString*.
srcCount	The number of characters to use from *srcString*. You can specify `nil` to go to the end of *srcString*.

Here is an example:

```
StrMunger("abcdef", 2, 3, "ZYXWV", 0, nil)
"abZYXWVf"
```

`StrMunger` can also be used to concatenate large strings; for example:

```
StrMunger(str1, StrLen(str1)+1, nil, str2, 0, nil);
```

StrPos

`StrPos(` *string*`,` *substr*`,` *start* `)`

Returns the position of *substr* in *string*, or `nil` if *substr* is not found. The search begins at character position *start*. (The first character position in a string is zero.) This function is not case sensitive.

string	A string.
substr	A string.
start	An integer.

Here is an example:

```
StrPos("abcdef", "Bcd", 0)
1
```

StrReplace

`StrReplace(` *string*`,` *substr*`,` *replacement*`,` *count* `)`

Replaces each occurrence of *substr* in *string* with *replacement*. The integer *count* is the number of replacements to perform, or `nil` to replace all occurrences. This function returns the number of replacements performed. This function is destructive to *string*.

string	A string.
substr	A string.
replacement	A string.
count	An integer.

`StrReplace` positions the replacement pointer after the current replacement for each iteration, so a three-time replacement of “a” in “aaa” with “ab” yields “ababab, ”not “abbbaa,” as in some editors.

StrTokenize

`StrTokenize(`*str*`,` *delimiters*`)`

Breaks up a string into chunks as defined by the *delimiters* argument. Each time you call the closure (passing it no arguments) you get back the next token, until there are no more tokens and it returns `nil`.

str A string to break up into tokens.

delimiters Either a character or string (list of characters) that is the delimiter separating the pieces of the string.

For example, to break a sentence into space-separated words, do something like the following:

```
fn := StrTokenize("the quick green fox", $ );
#441BE8D  <function, 0 arg(s) #441BE8D>
      while x := call fn with () do Print(x);
"the"
"quick"
"green"
"fox"
#2        NIL
```

StyledStrTruncate

`StyledStrTruncate(`*string*`,` *length*`,` *font*`)`

Truncates a string to the indicated length, in pixels. (Of course, the length does not include the null terminator.) Returns the truncated string.

string A string.

length An integer specifying the length, in pixels, at which to truncate the string.

font A font specification, which determines how many characters of the string will fit in the specified length. For details on specifying a font, refer to the section "Using Fonts for Text and Ink Display" (page 8-17) in the *Newton Programmer's Guide*.

This function adds an ellipsis (...) to the end of the truncated string.

SubstituteChars

`SubstituteChars(`*targetStr*`, `*searchStr*`, `*replaceStr*`)`

Substitutes characters in *targetStr* by searching for each character in *searchStr* and replacing it by the value of string length in *replaceStr*. That is, for each offset character "`x`" in *targetStr,* if it exists in *searchStr,* it will, in a copy of *targetStr,* replace

```
copy[x]
```

with

`replaceStr[y mod StrLength(replaceStr)]`.

If no substitutions are made, the original string is returned unmodified; otherwise, a modified copy is returned.

For example:

```
SubstituteChars("Text with spaces\tand\ttabs", " \t",
"-")
```

creates

```
Text-with-spaces-and-tabs
```

or

```
SubstituteChars( "(800) 41PHONE",
"ADGJMPTWBEHKNRUXCFILOSVY", "23456789"
```

to create

```
(800) 4174663
```

SubStr

`SubStr(` *string*`,` *start*`,` *count* `)`

Returns a new string containing *count* characters from *string*, starting at position *start*. Character positions begin with zero for the first character.

string	A string.
start	An integer.
count	Can be an integer or `nil` value. If nil, `SubStr` will return characters in to the end of the string.

TrimString

`TrimString(` *string* `)`

Removes any white space (spaces, tabs, and new line characters) from the beginning and end of *string* and returns the result. The *string* parameter is modified.

string	A string.

Upcase

`Upcase(` *string* `)`

Capitalizes each character in *string* and returns the result. The *string* parameter is modified.

string	A string or character (when used to interpret code).

Bitwise Functions

These functions perform logical operations on bits.

Band

`Band(`*a*`, `*b*`)`

Returns an integer result of its operation on one or two integer parameters. Performs bitwise AND.

a An integer.

b An integer.

Bor

`Bor(`*a*`, `*b*`)`

Returns an integer result of its operation on one or two integer parameters. Performs bitwise OR.

a An integer.

b An integer.

Bxor

`Bxor(`*a*`, `*b*`)`

Returns an integer result of its operation on one or two integer parameters. Perform bitwise XOR.

a An integer.

b An integer.

Bnot

`Bnot(`*a*`)`

Returns an integer result of its operation on one or two integer parameters. Performs bitwise NOT, respectively.

a An integer.

b An integer.

Array Functions

These functions operate on and manipulate arrays.

AddArraySlot

`AddArraySlot (`*array*`, `*value*`)`

Appends a new element onto an array.

array	An array.
value	A value to add as a new element in the array.

For example:

```
myArray := [123, 456]
#1634 myArray
addArraySlot (myArray, "I want chopstix")
#12 "I want chopstix"
myArray
#1634 [123, 456, "I want chopstix"]
```

Array

`Array( `*size*`, `*initialValue*` )`

Returns a new array with *size* number of elements that each contain *initialValue*.

size	An integer.
initialValue	A value.

ArrayInsert

`ArrayInsert(`*array*`, `*element*`, `*position*`)`

Inserts an element into an array and returns the modified array.

array	The array to modify.
element	The element to insert into the array.
position	The index where the new element is to be inserted. Specify zero to insert the element at the beginning of the array. Specify the result of `Length(`*array*`)` to insert the element at the end of the array.

The length of the array increases by one.

ArrayMunger

`ArrayMunger(` *dstArray*`, `*dstStart*`, `*dstCount*`, `*srcArray*`, `*srcStart*`,`
srcCount `)`

Replaces elements in *dstArray* using elements from *srcArray* and returns the destination array after munging is complete. This function is destructive to *dstArray*.

dstArray	The destination array.
dstStart	The starting element in the destination array.
dstCount	The number of elements to replace in *dstArray*. You can specify `nil` for *dstCount* to go to the end of the array.
srcArray	An array. You can specify `nil` for *srcArray* to delete the elements.
srcStart	The starting position in the source array from which to begin taking elements to place into the destination array.
srcCount	The number of elements to use from the source array. You can specify `nil` to go to the end of the source array.

Here is an example:

```
ArrayMunger([10,20,30,40,50], 2, 3, [55,66,77,88,99], 0, nil)
[10, 20, 55, 66, 77, 88, 99]
```

Using `ArrayMunger` is the most efficient way to join two arrays.

To put B at the front of A:

```
ArrayMunger(A, 0, 0, B, 0, nil)
```

To put B at the end of A:

```
ArrayMunger(A, Length(A), 0, B, 0, nil)
```

You can also do this with `SetUnion` (page 23-41), which has the additional property of eliminating duplicates. However, `ArrayMunger` is much faster if you don't need to eliminate duplicates.

ArrayRemoveCount

`ArrayRemoveCount(` *array*`,` *startIndex*`,` *count* `)`

Removes one or more elements from an array.

array	The array from which to remove elements.This parameter is modified by this function.
startIndex	An integer that is the index of the first element to remove.
count	An integer specifying the number of elements to remove.

Any elements following those removed are shifted left so that no empty elements remain.

InsertionSort

`InsertionSort(`*array*`, `*test*`, `*key*`)`

Sorts an array, preserving the original relative ordering of equivalent elements.

array The array to modify by sorting.

test Indicates how to sort the array. See the description of the *test* parameter in "Sorted Array Functions" (page 23-43).

key Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter in "Sorted Array Functions" (page 23-43).

This sort performs well on arrays that are nearly sorted already and on very small arrays. This sort is an $O(n^2)$ sort. To sort larger arrays, use `Sort` (page 23-41) or `StableSort` (page 23-42).

Length

`Length (`*array*`)`

Returns the number of elements in an array, the number of slots in a frame, or the size, in bytes, of a binary object.

array An array, frame, or binary object.

For example:

```
myArray := [123, 456, "I want chopstix"]
length (myArray)
#12    3
```

Note that arrays are indexed from 0, but length returns a count of the number of characters. Therefore, the last element of this example is element 2.

Note
If you pass a string to this function, you will get the number of bytes a string occupies. To get the length of strings, use `StrLen` instead. ◆

LFetch

`LFetch(`*array*`, `*item*`, `*start*`, `*test*`, `*key*`)`

Searches an array in a linear manner for the specified element. `LFetch` returns the element or `nil` if it is not found or if *start* is equal to or greater than the length of the array.

array — The array in which to search.

item — The key value for which to search.

start — The array index at which to begin searching.

test — Indicates how to compare key values to test for a match. Specify one of the following symbols for *test*:

`'|=|` — If the objects being compared are immediates and reals, their values are compared for equivalency. For reference objects, their identity is compared.

`'|str=|` — For string objects, the contents of the strings are compared for equivalency.

Alternatively, for nonstandard sorting situations, you can specify a function object that compares two key values and returns a Boolean or integer value indicating whether or not they are equivalent. This function is called to test for matches. The function is passed two parameters, *A* and *B*, where *A* is the *item* parameter passed to `LFetch` and *B* is the array element being tested.

The function must return a non-`nil` value (or zero) if the items are equivalent, or `nil` (or a non-zero integer) if the items are not equivalent.

Note that specifying a function object for *test* results in much slower performance than using one of the predefined symbols.

key — Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter in “Sorted Array Functions” (page 23-43).

This function works just like `LSearch`, except that `LSearch` returns the index of the found item.

If you know that the array you are working with is sorted, you can use the function `BFetch` to search for an element. This function, based on binary search algorithms, is much faster on large arrays than `LFetch` or `LSearch`, though it can be used only on sorted arrays.

LSearch

`LSearch(`*array*`,` *item*`,` *start*`,` *test*`,` *key*`)`

Searches an array in a linear manner for the specified element. It returns the index of the element, or `nil` if it is not found or if *start* is equal to or greater than the length of the array.

array — The array in which to search.

item — The key value for which to search.

start — The array index at which to begin searching.

test — Indicates how to compare key values to test for a match. Specify one of the following symbols for *test*:

- `'|=|` — If the objects being compared are immediates and reals, their values are compared for equivalency. For reference objects, their identity is compared.
- `'|str=|` — For string objects, the contents of the strings are compared for equivalency.

Alternatively, for nonstandard sorting situations, you can specify a function object that compares two key values and returns a Boolean or integer value indicating whether or not they are equivalent. This function is called to test for matches. The function is passed two parameters, *A* and *B*, where *A* is the *item* parameter passed to `LSearch` and *B* is the array element being tested. The function must return a non-`nil` value (or zero) if the items are equivalent, or `nil` (or a non-zero integer) if the items are not equivalent. Note that specifying a function object for *test* results in much slower performance than using one of the predefined symbols.

key Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter in "Sorted Array Functions" (page 23-43).

This function works just like `LFetch`, except that `LFetch` returns the found item instead of its index.

If you know that the array you are working with is sorted, you can use the function `BFind` to search for an element. This function, based on binary search algorithms, is much faster than `LSearch`, though it can be used only on sorted arrays.

NewWeakArray

`NewWeakArray(`*length*`)`

Returns a new weak array with *length* number of elements, which are initialized to `nil`.

length An integer specifying the size of the array to create.

A **weak array** is an array that does not prevent the objects it refers to from being garbage-collected. That is, if the only references to an object are from weak arrays, the object is destroyed during the next garbage collection cycle. When that happens, the references in the weak arrays are replaced with `nil`.

The purpose of weak arrays is to cache objects without preventing them from being garbage-collected. For example, if you want to keep an array of all objects in existence of a certain type, you could add each object to an array as it's created. If you use a regular array, those objects can never be garbage-collected, because there are always references to them in your array, and the system eventually runs out of memory. However, if you use a weak array, its references don't affect garbage collection, so the objects are garbage-collected normally, freeing memory when it is needed.

SetAdd

`SetAdd (`*array*`,`*value*`,`*uniqueOnly*`)`

Appends an element to the specified array and returns the modified array, or `nil` if the element was not added.

array	The array to which `SetAdd` appends the element in *value*.
value	The element to append to the array specified by *array*.
uniqueOnly	Whether to add only unique elements to the array; if the value of this parameter is non-`nil`, `SetAdd` appends *value* to the array only if it is not already present in the array. If the element specified by the *value* parameter is already present in the array, `SetAdd` returns `nil` and does not append the element. If *uniqueOnly* is `nil`, the item is appended to the array without checking whether it is unique.

Note
The type of comparison used in this function is pointer comparison, not content comparison. ◆

SetContains

`SetContains(` *array*`,` *item* `)`

array	An array.
item	An item that may be in the array.

Searches each element of an array to determine if *item* is equal to one of the array elements. If a match is found, this function returns the array index of the matching array element. If *item* is not found in the array, `nil` is returned.

Note
The type of comparison used in this function is pointer comparison, not content comparison. ◆

SetDifference

`SetDifference(` *array1*`,` *array2* `)`

Returns an array that contains all of the elements in *array1* that do not exist in *array2*.

array1	An array.
*array*2	An array.

If *array1* is nil, `nil` is returned.

Notes
The type of comparison used in this function is pointer comparison, not content comparison.

Arguments to this function can't contain duplicate elements (no two elements can be the same object). If they do, the return value of the function is undefined ◆

SetLength

`SetLength (`*array*`,` *length*`)`

Sets the length of an array.

array	An array.
length	An integer.

This function is useful for increasing or decreasing the size of an array. If you increase the size of the array, new elements are filled with a `nil` value. For example:

```
myArray := [123, 456, "I want chopstix"]
#1634 myArray
setLength (myArray, 4)
#1634 [123, 456, "I want chopstix", NIL]
myArray [3] := 789
#3156 789
myArray
#1634 [123, 456, "I want chopstix", 789]
```

SetOverlaps

`SetOverlaps(` *array1*`,` *array2* `)`

Compares each element in *array1* to each element in *array2*, and returns the index of the first element in *array1* that is equal to an element in *array2*. If no equivalent elements are found, `nil` is returned.

array1	An array.
*array*2	An array.

Note
The type of comparison used in this function is pointer comparison, not content comparison. ◆

SetRemove

`SetRemove (`*array*`,` *value*`)`

`SetRemove` removes the specified element from the specified array and returns the modified array. The length of the array shifts left by one and all elements after the deleted element shift by one to the next lowest-numbered array position. If the item is not found in the array, this function returns `nil`.

array	The array from which `SetRemove` removes the specified element.
value	The element to remove from the array specified by *array*.

Note
The type of comparison used in this function is identity comparison, not pointer comparison. ◆

SetUnion

`SetUnion(` *array1*`,` *array2*`,` *uniqueFlag* `)`

Returns an array that contains all of the elements in *array1* and *array2*.

array1 — An array.

*array*2 — An array.

uniqueFlag — If any non-`nil` value is found, `SetUnion` does not include any duplicate items in the array it returns. If *uniqueFlag* is `nil`, all elements from both arrays are included, even if there are duplicates.

If both arrays are `nil`, an empty array is returned.

`SetUnion` can eliminate duplicates. If you do not need that property, you can combine two arrays more efficiently using `ArrayMunger` (page 23-32).

Note
The type of comparison used in this function is identity comparison, not pointer comparison. ◆

Sort

`Sort(` *array*`,` *test*`,` *key* `)`

Sorts an array and returns it after it is sorted. The sort is destructive; that is, the array you give it is modified. The sort also is not stable; that is, elements with equal keys won't necessarily have the same relative order after the sort.

array — An array.

test — Indicates how to sort the array. See the description of the *test* parameter in "Sorted Array Functions" (page 23-43).

key Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter in "Sorted Array Functions" (page 23-43).

This example sorts `myArray` in ascending numerical order according to the `timestamp` slot of the entries:

```
Sort(myArray, '|<|, 'timestamp)
```

This example sorts `myArray` in descending string order according to the first and last names concatenated together:

```
Sort(myArray, '|str>|, func (e) e.first && e.last)
```

StableSort

`StableSort(`*array*`,` *test*`,` *key*`)`

Sorts an array, preserving the original relative ordering of equivalent elements.

array The array to modify by sorting.

test Indicates how to sort the array. See the description of the *test* parameter in "Sorted Array Functions" (page 23-43).

key Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter in "Sorted Array Functions" (page 23-43).

This sort requires working memory, so may not be suitable for extremely large arrays or in low-memory conditions.

Sorted Array Functions

This section describes new functions that operate on sorted arrays. These functions are based on binary search algorithms, hence the “B” prefix to the function names.

IMPORTANT

The arrays you pass to these functions must be ordered, otherwise the results are undefined. To sort an array, you can use the functions `Sort`, `InsertionSort`, or `StableSort`. ▲

These sorted array functions each use *test* and *key* parameters to allow them to be adapted to different data structures. Typically, these functions search, or iterate over several items in an array. As each element in an array is examined, the *key* argument extracts a value, called the key, from the element. That key is treated as specified by the *test* argument.

Here’s an explanation of these parameters:

test Indicates the sort order of the array. Specify one of the following symbols for *test*, to indicate how the array is sorted:

`'|<|`	Sorted in ascending numerical order.
`'|>|`	Sorted in descending numerical order.
`'|str<|`	Sorted in ascending string order, not case sensitive.
`'|str>|`	Sorted in descending string order, not case sensitive.
`'|sym<|`	Sorted in ascending symbol order, based on lexical comparison of symbol name.
`'|sym>|`	Sorted in descending symbol order, based on lexical comparison of symbol name.

Alternatively, for nonstandard sorting situations, you can specify a function object that compares two key values and returns an integer that indicates how they are sorted relative to each other. This function is called by any of the sorted array functions to determine sorting relationships between elements. The function is passed two parameters, *A* and *B*, and must return a positive integer if *A* sorts after *B*, must return zero if *A* sorts equivalently to *B*, and a must return a negative integer if *A* sorts before *B*. Note that specifying a function object for *test* results in much slower performance than using one of the predefined symbols.

key Defines the key within each array element. Specify `nil` to use the array elements directly as they are. You can specify a path expression, in which case the array elements are assumed to be frames or arrays and the path is applied to each element to find the key. You can also specify a function that takes one parameter (the element) and returns the key.

BDelete

`BDelete(`*array*`,` *item*`,` *test*`,` *key*`,` *count*`)`

Deletes elements from an ordered array.

This function returns the number of elements deleted.

array The array to modify.

item The key value for which to search. Elements with this key are deleted.

test Indicates the sort order of the array. See the description of the *test* parameter in “Sorted Array Functions” (page 23-43).

key Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one

	parameter. See the description of the *key* parameter in "Sorted Array Functions" (page 23-43).
count	The maximum number of elements to delete. Specify `nil` to indicate that all matching elements are to be deleted.

BDifference

`BDifference(`*array1*`, `*array2*`, `*test*`, `*key*`)`

Returns a new sorted array containing those elements from *array1* that do not have equivalent elements in *array2*.

array1	The first array. This array is not modified.
array2	The second array. This array is not modified.
test	Indicates the sort order of the array. See the description of the *test* parameter in "Sorted Array Functions" (page 23-43).
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter in "Sorted Array Functions" (page 23-43).

BFetch

`BFetch(`*array*`, `*item*`, `*test*`, `*key*`)`

Uses a binary search to find an element in a sorted array. The leftmost matching element is returned, or `nil` is returned if no elements are found.

array	The array to search.
item	The key value for which to search.
test	Indicates the sort order of the array. See the description of the *test* parameter in "Sorted Array Functions" (page 23-43).

key Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter in "Sorted Array Functions" (page 23-43).

This function works just like `BFind`, except that `BFind` returns the index of the matched item.

BFetchRight

`BFetchRight(`*array*`, `*item*`, `*test*`, `*key*`)`

Uses a binary search to find an element in a sorted array. The rightmost matching element is returned, or `nil` is returned if no elements are found.

array The array to search.

item The key value for which to search.

test Indicates the sort order of the array. See the description of the *test* parameter in "Sorted Array Functions" (page 23-43).

key Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter in "Sorted Array Functions" (page 23-43).

This function works just like `BFindRight`, except that `BFindRight` returns the index of the matched item.

BFind

`BFind(`*array*`, `*item*`, `*test*`, `*key*`)`

Uses a binary search to find an element in a sorted array. The index of the leftmost matching element is returned, or `nil` is returned if no elements are matched.

array The array to search.

item The key value for which to search.

test	Indicates the sort order of the array. See the description of the *test* parameter in “Sorted Array Functions” (page 23-43).
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter in “Sorted Array Functions” (page 23-43).

This function works just like `BFetch`, except that `BFetch` returns the matched item instead of its index.

BFindRight

`BFindRight(`*array*`,` *item*`,` *test*`,` *key*`)`

Uses a binary search to find an element in a sorted array. The index of the rightmost matching element is returned, or `nil` is returned if no elements are found.

array	The array to search.
item	The key value for which to search.
test	Indicates the sort order of the array. See the description of the *test* parameter in “Sorted Array Functions” (page 23-43).
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter in “Sorted Array Functions” (page 23-43).

This function works just like `BFetchRight`, except that `BFetchRight` returns the matching item instead of its index.

BInsert

`BInsert(`*array*`,` *element*`,` *test*`,` *key*`,` *uniqueOnly*`)`

Inserts an element into the proper position in a sorted array. In the case of equivalent elements, the element is inserted to the left of its equivalent.

array	The array to modify.
element	The new element to insert. Note that the *key* parameter is used to extract its key value.
test	Indicates the sort order of the array. See the description of the *test* parameter in “Sorted Array Functions” (page 23-43).
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter in “Sorted Array Functions” (page 23-43).
uniqueOnly	Specify non-`nil` to indicate that the element is not to be inserted if the array already contains an element with an equivalent key value. Specify `'returnElt` to indicate the same thing, and also that this function should return an array element. It returns either the element that was inserted, or if a matching element is found in the array, that element is returned. This is useful when you want to maintain an object list in order to conserve space or ensure pointer equality. Specify `nil` to indicate that the element is to be inserted even if the array already contains an element with an equivalent key. In this case, the new element is inserted to the left of the existing equivalent elements.

This function has three possible return values, as follows:

- It can return `nil`, signaling that the element was not inserted.
- It can return an integer, which is the index at which the element was inserted.

- It can return an array element—either the element that was inserted (if it was unique), or an element that already exists in the array, whose key value matches the key value of the element you wanted to insert. This type of return value can occur only if you specify `'returnElt` for *uniqueOnly*.

Here is an example of how you might use this function with *uniqueOnly* set to `'returnElt` to ensure pointer equality:

```
// :GetStr() returns a string input by the user
bodyColor := BInsert(colorList,:GetStr(),'|str<|,nil,'returnElt);
interiorColor:= BInsert(colorList,:GetStr(),'|str<|,nil,'returnElt);
if bodyColor = interiorColor then Print("bad idea");
```

If `GetString` returns a string already in `colorList`, this code makes sure that the original string is reused. This is why using the `=` operator to test for equality works. It also allows the duplicate string to be garbage-collected, provided there are no remaining references to it.

BInsertRight

`BInsertRight(`*array*`,` *element*`,` *test*`,` *key*`,` *uniqueOnly*`)`

Inserts an element into the proper position in a sorted array. In the case of equivalent elements, the element is inserted to the right of its equivalent. The index at which it was inserted is returned, or `nil` is returned if it was not inserted.

array	The array to modify.
element	The new element to insert. Note that the *key* parameter is used to extract its key value.
test	Indicates the sort order of the array. See the description of the *test* parameter in "Sorted Array Functions" (page 23-43).
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one

parameter. See the description of the *key* parameter in "Sorted Array Functions" (page 23-43).

uniqueOnly A Boolean value. Specify a non-`nil` value to indicate that the element is not to be inserted if the array already contains an element with an equivalent key value. Specify `nil` to indicate that the element is to be inserted even if the array already contains an element with an equivalent key. In the latter case, the new element is inserted to the right of the existing equivalent elements.

BIntersect

`BIntersect(`*array1*`,` *array2*`,` *test*`,` *key*`,` *uniqueOnly*`)`

Returns a new sorted array consisting of the equivalent elements from the two specified arrays.

array1 The first array.; this array is not modified.

array2 The second array; this array is not modified.

test Indicates the sort order of the array. See the description of the *test* parameter in "Sorted Array Functions" (page 23-43).

key Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter in "Sorted Array Functions" (page 23-43).

uniqueOnly A Boolean value. Specify a non-`nil` value to indicate that elements with duplicate key values are not allowed in the resulting array. Note that this works only if *array1* and *array2* are both free of equivalent elements.

Specify `nil` to indicate that elements with duplicate key values are allowed in the resulting array. Note that this guarantees that the resulting array has at least two equivalent elements for every intersecting value, since intersection finds equivalent elements.

If equivalent elements are found in the resulting array, they are ordered as follows: equivalent elements from the same source array retain their original ordering, and equivalent elements from *array1* come before those in *array2*.

BMerge

`BMerge(`*array1*`, `*array2*`, `*test*`, `*key*`, `*uniqueOnly*`)`

Merges two ordered arrays into one new ordered array, which is returned.

array1 — The first array; this array is not modified.

array2 — The second array; this array is not modified.

test — Indicates the sort order of the array. See the description of the *test* parameter in “Sorted Array Functions” (page 23-43).

key — Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter in “Sorted Array Functions” (page 23-43).

uniqueOnly — A Boolean value. Specify a non-`nil` value to indicate that elements with duplicate key values are not allowed in the resulting array. Note that this works only if *array1* and *array2* are both free of equivalent elements.

Specify `nil` to indicate that elements with duplicate key values are allowed in the resulting array.

If equivalent elements are found in the resulting array, they are ordered as follows: equivalent elements from the same source array retain their original ordering, and equivalent elements from *array1* come before those in *array2*.

BSearchLeft

`BSearchLeft(`*array*`,` *item*`,` *test*`,` *key*`)`

Uses binary search to find an element in a sorted array. The index of the smallest and leftmost element that is greater than or equal to *item* is returned. The value `Length(`*array*`)` is returned if *item* is larger than all elements.

array	The array to search.
item	The key value for which to search.
test	Indicates the sort order of the array. See the description of the *test* parameter in "Sorted Array Functions" (page 23-43).
key	Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter in "Sorted Array Functions" (page 23-43).

Here is an example of how this function might be used:

```
// This code extracts all elements between "F" and "Na"
array := ["Ag","C","F","Fe","Hg","K","N","Na","Ni","Pu","Zn"];
pos1   := Min(Length(array)-1,BSearchLeft(array,"F",'|str<|,nil));
pos2   := Max(0,BSearchRight(array,"Na",'|str<|,nil));
ArrayMunger([],0,nil,array, pos1, pos2-pos1+1);
```

BSearchRight

`BSearchRight(`*array*`,` *item*`,` *test*`,` *key*`)`

Uses binary search to find an element in a sorted array. The index of the largest and rightmost element that is less than or equal to *item* is returned. The value –1 is returned if all elements are larger than *item*.

array	The array to search.
item	The key value for which to search.

test Indicates the sort order of the array. See the description of the *test* parameter in “Sorted Array Functions” (page 23-43).

key Defines the key within each array element. Specify `nil`, a path expression, or a function that takes one parameter. See the description of the *key* parameter in “Sorted Array Functions” (page 23-43).

Integer Math Functions

These math functions operate on or return integers. (Some of the floating point functions can also operate on integers.)

Abs

`Abs(`*x*`)`

Returns the absolute value of an integer or real number.

x An integer or real number.

Ceiling

`Ceiling(`*x*`)`

Returns the smallest integer that is not less than the specified real number. (Rounds up the real number to an integer.)

x A real number.

Floor

`Floor(`*x*`)`

Returns the largest integer that is not greater than the specified real number. (Rounds down the real number to an integer.)

x A real number.

GetRandomState

`GetRandomState()`

Returns the current state of the random number generator as a binary object of unspecified format. The random state object is useful only for passing to `SetRandomState`.

Max

`Max(` *a*`,` *b* `)`

Returns the maximum value of the two integers *a* and *b*.

a	An integer.
b	An integer.

Min

`Min(` *a*`,` *b* `)`

Returns the minimum value of the two integers *a* and *b*.

a	An integer.
b	An integer.

Real

`Real(`*x*`)`

Converts the specified integer to a real number.

x	An integer.

Random

`Random (`*low*`,` *high*`)`

Returns a random integer in the range between the two integers, `low` and `high`. The range is inclusive of the numbers `low` and `high`.

low	An integer.
high	An integer.

For example:

```
random (0, 100)
#120      72
```

SetRandomSeed

`SetRandomSeed (`*seedNumber*`)`

Seeds the random number generator with the number you specify.

seedNumber An integer.

When seeded with the same number, the random number generator (`Random` function) returns the same sequence of random numbers each time you reseed it. Do not use 0 to seed the generator as it will return 0 instead of a random number. To generate virtually random numbers, seed it with the value returned from the time function `Ticks`, as follows:

```
SetRandomSeed(Ticks());
```

Note
There is only one random number generator on the Newton, so calls by other functions may interfere with your function getting a consistent sequence of values. ◆

SetRandomState

`SetRandomState(`*randomState*`)`

Resets the random number generator to a previously saved state.

randomState A random state object returned by `GetRandomState`.

The return value of this function is unspecified.

Note that this function provides different functionality from `SetRandomSeed`, which lets you conveniently initialize the random state by providing an integer seed value.

Floating Point Math Functions

NewtonScript provides the floating point math functions documented in this section.

The NewtonScript floating point number system is based on standards 754 and 854, adopted by the Institute of Electrical and Electronics Engineers (IEEE). For more details on IEEE-standard arithmetic than are given here, refer to the *PowerPC Numerics* volume of *Inside Macintosh* or to the *Apple Numerics Manual*, *Second Edition*. These books describe SANE, the standard Apple numeric environment. The NewtonScript environment supports many features of SANE.

NewtonScript floating point numbers (also called *real* numbers) correspond to the double format of the IEEE standards. The number system supports representations for the following values:

- Normal numbers—numbers with approximately 16 decimal digits of precision, ranging from 1.8×10^{308} down to 2.2×10^{-308}.
- Subnormal numbers—numbers ranging from 2.2×10^{-308} down to 4.9×10^{-324}, whose precision diminishes from approximately 16 decimal digits down to less than one digit.
- Signed zeros—the values +0 and –0, which compare equal, but whose behavior differs when, for example, it is divided into nonzero values.
- Signed infinities—the values `+INF` and `-INF`, which indicate results too large to represent or the result of dividing a nonzero numerator by a zero denominator.
- Not-a-Number symbols, or NaNs—values used to represent missing or uninitialized data, or the results of operations, such as $\sqrt{-3}$, which have no meaning in the real number system.

In some application areas, you may find it useful to think of signed zeros and infinities in terms of mathematical *limits*. For example, although +0 and –0 compare as equal, it may be the case for a function *f* that

$\lim_{x \to 0-} f(x) \neq \lim_{x \to 0+} f(x)$, and you may find it useful to exploit that fact.

Similarly, you may find it useful to interpret *g*(`+INF`) as $\lim_{y \to \infty} g(y)$.

The functions in this section follow the model of the arithmetic operations set forth in the IEEE standards; namely, they produce results that are exact when the results are exactly representable in the number system, and otherwise deliver the nearest (or nearly so) representable number to the mathematically correct result. The IEEE standards specify that one or more exceptions be raised when the result of an operation is different from the mathematical result, or when the result is not defined in the real number system. The possible exceptions are

- Inexact—the result is *rounded* or otherwise altered from the mathematical result.
- Underflow—the nonzero result is too tiny to represent except as zero or a subnormal number, and is rounded to less precision than a normal number.
- Overflow—the result is too huge to represent as a normal number.
- Divide by zero—the quotient of a nonzero value divided by zero produces `+INF` or `-INF`, according to the arguments' signs.
- Invalid—the result is not mathematically defined, as is the case with 0/0.

See "Managing the Floating Point Environment" (page 23-73) for further discussion of the handling of floating point exceptions.

One feature of the IEEE standards and SANE is the choice of rounding direction for results not exactly representable. In NewtonScript systems, rounding is *always* to the nearest representable number (with ties going to the value whose least-significant bit is zero). The IEEE standards also specify rounding to the nearest value toward 0, toward `+INF`, or toward `-INF`. However, the standards are written as though the rounding direction is determined by a state variable in the floating point environment (see "Managing the Floating Point Environment" (page 23-73)), while on the ARM family of processors used by NewtonScript systems, rounding direction is determined on an instruction-by-instruction basis.

Acos

`Acos(`*x*`)`

Returns the inverse cosine in radians of *x*. `Acos` raises invalid for $x < -1$ or $x > 1$. It raises inexact for all values except 1. `Acos` returns values between zero and π.

x An integer or real number.

Acosh

`Acosh(`*x*`)`

Returns the inverse hyperbolic cosine of *x*. `Acosh` raises invalid for $x < 1$. It raises inexact for all values except 1. `Acosh(+INF)` returns `+INF`, but `Acosh` never overflows. Its value at the largest finite real number is approximately 710.

x An integer or real number.

Asin

`Asin(`*x*`)`

Returns the inverse sine in radians of *x*. `Asin` raises invalid for $x < -1$ or $x > 1$. It raises inexact for all values except zero and raises underflow for any finite *x* near zero. `Asin` returns values between $-\pi/2$ and $\pi/2$.

x An integer or real number.

Asinh

`Asinh(`*x*`)`

Returns the inverse hyperbolic sine of *x*. `Asinh` raises inexact for any values except zero. `Asinh(-INF)` returns `-INF` and `Asinh(+INF)` returns `+INF`. `Asinh` raises underflow for *x* near zero.

x An integer or real number.

Atan

`Atan(`*x*`)`

Returns the inverse tangent in radians of *x*. It raises inexact for any values except zero. `Atan(-INF)` returns $-\pi/2$ and `Atan(+INF)` returns $\pi/2$. `Atan` returns values between $-\pi/2$ and $\pi/2$. It raises inexact for any nonzero *x*.

x	An integer or real number.

Atan2

`Atan2(`*x*`,`*y*`)`

Returns the inverse tangent in radians of x/y. `Atan2` uses the algebraic signs of *x* and *y* to determine the quadrant of the result. It returns values between $-\pi$ and π. Its special cases are those of `Atan`.

x	An integer or real number.
y	An integer or real number.

Atanh

`Atanh(`*x*`)`

Returns the inverse hyperbolic of *x*. `Atanh` raises invalid for $x < -1$ or $x > 1$. It raises inexact for all valid arguments except zero and raises underflow for any finite *x* near zero. `Atanh(-1.0)` returns `-INF` and `Atan(+1.0)`returns `+INF`.

x	An integer or real number.

CopySign

`CopySign(`*x*`,`*y*`)`

Returns the value with the magnitude of *x* and sign of *y*.

x	An integer or real number.
y	An integer or real number.

Note
The order of the parameters for `CopySign` matches the recommendation of the IEEE 754 floating point standard, which is the opposite of the SANE `copysign` function. ◆

Cos

`Cos(`*x*`)`

Returns the cosine of the radian value *x*. `Cos` raises inexact for all finite arguments except zero. It is periodic with period 2π. `Cos` raises invalid when *x* is infinite.

x An integer or real number.

Cosh

`Cosh(`*x*`)`

Returns the hyperbolic cosine of *x*. `Cosh` raises inexact for all finite arguments except zero. `Cosh(-INF)` and `Cosh(+INF)` return `+INF`. `Cosh` raises overflow for finite values of large magnitude.

x An integer or real number.

Erf

`Erf(`*x*`)`

Returns $\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$, the *error function* of *x*. `Erf` raises inexact for all arguments except zero. It raises underflow for arguments near zero. `Erf(-INF)` returns –1 and `Erf(+INF)` returns 1.

x An integer or real number.

Mathematically, the sum of `Erf(`*x*`)` and `Erfc(`*x*`)` should be 1, though the relationship may not hold when roundoff or underflow affects the results significantly.

Erfc

`Erfc(`*x*`)`

Returns $\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}}\int_x^{\infty} e^{-t^2}\,dt$, the *complementary error function* of *x*. `Erfc` raises inexact for all arguments except zero. `Erfc(-INF)` returns 2 and `Erfc(+INF)` returns +0.

x An integer or real number.

Exp

`Exp(`*x*`)`

Returns e^x, the exponential of the *x*. `Exp` is inexact for all nonzero finite arguments. `Exp(-INF)` returns +0 and `Exp(+INF)` returns `+INF`. `Exp` raises overflow for large, positive, finite *x*, and raises underflow for negative, finite *x* of large magnitude.

x An integer or real number.

Expm1

`Expm1(`*x*`)`

Returns $e^x - 1$, one less than the exponential of *x*. `Expm1` avoids loss of accuracy when *x* is nearly zero, and the difference is nearly zero. `Expm1` is inexact for all nonzero finite arguments. `Expm1(-INF)` returns –1 and `Expm1(+INF)` returns `+INF`. `Expm1` raises overflow for large, positive, finite *x*, and raises underflow for *x* near zero.

x An integer or real number.

Fabs

`Fabs(`*x*`)`

Returns the absolute value of *x*. It never raises an exception.

x An integer or real number.

FDim

`FDim(`*x*`,`*y*`)`

Returns the *positive difference* between its parameters:

- If $x > y$, `FDim` returns $x - y$
- Otherwise, if $x <= y$, `FDim` returns +0
- Otherwise, if *x* is a NaN, `FDim` returns *x*.
- Otherwise (*y* is a NaN), `FDim` returns *y*.

x	An integer or real number.
y	An integer or real number.

FMax

`FMax(`*x*`,`*y*`)`

Returns the maximum of its two parameters. NaN parameters are treated as missing data:

- If one parameter is a NaN and the other is a number, the number is returned.
- If both parameters are NaNs, the first parameter is returned.

(This corresponds to the `max` function in FORTRAN.)

x	An integer or real number.
y	An integer or real number.

FMin

`FMin(`*x*`,`*y*`)`

Returns the minimum of its two parameters. NaN parameters are treated as missing data:

- If one parameter is a NaN and the other is a number, then the number is returned.
- Otherwise, if both are NaNs, the first parameter is returned.

(This corresponds to the `min` function in FORTRAN.)

x An integer or real number.

y An integer or real number.

Fmod

`Fmod(`*x*`,`*y*`)`

Returns the remainder when *x* is divided by *y* to produce a truncated integral quotient. That is, `Fmod` returns the value *x* `-` *y*`*Trunc(`*x*`/`*y*`)`.

x An integer or real number.

y An integer or real number.

Gamma

`Gamma(`*x*`)`

Returns $\Gamma(x)$, the gamma function applied to *x*. `Gamma` raises inexact for all nonintegral *x*. It raises invalid for nonpositive integral arguments *z*. `Gamma(`*p*`)` returns (*p*-1)! for positive, integral *p*, with 0! defined as 1. `Gamma(+INF)` returns `+INF`. `Gamma` can raise overflow.

x An integer or real number.

Hypot

`Hypot(`*x*`,`*y*`)`

Returns the square root of the sum of the squares of *x* and *y*, avoiding the hazards of overflow and underflow when the arguments are large or tiny in magnitude but the result is within range.

x An integer or real number.

y An integer or real number.

IsFinite

`IsFinite(`*x*`)`

Returns `true` if *x* is finite; returns `nil` if *x* is infinite.

x An integer or real number.

IsNaN

`IsNaN(`*x*`)`

Returns `true` if *x* is a NaN; returns `nil` if *x* is a number.

x An integer or real number.

Note

Saying that *x* "is a NaN" and "is not a number" are not the same thing. A NaN is a nonnumerical value in a numerical format; on the other hand, a string such as `"foo"` is not a number because it is not a numerical object. ◆

IsNormal

`IsNormal(`*x*`)`

Returns `true` if *x* is a normal number; returns `nil` if *x* is zero, subnormal, infinite, or a NaN.

x An integer or real number.

LessEqualOrGreater

`LessEqualOrGreater(`*x*`, `*y*`)`

Returns `true` if neither *x* nor *y* is a NaN; therefore, the two arguments are ordered; otherwise, returns `nil`.

x An integer or real number.

y An integer or real number.

LessOrGreater

`LessOrGreater(`*x*`, `*y*`)`

Returns `true` if either $x < y$ or $x > y$; otherwise, returns `nil`.

x	An integer or real number.
y	An integer or real number.

LGamma

`LGamma(`*x*`)`

Returns the natural logarithm of $\Gamma(x)$, the gamma function applied to *x*. `LGamma` raises inexact for all positive *x*. It raises invalid for negative or zero *x*. `LGamma(+INF)` returns `+INF`.

x	An integer or real number.

Log

`Log(`*x*`)`

Returns the natural logarithm of *x*. `Log` raises inexact for positive, finite arguments except 1. `Log(0.0)` returns `-INF` and raises divide by zero. `Log(+INF)` returns `+INF`. `Log` raises invalid for $x < 0$.

x	An integer or real number.

Logb

`Logb(`*x*`)`

Returns the integral value *k* such that $1 \leq |x| * 2^{-k} < 2$, when *x* is finite and nonzero. `Logb(0.0)` returns `-INF` and raises divide by zero. `Logb(-INF)` and `Logb(+INF)` return `+INF`.

Log1p

`Log1p(`*x*`)`

Returns the natural logarithm of `1+`*x*. While accurate for all arguments no less than –1, `Log1p` preserves accuracy when *x* is nearly zero—when computing `Log(1.0 + x)` would suffer from the mere addition of *x* to 1. `Log1p` raises inexact for all finite arguments greater than –1 except 0. It raises invalid for any *x* less than –1 and raises underflow for *x* near zero. `Log1p(-1.0)` returns `-INF` and raises divide by zero. `Log1p(+INF)` returns `+INF`.

x An integer or real number.

Log10

`Log10(x)`

Returns the logarithm base 10 of *x*. Because of the mathematical relationship $\log_{10}(x) = \log(x)/\log(10)$, `Log10` shares the computational properties of `Log`.

x An integer or real number.

NearbyInt

`NearbyInt(x)`

Returns *x* rounded to the nearest integral value. `NearbyInt` differs from `Rint` only in that it does not raise the inexact exception.

x An integer or real number.

NextAfterD

`NextAfterD(x,y)`

Returns the next representable number after *x* in the direction of *y*.

If *x* and *y* are equal, the result is *x*. If either argument is a NaN, `NextAfterD` returns one of the NaN arguments. When *x* is finite but the result is infinite, `NextAfterD` raises overflow. When the result is zero or subnormal, `NextAfterD` raises underflow.

x An integer or real number.

y An integer or real number.

Pow

`Pow(x,y)`

Returns x^y. When $x < 0$, `Pow` raises invalid unless y is an integral value. It can raise inexact, overflow, underflow, and invalid.

x An integer or real number.

y An integer or real number.

RandomX

`RandomX(x)`

Returns a two-element array, based on the random seed *x*. The first element of the result is a pseudo-random number that is the result of the SANE `randomx` function. The second element is the new seed returned by the `randomx` function. The result is an integral value between 0 and $2^{31} - 1$.

x An integer or real number.

Remainder

`Remainder(x,y)`

Returns the *exact* difference $x - n*y$, where *n* is a mathematical integer (as opposed to a NewtonScript integer—*n* may be thousands of bits wide) nearest to x/y in the sense of rounding to nearest integral value. The magnitude of the result is no greater than half the magnitude of *y*. When the result is zero, it has the sign of *x*. `Remainder` raises invalid when *y* is zero or *x* is infinite. It never raises overflow, underflow, or inexact.

x An integer or real number.

y An integer or real number.

RemQuo

`RemQuo(`*x*`,`*y*`)`

Returns a two-element array. The first element is `Remainder(`*x*`, `*y*`)`. The second element is the seven low-order bits of the quotient *x*/*y* rounded to the nearest integer and given the sign of the quotient.

x	An integer or real number.
y	An integer or real number.

Rint

`Rint(`*x*`)`

Is identical to `Nearbyint` except that it raises inexact when its result differs from *x*.

x	An integer or real number.

RintToL

`RintToL(`*x*`)`

Returns an integer obtained by rounding *x* to an integral (real) value and then converting that value to an integer. `RintToL` raises inexact when its result differs in value from *x*. It raises invalid and returns an unspecified value when the rounded value of *x* cannot be represented exactly as an integer object.

x	An integer or real number.

Note

`RintToL` always rounds to nearest integral value. ◆

Round

`Round(x)`

Returns the integral real number obtained from *x* by adding 1/2 to *x* and truncating the result to the nearest integer toward 0. It raises inexact when the result differs from *x*.

x	An integer or real number.

Scalb

`Scalb(x, k)`

Returns $x * 2^k$. `Scalb` avoids explicit computation of 2^k and so avoids the complications of overflow or underflow when 2^k is out of range but the result isn't. `Scalb` can raise overflow, underflow, and inexact. `Scalb` and `Logb` are related by the formula 1 ≤ `Scalb(x, RintToL(-Logb(x)))` < 2 for finite, nonzero *x*.

x	An integer or real number.
y	An integer.

SignBit

`SignBit(x)`

Returns a nonzero integer if the sign of *x* is negative; otherwise (the sign of *x* is positive), returns the integer 0.

x	An integer or real number.

Signum

`Signum(x)`

Returns the integer value –1 if $x < 0$, 0 if $x = 0$, or 1 if $x > 0$. If *x* is an integer, `Signum` returns an integer; otherwise, if *x* is a real number, `Signum` returns a real number. If *x* is neither an integer nor a real, `Signum` throws the exception `kFramesErrNotANumber`.

x	An integer or real number.

Sin

`Sin(x)`

Returns the sine of the radian value *x*. `Sin` raises inexact for all finite values except zero. It is periodic with period 2π. `Sin` raises invalid for infinite *x* and raises underflow for *x* near zero.

x An integer or real number.

Sinh

`Sinh(x)`

Returns the hyperbolic sine of *x*. `Sinh` raises inexact for all finite arguments except zero. `Sinh(-INF)` returns `-INF` and `Sinh(+INF)` returns `+INF`. `Sinh` raises overflow for large finite values and raises underflow near zero.

x An integer or real number.

Sqrt

`Sqrt(x)`

Returns the square root of *x*. It raises invalid for $x < 0$, and can raise inexact for positive *x*.

x An integer or real number.

Tan

`Tan(x)`

Returns the tangent of the radian value *x*. `Tan` raises inexact for all finite values except zero. It is periodic with period π. `Tan` raises invalid for infinite *x* and raises underflow for *x* near zero.

x An integer or real number.

Tanh

`Tanh(x)`

Returns the hyperbolic tangent of *x*. `Tanh` raises inexact for all finite arguments except zero. `Tanh(-INF)` returns –1 and `Tanh(+INF)` returns +1. `Tanh` raises overflow for large finite values and raises underflow near zero.

x	An integer or real number.

Trunc

`Trunc(x)`

Returns the integral real number nearest to but no larger in magnitude than *x*.

x	An integer or real number.

Unordered

`Unordered(x, y)`

Returns `true` if *x* and *y* satisfy none of $x < y$, $x = y$, or $x > y$ (because one or both of *x* and *y* are NaNs); if neither *x* nor *y* is a NaN, they satisfy one of the three order relations and `Unordered` returns `nil`.

x	An integer or real number.
y	An integer or real number.

UnorderedGreaterOrEqual

`UnorderedGreaterOrEqual(x, y)`

Returns `true` if *x* and *y* satisfy $x \geq y$ or are unordered (because one or both of *x* and *y* are NaNs); otherwise, returns `nil`.

x	An integer or real number.
y	An integer or real number.

UnorderedLessOrEqual

`UnorderedLessOrEqual(x, y)`

Returns `true` if *x* and *y* satisfy $x \leq y$ or are unordered (because one or both of *x* and *y* are NaNs); otherwise, returns `nil`.

x	An integer or real number.
y	An integer or real number.

UnorderedOrEqual

`UnorderedOrEqual(x, y)`

Returns `true` if *x* and *y* satisfy $x = y$ or are unordered (because one or both of *x* and *y* are NaNs); otherwise, returns `nil`.

x	An integer or real number.
y	An integer or real number.

UnorderedOrGreater

`UnorderedOrGreater(x, y)`

Returns `true` if *x* and *y* satisfy $x > y$ or are unordered (because one or both of *x* and *y* are NaNs); otherwise, returns `nil`.

x	An integer or real number.
y	An integer or real number.

UnorderedOrLess

`UnorderedOrLess(x, y)`

Returns `true` if *x* and *y* satisfy $x < y$ or are unordered (because one or both of *x* and *y* are NaNs); otherwise, returns `nil`.

x	An integer or real number.
y	An integer or real number.

Managing the Floating Point Environment

The floating point environment is a set of state variables maintained by the Newton system and the underlying processor. The environment contains information about which floating point exceptions have occurred. Floating point exceptions are distinct from NewtonScript exceptions. When floating point exceptions arise (for example, overflow arises when the sum of two huge numbers is too large to represent in the number system), the system raises an exception flag in the environment. Exception flags can be tested, cleared, or raised by functions in this section. Once raised, an exception flag remains raised until you clear it using calls from this section. The predefined constants used to select the floating point exception flags are shown in Table 23-2.

Table 23-2 Floating point exceptions

Constant	Value	Meaning
`fe_Inexact`	`0x010`	inexact
`fe_DivByZero`	`0x002`	divide by zero
`fe_Underflow`	`0x008`	underflow
`fe_Overflow`	`0x004`	overflow
`fe_Invalid`	`0x001`	invalid
`fe_All_Except`	`0x01F`	all exceptions

You can refer to multiple exceptions in a single function invocation by forming the bitwise-OR of the predefined constants, using expressions like `Bor(Bor(fe_Invalid, fe_DivByZero), fe_Overflow)`.

Note

The representation of the floating point environment is implementation dependent. Functions that manipulate the environment and its components do so without exposing their implementation. In particular, the floating point exception flags may or may not be implemented as single bits. ◆

The functions that manage the floating point environment are based on recommended numerical extensions to the ANSI C language. The recommendations for C include functions to test and alter the direction of rounding. Although the direction of rounding is determined by the environment on most systems, Newton systems based on the ARM family of processors determine the rounding direction on an instruction-by-instruction basis, so rounding is not determined by the environment.

You can pass the predefined constant `fe_Dfl_Env` to the functions `FeSetEnv` and `FeUpdateEnv`, which take an environment object as a parameter. `Fe_Dfl_Env` indicates the default environment, in which all exception flags are clear.

FeClearExcept

`FeClearExcept(`*excepts*`)`

Clears the floating point exception flags indicated by *excepts*.

excepts	The integer bitwise-OR of one or more floating point exceptions.

FeGetEnv

`FeGetEnv()`

Returns a data object representing the current floating point environment.

FeGetExcept

`FeGetExcept(`*excepts*`)`

Returns a data object representing the current state of the exception flags indicated by *excepts*.

excepts	The integer bitwise-OR of one or more floating point exceptions.

Note

The representation of the exception flags is unspecified. ◆

FeHoldExcept

`FeHoldExcept()`

Returns a data object representing the current floating point environment, and clears the exception flags.

FeRaiseExcept

`FeRaiseExcept(`*excepts*`)`

Raises the floating point exception flags indicated by *excepts*.

excepts	The integer bitwise-OR of one or more floating point exceptions.

Note

Because floating point exceptions are not tied to the general NewtonScript exception-handling mechanism, raising a flag merely sets an internal variable; raising a flag does not alter the flow of control. ◆

FeSetEnv

`FeSetEnv(`*envObj*`)`

Installs the floating point environment represented by the object *envObj*.

envObj	Either the predefined constant `fe_Dfl_Env` or an object returned by a call to `FeGetEnv` or `FeHoldExcept`.

FeSetExcept

`FeSetExcept(`*flagObj*`,` *excepts*`)`

The parameter *flagObj* is an object containing an implementation-dependent representation of one or more floating point exception flags; *flagObj* must be set by a previous call to `FeGetExcept`. `FeSetExcept` alters the current environment so that those floating point exception flags indicated by *excepts* match the corresponding values in *flagObj*.

flagObj	An object (returned by a previous call to `FeGetExcept`) containing a representation of one or more floating point exception flags.
excepts	The integer bitwise-OR of one or more floating point exceptions.

This function does not raise exceptions; it just alters the state of the flags.

FeTestExcept

`FeTestExcept(`*excepts*`)`

Returns the bitwise-OR of the floating point exceptions indicated by *excepts* whose flags are raised in the current environment.

excepts	The integer bitwise-OR of one or more floating point exceptions.

FeUpdateEnv

`FeUpdateEnv(`*envObj*`)`

Saves the state of the current exception flags, installs the environment represented by *envObj*, and then re-raises the saved exceptions.

envObj Either the predefined constant `fe_Dfl_Env` or an object returned by a call to `FeGetEnv` or `FeHoldExcept`.

You can use `FeUpdateEnv` in conjunction with `FeHoldExcept` to write functions that hide spurious exceptions from their callers:

```
func() begin
   savedEnv := FeHoldExcept(); // clears flags
   result := ...; // ecomputation in which underflow and
                  // divide by zero are benign
   FeClearExcept(BOR(fe_Underflow, fe_DivByZero));
   FeUpdateEnv(savedEnv); // merge old flags with new
   return result
end
```

Financial Functions

These functions perform financial calculations.

Annuity

`Annuity(`*r*`, `*n*`)`

Returns the value of the financial formula $\frac{1-(1+r)^{-n}}{r}$. When *r* is the periodic interest rate and *n* the number of periods, *p*`*Annuity(`*r*`, `*n*`)` is the *present value* of a series of *n* periodic payments of size *p*. `Annuity` is robust over the entire range of *r* and *n*, whether financially meaningful or not.

`Annuity` raises invalid for $r < -1$. When $r = -1$:

- `Annuity(-1, n)` returns –1 for $n < 0$.
- `Annuity(-1, 0)` returns 0.
- `Annuity(-1, n)` returns `+INF` and raises divide by zero for $n > 0$.

Otherwise, $r > -1$. When *r* is nonzero, `Annuity(r, 0)` returns *r*; otherwise, `Annuity(0, n)` returns *n*. `Annuity` raises inexact in all other cases, and can raise overflow or underflow.

r	An integer or real number.
n	An integer or real number.

Compound

`Compound(r, n)`

Returns the value of the financial formula $(1 + r)^n$. When *r* is the periodic interest rate and *n* the number of periods, `P*Compound(r, n)` is the *future value* of a principal amount *P*. `Compound` is robust over the entire range of *r* and *n*, whether financially meaningful or not.

`Compound` raises invalid for $r < -1$. When $r = -1$:

- `Compound(-1, n)` returns `+INF` and raises divide by zero for $n < 0$.
- `Compound(-1, 0)` returns 1.
- `Compound(-1, n)` returns +0 for $n > 0$.

Otherwise, $r > 0$. `Compound(r, 0)` returns 1; `Compound(0, n)` raises invalid when *n* is infinite. `Compound` can raise inexact, overflow, or underflow.

r	An integer or real number.
n	An integer or real number.

GetExchangeRate

`GetExchangeRate(`*country1*`,` *country2*`)`

Returns the currency exchange rate between two countries as a floating point number. This function first checks for an updated rate stored in the system soup and then checks for the rate stored in ROM. This function returns `nil` if it can't find the rate in either place.

country1	A symbol identifying a country.
country2	A symbol identifying a country.

Here is an example:

```
rate := GetExchangeRate('USA, 'Japan);
```

Note

country1 and *country2* can be any country name (with the spaces dropped) for example; SouthKorea. ◆

SetExchangeRate

`SetExchangeRate(`*country1*`,` *country2*`,` *rate*`)`

Saves the currency exchange rate between any two countries as a floating point number in the system soup. Subsequent calls to `GetExchangeRate` return this value instead of the original value stored in ROM.

country1	A symbol identifying a country.
country2	A symbol identifying a country.
rate	The currency exchange rate between *country1* and *country2*.

Here is an example:

```
SetExchangeRate('USA, 'Japan, 87.5);
```

Note

country1 and *country2* can be any country name (with the spaces dropped) for example; SouthKorea. ◆

GetUpdatedExchangeRates

`GetUpdatedExchangeRates()`

Returns a frame containing the updated currency exchange rates that have been stored in the system soup by use of the `SetExchangeRate` function. The `GetUpdatedExchangeRates` function is called by both `GetExchangeRate` and `SetExchangeRate`. Normally, you do not need to call this function unless you want to retrieve all of the updated exchange rates together.

Exception Functions

These functions raise and handle NewtonScript exceptions in an application. For more information about exception handling and how to use these functions, refer to the chapter "Flow of Control" in *The NewtonScript Programming Language*. For a list of system exceptions, see Appendix A, "Error Codes."

The section "Managing the Floating Point Environment" (page 23-73) describes some functions that deal with floating-point exceptions, which are not related to NewtonScript exceptions.

Throw

`Throw(`*name*`, `*data*`)`

Raises an exception and creates an exception frame with the specified name and data.

name	An exception symbol that names the exception being raised.
data	The data for the exception. The possible values for this parameter depend on the composition of *name* and are shown in Table 23-3.

Table 23-3 Exception frame data slot name and contents

Exception symbol	Slot name	Slot contents
Contains part with prefix `type.ref`	`data`	A data object, which can be any NewtonScript object
Contains part with prefix `evt.ex.ms|`	`message`	A message string
Any other	`error`	An integer error code

See the chapter "Flow of Control" in *The NewtonScript Programming Language* for more information on `Throw`.

Rethrow

`Rethrow()`

Reraises the current exception to allow the next enclosing `Try` statement an opportunity to handle it. `Rethrow` throws the current exception again, passing along the same parameters that were passed with the original call to the `Throw` function. This functionality lets you pass control from within an exception handler to the next enclosing `Try` statement.

IMPORTANT

You can call the `Rethrow` function only from within the dynamic extent of an `onexception` clause. ▲

CurrentException

`CurrentException()`

During exception processing (that is, inside the dynamic extent of an `onexception` block), returns the frame associated with the current exception. You can examine the frame returned by `CurrentException` to determine what kind of exception you are handling. For example, you can call the `HasSlot` function to determine if the frame contains a slot named `error`, and take appropriate action thereafter. (The format of the frame depends on the exception, but it always contains a *name* slot with the exception symbol.)

`CurrentException` gives a meaningful response only from within the dynamic extent of an `onexception` clause. Outside the extent of `onexception`, it returns `nil`.

RethrowWithUserMessage

`RethrowWithUserMessage(`*userTitle*`,` *userMessage*`,` *override*`)`

Unhandled exceptions currently end up displaying a `Notify` dialog whose contents are sometimes not very informative to the user. This function allows you to catch an exception, specify a more descriptive message, and then rethrow the exception. If it remains unhandled, the system uses the *userTitle* and *userMessage* in the `Notify` dialog.

userTitle	A string used as the title of the `Notify` dialog.
userMessage	A string used as the body text of the `Notify` dialog.
override	If the exception has already been annotated with a title and a message, this flag controls whether or not to override the existing annotations. Set this slot to non-`nil` to override any existing annotations, or `nil` to preserve them.

This function does not return.

If the exception has a `type.ref` part, the *userTitle* and *userMessage* are added to the existing data. Otherwise, the exception that is rethrown is changed to have a `type.ref` part. For example, an exception named `|evt.ex.bozo|` becomes `|evt.ex.bozo;type.ref|`, and the error is put into the `error` slot of the data frame. Because this change adds an exception part—leaving the existing ones intact—it shouldn't interfere with other `try` blocks looking for the exception (unless they make dangerous assumptions about the format of the exception frame).

Note

Exceptions of the type `|evt.ex.msg|` are changed to `|evt.ex;type.ref|`. You shouldn't use exceptions of the type `|evt.ex.msg|` in final code anyway—they're only for debugging. ◆

Message-Sending Functions

These functions send messages or execute other functions.

Apply

`Apply(`*function*`, `*parameterArray*`)`

Calls a function, passing the supplied parameters. The `Apply` function returns the return value of the function it called.

function	The function to call.
parameterArray	An array of parameters to pass to the function. You can specify `nil` if there are no parameters to pass (this saves allocating an empty array).

`Apply` respects the environment of the function object it is passed. Using `Apply` is similar to using the NewtonScript `call` statement.

`Apply` is useful when you want to call a function, but don't know until run time the number of parameters it takes. If you know the number of parameters the function takes ahead of time, you can use the NewtonScript `call` statement to call the function.

Here's an example of using this function in the Inspector:

```
f:=func(x,y) x*y;
Apply(f, [10,2]);
#50      20
```

The `Apply` call is equivalent to:

```
f(10,2);
```

IsHalting

`IsHalting(`*functionObject*`, `*args*`)`

Returns non-`nil` if the function object returns. `IsHalting` can test a function object before calling it with the specified arguments. It does not actually call the function object; instead, it determines if it will ever return a value (as opposed to getting into an infinite loop). If the function will throw an exception, it returns the symbol `'throws`.

functionObject	The function object you want to test.
args	Array of arguments for the function object.

Perform

`Perform(`*frame*`, `*message*`, `*parameterArray*`)`

Sends a message to a frame; that is, a method with the name of the message is executed in the frame. Both parent and proto inheritance search for the method if it does not exist in the frame. If the method is not found, an exception is thrown.

frame	The frame to which to send the message.
message	A symbol naming the message to send.
parameterArray	An array of parameters to pass along with the message. You can specify `nil` if there are no parameters to pass (this saves allocating an empty array).

The `Perform` function returns the return value of the message it sent.

Note that the method named by *message* is executed in the context of *frame,* not in the context of the frame from within which `Perform` is called.

The `Perform` function is useful when you want to send a message, but don’t know until run time the name of the message or the number of parameters it takes. If you do know these things ahead of time, you can use the standard NewtonScript message sending syntax.

For variations of the `Perform` function, see `PerformIfDefined`, `ProtoPerform`, and `ProtoPerformIfDefined`.

Here's an example of using this function in the Inspector:

```
f:={multiply: func(x,y) x*y};
perform(f, 'multiply, [10,2]);
#50      20
```

Note that

```
f:multiply(10,2)
```

is equivalent to

```
Perform(f, 'multiply,[10,2])
```

PerformIfDefined

`PerformIfDefined(`*receiver*`,` *message*`,` *paramArray*`)`

Sends a message to a frame; that is, a method with the name of the message is executed in the frame. Both parent and proto inheritance search for the method if it does not exist in the frame. If the method is not found, an exception is not thrown.

receiver	The frame to which you want to send the message.
message	A symbol that is the name of the message to send to *receiver*.
paramArray	An array of parameters to pass with the *message*. You can specify `nil` if there are no parameters to pass (this saves allocating an empty array).

This function returns the return value of the message it sent. If the method is not found, this function returns `nil`.

Contrast this function with `Perform` (page 23-84), which is exactly the same, except that `Perform` throws an exception if the method is not found.

Also, contrast this function with `ProtoPerform` and `ProtoPerformIfDefined` (page 23-86), which search only the proto chain for the method.

ProtoPerform

`ProtoPerform(`*receiver*`,` *message*`,` *paramArray*`)`

Sends a message to a frame; that is, a method with the name of the message is executed in the frame. Only proto inheritance searches for the method if it does not exist in the frame. If the method is not found, an exception is thrown.

receiver	The frame to which you want to send the message.
message	A symbol that is the name of the message to send to *receiver*.
paramArray	An array of parameters to pass with the *message*. You can specify `nil` if there are no parameters to pass (this saves allocating an empty array).

This function returns the return value of the message it sent.

Contrast this function with `Perform`, which is exactly the same, except that `Perform` searches both the parent and proto chains for the method.

Also, contrast this function with `PerformIfDefined` and `ProtoPerformIfDefined`, which do not throw exceptions if the method is not found.

ProtoPerformIfDefined

`ProtoPerformIfDefined(`*receiver*`,` *message*`,` *paramArray*`)`

Sends a message to a frame; that is, a method with the name of the message is executed in the frame. Only proto inheritance searches for the method if it does not exist in the frame. If the method is not found, an exception is not thrown.

receiver The frame to which you want the message sent.

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

paramArray An array of parameters to pass with the *message*. You can specify `nil` if there are no parameters to pass (this saves allocating an empty array).

This function returns the return value of the message it sent. If the method is not found, this function returns `nil`.

Contrast this function with `PerformIfDefined` (page 23-85), which is exactly the same, except that `PerformIfDefined` searches both the parent and proto chains for the method.

Also, contrast this function with `Perform` (page 23-84) and `ProtoPerform` (page 23-86), which search both the parent and proto chains for the method.

Deferred Message Sending Functions

This section describes utility functions for delayed and deferred actions.

AddDeferredCall

`AddDeferredCall(`*functionObject*`,`*paramArray*`)`

Queues a function object to execute the next time the system main event loop is executed.

functionObject The function object to execute.

paramArray An array of parameters to pass to the *functionObject*. You can specify `nil` if there are no parameters to pass (this saves allocating an empty array).

This function always returns non-`nil`.

Use this function so that the currently executing method (within which this function is called) has a chance to finish its execution and return up the call chain before the deferred function object is called. The function object is called before the next event is handled.

The `AddDeferredCall` function puts a type of event in a first-in-first-out queue that also contains user actions. Normally, this means that if you call `AddDeferredCall` and then the user taps, the deferred function call occurs first. However, just because the user takes an action does not mean that it is processed immediately. Different components of the Newton operating system are processed in separate threads and thus, you cannot rely on events being processed in a predictable order.

Note also that `ViewIdleScript` methods can be called several times before deferred function calls are executed. Suppose you have, for example, some networking code that initializes a view. Since this is networking code, you have a `ViewIdleScript` method that's called every 200 milliseconds to look for new names on the network. You then have the view initialized by a deferred function call. The `ViewIdleScript` method may be called two or three times before the deferred function call is made.

AddDelayedCall

`AddDelayedCall(`*functionObject*`,`*paramArray*`,`*delay*`)`

Schedules a function object to execute after a specific delay.

functionObject	The function object to execute.
paramArray	An array of parameters to pass to the *functionObject*. You can specify `nil` if there are no parameters to pass (this saves allocating an empty array).
delay	The time in milliseconds after which the *functionObject* is executed

This function always returns non-`nil`.

AddDeferredSend

`AddDeferredSend(`*receiver*`,` *message*`,` *paramArray*`)`

Queues a message to be send the next time the system main event loop executes.

receiver	The frame to which to send the message.
message	A symbol that is the name of the message to send to *receiver*.
paramArray	An array of parameters to pass with the *message*. Specify `nil` if there are no parameters to pass (this saves allocating an empty array).

This function always returns non-`nil`.

Use this function so that the currently executing method (within which this function is called) has a chance to finish its execution and return up the call chain before the deferred message is sent. The message is sent before the next event is handled.

The `AddDeferredSend` function puts a type of event in a first-in-first-out queue that also contains user actions. Normally, this means that if you call `AddDeferredSend` and then the user taps, the deferred message send occurs first. However, just because the user takes an action does not mean that it is processed immediately. Different components of the Newton operating system are processed in separate threads and thus, you cannot rely on events being processed in a predictable order.

Note also that `ViewIdleScript` methods can be called several times before deferred message sends are executed. Suppose you have, for example, some networking code that initializes a view. Since this is networking code, you have a `ViewIdleScript` method that's called every 200 milliseconds to look for new names on the network. You then have the view initialized by a deferred message send. The `ViewIdleScript` method may be called two or three times before the deferred message send is made.

AddDelayedSend

`AddDelayedSend(`*receiver*`,` *message*`,` *paramArray*`,` *delay*`)`

Schedules a message to send after a specific delay.

receiver — The frame to which to send the message.

message — A symbol that is the name of the message to send to *receiver*.

paramArray — An array of parameters to pass with the *message*. You can specify `nil` if there are no parameters to pass (this saves allocating an empty array).

delay — The time in milliseconds after which the *message* is sent.

This function always returns non-`nil`.

AddProcrastinatedCall

`AddProcrastinatedCall(`*funcSymbol*`,` *functionObject*`,` *paramArray*`,` *delay*`)`

Queues a function object to execute at a later time.

funcSymbol — A unique symbol identifying the function object to execute. Append your developer signature to form this symbol to ensure that it is unique in the system.

functionObject — The function object to execute at a later time.

paramArray — An array of parameters to pass to the *functionObject*. You can specify `nil` if there are no parameters to pass (this saves allocating an empty array).

delay — The approximate time in milliseconds after which the *functionObject* is executed. Specify zero to cause the function to execute the next time the system main event loop executes. Zero does not cause immediate execution of the function.

The return value of this function is undefined.

If, prior to executing *functionObject*, another function object with the same identifying *funcSymbol* is queued, the originally queued function is cancelled.

Similarly, the execution of this second function can be preempted by yet another queued function with the same *funcSymbol*, and so on.

This function is useful for preventing lengthy operations from occurring multiple times in a row when a single operation would suffice. For example, you might call `EntryChange` in several places in your code, to flush an entry to a soup. However, you really need to call `EntryChange` only once, after the last slot changes. You could use a function call like this to help prevent multiple calls to `EntryChange` from occurring one after another:

```
AddProcrastinatedCall('|flush:mySignature|,
                      functions.EntryChange, [entry], 0);
```

AddProcrastinatedSend

`AddProcrastinatedSend(`*msgSymbol*`,`*receiver*`,`*message*`,`*paramArray*`,`*delay*`)`

Queues a message to send at a later time.

msgSymbol	A unique symbol identifying the message to send. Append your developer signature to form this symbol to ensure that it is unique in the system.
receiver	The frame to which you want to send the message.
message	A symbol that is the name of the message to send to *receiver*.
paramArray	An array of parameters to pass with the *message*. You can specify `nil` if there are no parameters to pass (this saves allocating an empty array).
delay	The approximate time in milliseconds after which the *message* is sent. Specify zero to cause the function to execute the next time the system main event loop executes. Zero does not cause immediate execution of the function.

The return value of this function is undefined.

If, prior to sending *message*, another message with the same identifying *msgSymbol* is queued, the originally queued message is cancelled. Similarly, sending the second message can be preempted by yet another queued message with the same *msgSymbol*, and so on.

This function is useful for preventing lengthy operations from occurring multiple times in a row when a single operation would suffice. Here is an example of calling this function:

```
AddProcrastinatedSend('|update:mySignature|, base,
'updateViews, nil, 0);
```

Data Extraction Functions

These functions extract chunks of data from other objects of various types.

All integers are stuffed and extracted in 2's-complement **big-endian** form. In this form, byte 0 is the most significant byte, as found on the Newton and Macintosh. The opposite of this is **little-endian,** where byte 0 is least-significant byte, as found on Intel-based computers. For example, the number 0x12345678 is stored as:

big-endian	12	34	56	78
little-endian	78	56	34	12

All Unicode conversions use the Macintosh extended character set for codes greater than or equal to 128.

ExtractByte

`ExtractByte(`*data*`,` *offset*`)`

Returns one signed byte from the given offset.

data — The data from which to extract the return value.

offset — An integer giving the position in data from which to extract the return value.

For example:

```
ExtractByte("\u12345678",0);
#3FC      255
```

ExtractBytes

`ExtractBytes(`*data*`,` *offset*`,` *length*`,` *class*`)`

Returns a binary object of class *class* containing *length* bytes of data starting at *offset* within *data*.

data	The data from which to extract the return value.
offset	An integer giving the position in data from which to extract the return value.
length	An integer giving the number of bytes to extract.
class	A symbol specifying the class of the return value.

ExtractChar

`ExtractChar(`*data*`,` *offset*`)`

Returns a character object of the character at the given *offset* in the *data*.

data	The data from which to extract the return value.
offset	An integer giving the position in data from which to extract the return value.

Gets one byte at the specified offset, converts it to Unicode, and returns the character it makes from it.

For example:

```
ExtractChar("\uFFFFFFFF",0);
//$\u02C results from a ASCII to UNICODE conversion.
#2C76     $\u02C7
//Note $a is at offset 1 in a Unicode string
ExtractChar("abc",0);
#6        $\00
```

```
ExtractChar("abc",1);
#616      $a
```

ExtractLong

`ExtractLong(`*data*`,` *offset*`)`

Returns an integer object of the low 29 bits of an unsigned long int at the given offset, right-justified (that is, the low 29 bits of a 4-byte value).

data The data from which to extract the return value.

offset An integer giving the position in data from which to extract the return value.

Reads four bytes at the specified offset, but ignores the high-order bits (first two). Returns a 30-bit signed value.

```
ExtractLong("\uFFFFFFFF",0);
#FFFFFFFC -1
ExtractLong("\uC0000007",0);
#1C       7
```

ExtractXLong

`ExtractXLong(`*data*`,` *offset*`)`

Returns an integer object of the high 29 bits of an unsigned long int at the given offset, right-justified (that is, the high 29 bits of a 4-byte value).

data The data from which to extract the return value.

offset An integer giving the position in data from which to extract the return value.

For example:

```
ExtractXLong("\u0000000F",0);
#4        1
```

ExtractWord

`ExtractWord(`*data*`, `*offset*`)`

Returns a 2-byte signed integer object from the given offset.

data The data from which to extract the return value.

offset An integer giving the position in data from which to extract the return value.

For example:

```
ExtractWord("\uFFFFFFFF",0);
#FFFFFFFC -1
//if you want unsigned use:
band(ExtractWord(-),0xFFFF);
#40004    65535
```

ExtractCString

`ExtractCString(`*data*`, `*offset*`)`

Returns a Unicode string object derived from the null-terminated C-style string at the given offset.

data The data from which to extract the return value.

offset An integer giving the position in data from which to extract the return value.

ExtractPString

`ExtractPString(`*data*`, `*offset*`)`

Returns a Unicode string object derived from the Pascal-style string (a length byte followed by text) at the given offset.

data The data from which to extract the return value.

offset An integer giving the position in data from which to extract the return value.

ExtractUniChar

`ExtractUniChar(`*data*`, `*offset*`)`

Gets two bytes at the specified offset and returns the Unicode character represented by those bytes.

data	The data from which to extract the return value.
offset	An integer giving the position in data from which to extract the return value.

For example:

```
ExtractUniChar("abc",0);
#616      $a
```

Data Stuffing Functions

These functions are used to stuff chunks of data into objects of various types.

All integers are stuffed in 2's-complement **big-endian** form. For a discussion of this, see "Data Extraction Functions" (page 23-92).

StuffByte

`StuffByte(`*obj*`, `*offset*`, `*toInsert*`)`

Writes the low-order byte of *toInsert*, at the specified *offset* in *obj*.

obj	A binary object into which to stuff the data.
offset	The position, in bytes, in *obj* at which to begin stuffing.
toInsert	The data to stuffed into *obj*.

For example:

```
x := "\u00000000";
StuffByte(x,0,-1);
x[0]
```

```
#FF006    $\uFF00

x := "\u00000000";
StuffByte(x,0,0xFF);
x[0]
#FF006    $\uFF00
```

StuffChar

StuffChar(*obj*, *offset*, *toInsert*)

Stuffs one byte into *obj* at the specified offset.

obj	A binary object into which to stuff the data.
offset	The position, in bytes, in *obj* at which to begin stuffing.
toInsert	A character or integer to stuff into *obj*. You pass it a two-byte Unicode value as *toInsert*. The function makes a one-byte character from that value and stuffs the one-byte character.

This accepts a character or integer as its third parameter, *toInsert*:

- If *toInsert*: is an integer: writes the low byte of *toInsert*.
- If *toInsert*: is a character: converts from Unicode and writes a byte.

For example:

```
x := "\u00000000";
StuffChar(x,1,Ord($Z));
x[0]
#5A6      $Z

x := "\u00000000";
StuffChar(x,1,-1);
x[0]
#1A6      $\1A
```

```
ExtractByte(x,1)
#68       26
ExtractByte(x,0)
#0        0
```

StuffCString

`StuffCString(`*obj*`, `*offset*`, `*aString*`)`

Converts a Newton Unicode string into a null-terminated C-style string and stuffs it at the given offset into a binary object.

obj	A binary object into which to stuff the data.
offset	The position, in bytes, in *obj* at which to begin stuffing.
aString	A Unicode string to stuff into *obj*.

The string *aString* is converted into ASCII format using Macintosh roman string encoding. It is then stuffed into *obj*, beginning at the byte offset *offset*. It is followed by a null-byte terminator.

This function throws an exception if *aString* does not fit into *obj* beginning at the given offset, or if the offset is negative. The length of *obj* is not altered.

StuffLong

`StuffLong(`*obj*`, `*offset*`, `*toInsert*`)`

Writes four bytes at the specified offset using the 30 bit signed value you pass it as the third parameter, and sign extends it to 32 bytes.

obj	A binary object into which to stuff the data.
offset	The position, in bytes, in *obj* at which to begin stuffing.
toInsert	The data to stuff into *obj*.

For example:

```
x := "\u00000000";
StuffLong(x,0,-1);
x[0]
```

```
#FFFF6    $\uFFFF
x[1]
#FFFF6    $\uFFFF
x := "\u00000000";
StuffLong(x,0,0x3FFFFFFA);
x[0]
#FFFF6    $\uFFFF
x[1]
#FFFA6    $\uFFFA
```

StuffPString

`StuffPString(`*obj*`, `*offset*`, `*aString*`)`

Converts a Newton Unicode string into a Pascal-style string (a length byte followed by text) and stuffs it at the given offset into a binary object.

object	A binary object into which to stuff the data.
offset	The position, in bytes, in *obj* at which to begin stuffing.
aString	A Unicode string to stuff into *obj*. This string must be no longer than 255 characters.

The string *aString* is converted into ASCII format using Macintosh roman string encoding. Then a length byte followed by the string is stuffed into *obj*, beginning at the byte offset *offset*. The length byte indicates the number of characters in the string.

This function throws an exception if *aString* does not fit into *obj* beginning at the given offset, or if the offset is negative. The length of *obj* is not altered.

StuffUniChar

`StuffUniChar(`*obj*`, `*offset*`, `*toInsert*`)`

Stuffs the two-byte Unicode encoding for the character indicated by *toInsert* into *obj* at the specified offset.

obj	A binary object into which to stuff the data.
offset	The position, in bytes, in *obj* at which to begin stuffing.

toInsert A character or integer to stuff into *obj*.

For example:

```
x := "\u00000000";
StuffUniChar(x,0,"\uF00F"[0]);
x[0]
#F00F6    $\uF00F

x := "\u00000000";
StuffUniChar(x,0,0x0AA0);
x[0]
#AA06     $\u0AA0
```

StuffWord

`StuffWord(`*obj*`, `*offset*`, `*toInsert*`)`

Writes the low-order two bytes of *toInsert* at the specified offset.

obj A binary object into which to stuff the data.

offset The position, in bytes, in *obj* at which to begin stuffing.

toInsert The data to stuff into *obj*.

For example:

```
x := "\u00000000";
StuffWord(x,0,0x3FFF1234);
x[0]
#12346    $\u1234

x := "\u00000000";
StuffWord(x,0,-1);
x[0]
#FFFF6    $\uFFFF
```

Getting and Setting Global Variables and Functions

These functions get, set, and test for the existence of global variables and functions.

GetGlobalFn

`GetGlobalFn(`*symbol*`)`

Returns a global function. If the function is not found, `nil` is returned.

symbol	A symbol naming the global function to get.

GetGlobalVar

`GetGlobalVar(`*symbol*`)`

Returns the value of a slot in the system globals frame. If the slot is not found, `nil` is returned.

symbol	A symbol naming the global variable whose value to get.

GlobalFnExists

`GlobalFnExists(`*symbol*`)`

Returns non-`nil` if the global function identified by *symbol* exists, otherwise returns `nil`.

symbol	A symbol naming the global function's existence to check.

GlobalVarExists

`GlobalVarExists(`*symbol*`)`

Returns non-`nil` if the global variable identified by *symbol* exists, otherwise returns `nil`.

symbol	A symbol naming the global variable's existence to check.

DefGlobalFn

`DefGlobalFn(`*symbol*`, `*function*`)`

Defines a global function. The symbol identifying the function is returned.

symbol	A symbol naming the global function to define. To avoid naming conflicts with other global functions, choose a name that includes your `appSymbol`, which includes the developer signature you have registered with Newton DTS.
function	A function object.

Note that the global function is destroyed if the system is reset.

You must remove any global functions created by your application when your application is removed. You can do this with `UnDefGlobalFn` in the application `RemoveScript` function.

IMPORTANT

Do not create global functions unless it is absolutely necessary. Global functions occupy NewtonScript heap space. They can conflict with system global functions and other applications' global functions. In most cases, you can use methods in your application base view instead of global functions. ▲

DefGlobalVar

`DefGlobalVar(`*symbol*`, `*value*`)`

Defines a global variable—that is, a slot in the system globals frame. The value of the variable is returned.

symbol A symbol naming the global variable to define. To avoid naming conflicts with other globals, choose a name that includes your `appSymbol`, which includes the developer signature you have registered with Newton DTS.

value The value to assign to the global variable.

The system ensures that the object created exists entirely in internal RAM (it calls `EnsureInternal` on the object identified by *symbol*. Note that the global variable is destroyed if the system is reset.

You must remove any globals created by your application when your application is removed. You can do this with `UnDefGlobalVar` in the application `RemoveScript` function.

IMPORTANT

Do not create global variables unless it is absolutely necessary. Global variables occupy NewtonScript heap space. They can conflict with system globals and other applications' globals. In most cases, you can put any global data that you need in your application base view or in a soup. ▲

UnDefGlobalFn

`UnDefGlobalFn(`*symbol*`)`

Removes a global function you previously defined. This function returns `nil`.

symbol A symbol naming the global function to remove.

UnDefGlobalVar

`UnDefGlobalVar(`*symbol*`)`

Removes a previously defined global variable. This function returns `nil`.

symbol A symbol naming the global variable to remove.

Debugging Functions

These functions are used to debug Newton applications. See the *Newton Toolkit User's Guide* for complete details on debugging an application.

▲ **WARNING**
Do not use these functions in release applications. ▲

BreakLoop

`BreakLoop()`

Halts execution and allows you to examine the state of your application on the Newton. You can also execute any valid NewtonScript code, including the functions built into the Newton, while in a break loop.

If the Newton executes the `BreakLoop` function when it's already in a break loop, it enters a subsidiary `breakloop`.

To exit a break loop, click the Exit Break Loop button or execute the `ExitBreakLoop` function.

DV

`DV(`*view*`)`

Displays a view and its children in the Inspector window.

view The view object that you want to display.

The `DV` function always returns `nil`.

A quick way to display the contents of a view is to use the Debug function. To display the view made from a template named helloBase, for example, you would enter this text:

```
DV(Debug("helloBase"));
```

If a view is visible on the screen, `DV` produces a display of the view contents in the Inspector window and, if the application was built with Compile for Debugging in effect, flashes the view on the Newton screen. If the view is not visible, `DV` returns `nil`.

You can also specify one of three special symbols for the view argument:

- The `'viewFrontMost` symbol returns the frontmost view on the screen that has the vApplication flag set in its `viewFlags` slot
- The `'viewFrontMostApp` symbol returns the frontmost view on the screen that has the vApplication flag set in its `viewFlags` slot, but not including floating views (those with `vFloating` set in their `viewFlags` slot)
- The `'viewFrontKey` symbol returns the view on the screen that currently accepts keystrokes

GC

```
GC()
```

Forces a garbage collection in the NewtonScript frames heap, a reserved area of system memory from which the system allocates space for all NewtonScript objects.

The `GC` function frees all allocated objects that are no longer referenced. The Newton system software automatically performs a garbage collection when memory is needed. You can call `GC` to ensure that unallocated space is consolidated before you call the `Stats` or `TrueSize` functions.

The `GC` function always returns `nil`.

ExitBreakLoop

`ExitBreakLoop()`

Exits a break loop.

When an Inspector connection is open, the Newton enters a break loop if

- it executes the `BreakLoop` function or
- an exception occurs while `BreakOnThrows` is non-`nil`.

If one of these conditions arises when the Newton is already in a break loop, it enters a subsidiary break loop. Execution of the `ExitBreakLoop` function exits only the current-level break loop. Program execution resumes when you exit the first-level break loop.

The `ExitBreakLoop` function always returns `nil`.

StackTrace

`StackTrace()`

Prints a stack trace in the Inspector window.

The `StackTrace` function always returns `nil`.

Stats

`Stats()`

Returns the amount of free memory in the NewtonScript heap and displays the amount of free memory and the size of the largest area of free memory.

The `Stats` function returns the amount of free memory in bytes. You can call `GC` first to ensure that any space occupied by unreferenced objects has been reclaimed.

StrHexDump

`StrHexDump(`*object*`,` *spaceInterval*`)`

Returns a hexadecimal string representing the value of the object.

object	The binary object you want to examine.
spaceInterval	An integer specifying where to put spaces in the hexadecimal string output. To put spaces after every four bytes, for example, specify 4. For no spaces at all, specify 0.

You can use `StrHexDump` to examine the contents of a binary object.

Note

This function can return an extremely large string object, depending on the length of the binary object you specify. Use it carefully. ◆

TrueSize

`TrueSize(`*object*`,` *filter*`)`

Measures the total RAM requirements of an object by adding together its size and the sizes of all objects it points to. The total does not include read-only objects, such as objects in ROM or in the package.

object — A reference to the object to be measured.

If you pass a value of `nil`, `TrueSize` looks at the root frame, the global variables, and the undo-buffer frame. You use this option when looking for references to an object, as described in the description of the filter parameter.

filter — A filter that controls what data is collected and displayed.

`nil`	Displays the summary of objects by type and the frame in which the data was collected.

`'all`	Displays the summary and a list of all objects measured, sorted by the size of the objects exclusive of the objects they point to.
`'allKids`	Displays the summary and a list of all objects measured, sorted by the size of the objects inclusive of the objects they point to.
`classSymbol`	Displays the summary and all objects of the specified class.
`reference`	Displays the summary and all paths within the specified object that point to the specified reference. To look for the reference throughout most of memory, pass a value of `nil` for the object parameter.

The `TrueSize` function summarizes the number and kinds of objects measured and collects specific data about some or all of them.

ViewAutopsy

`ViewAutopsy(`*functionSpec*`)`

Provides two ways to examine how views are drawn. Supply a value of `nil` to turn on and off the outlining of views, in which the boundary of each view is marked by a gray line. Supply an integer to specify a pause (in ticks) after each view is drawn.

functionSpec A value that specifies which drawing option you're manipulating:

`nil`	Toggles view outlining. This option affects both the Newton screen and printed output. Use it for debugging justification and view-layering problems.

`integer`	Forces a pause for the specified number of ticks after each view is drawn. This option allows you to examine the drawing of views, so you can eliminate unnecessary redrawing. A value of 0 turns off the delay option with no effect on outlining.

Miscellaneous Functions

These functions send messages or execute functions.

AddMemoryItem

`AddMemoryItem(`*memSymbol*`,` *value*`)`

Adds a memorized value that can be any string you want to pass as the second parameter to this function. Unlike `AddMemoryItemUnique` (page 23-110), this function does not test for uniqueness.

memSymbol — An identifier symbol that names the memorized value that can be retrieved later with `GetMemoryItems` (page 23-119). Use a symbol that has your developer signature appended to ensure that the symbol is unique to the system.

value — The string to add to the memorized items.

For example, if you call

```
AddMemoryItem('|widget:MYSIG|, "Frazzle Wrench") ;
```

you can later call

```
GetMemoryItems('|widget:MYSIG|) ;
```

to get

```
[{item:"Frazzle Wrench"}]
```

This function returns an array of memory items that have been added under that *memSymbol*.

AddMemoryItemUnique

`AddMemoryItemUnique(`*memorySlot*`,` *value*`,` *testFunc*`)`

Adds a memorized value that can be any object you want to pass as the second parameter to this function. For example, when used with a picker, the second parameter is usually an object from the picker.

memSymbol — An identifier symbol that names the memorized value to retrieve later with `GetMemoryItems`. You must use a symbol that has your developer signature appended to ensure that the symbol is unique to the system. See `AddMemoryItem` (page 23-109) for an example.

value — The object to add to the memorized items.

testFunc — A function object that must accept two parameters, which are two memorized values. The system calls this function object and compares the memorized values returning non-`nil` if the values are equivalent and `nil` otherwise.

If you pass `nil` for the *testFunc* parameter, this function behaves like `AddMemoryItem`; that is, the item is added even if it's not unique.

BackLight

`BackLight(state)`

Turns the backlight on or off. The return value is unspecified.

state — A Boolean value. If `nil`, the backlight is turned off; if non-`nil`, the backlight is turned on.

Note

Use the `Gestalt` function to determine if the Newton has backlighting hardware before using this function. ◆

BackLightStatus

`BackLightStatus()`

Returns nil if the backlight is off and non-`nil` if the backlight is on.

Note
Use the `Gestalt` function to determine if the Newton has backlighting hardware before using this function. ◆

BinEqual

`BinEqual(`*a*`,` *b*`)`

Compares two binary objects' data as raw bytes. Returns non-`nil` if they are identical.

a	A binary object.
b	A binary object.

BinaryMunger

`BinaryMunger(` *dst*`,` *dstStart*`,` *dstCount*`,` *src*`,` *srcStart*`,` *srcCount* `)`

Replaces bytes in *dst* using bytes from *src* and returns *dst* after munging is complete. This function is destructive to *dst*.

dst	A value to change.
dstStart	The starting position in *dst*.
dstCount	The number of bytes to replace *in dst*. You can specify `nil` for *dstCount* to go to the end of *dst*.
src	A value. Can be `nil` to simply delete the contents of *dst*.
srcStart	The starting position in the source binary from which to begin taking elements to place into the destination binary.
srcCount	The number of bytes to use from the source binary. You can specify `nil` to go to the end of the source binary.

Bytes are numbered counting from zero.

Chr

`Chr(`*integer*`)`

Converts a decimal integer to its Unicode character equivalent.

integer An integer.

Here is an example:

```
chr(65)
$A
```

Compile

`Compile(`*string*`)`

Compiles an expression sequence and returns a function that evaluates it.

string The expression to compile.

Here are two examples; in the first example, `x` is a local variable:

```
compile("x:= {a:self.b, b:1234}")
#440F711  <CodeBlock, 0 args #440F711>
f:=compile("2+2")
f();
#440F712 4
```

Note

All characters used in NewtonScript code must be 7-bit ASCII. While this usually is no problem, it may create problems with `Compile` in certain situations. Suppose you tried this call:

```
Compile ("blah, blah, blah,  \u0F0F\u")
```

The Unicode character is not a 7-bit character; it is 16 bits. Therefore, you get an error. (The `\u` switch turns on Unicode character mode.) You should do this instead:

```
Compile ("blah, blah, blah,  \\u0F0F\\u")
```

The backslash escape character preceding the `\u` prevents Unicode mode from being turned on for the compile. (The `\u` is read simply as the string `"\u"` instead of the Unicode switch.)

Note, also, that

```
compile("func()...")
```

returns a function that constructs the function. The environment is captured when the function constructor is executed:

```
f := compile("func()b");
x := {a:f, b:0};
g:=x:a();
#440F713 <CodeBlock, 0 args #440F711>
```

Executing the function construction captures the message environment with `x` as receiver.

```
g();
#440F714 0
```

Now it can find `b`. ◆

Gestalt

`Gestalt(`*selector*`)`

Returns information about the Newton System depending on the value of the selector parameter.

selector	A constant that specifies the type of information that is returned on the system. `kGestalt_SystemInfo` and `kGestalt_Backlight` are the only constants currently supported.

The `kGestalt_Backlight` constant can be passed to `Gestalt` to determine if the unit supports backlighting. `Gestalt` will return either

– `nil`, indicating the unit does not have backlight hardware.

– a one element array where the value of the element (`nil`/ non-`nil`) indicates if backlight hardware is present.

The following code correctly tests if a unit has a backlight:

```
local result:=Gestalt(kGestalt_Backlight);
   if result and result[0] then
  // unit has backlighting
   else
  // unit does not have backlighting
```

`kGestalt_SystemInfo`, which has a value of 0x1000003, returns a frame with the following slots:

`manufacturer`
: A decimal integer indicating the manufacturer of the Newton Device.

`machineType`
: A decimal integer indicating the hardware type this ROM was built for.

`ROMStage`
: A decimal integer indicating the language (English, German, French) and the stage of the ROM (alpha, beta, final).

`ROMVersion`
: A decimal integer indicating the major and minor ROM version numbers. The major number is in front of the decimal place; the minor number follows.

Note

The `Machinetype`, `ROMStage` and `ROMVersion` slots provide internal configuration information and should not be relied on. ◆

`screenWidth`
: An integer representing the width of the screen in pixels. The width takes into account the current screen orientation.

  For example, on the MessagePad 120, because the screen width is 240 and the screen height is 320, in portrait orientation `Gestalt` returns a width of 240. If the screen is rotated, `Gestalt` returns a width of 320.

`screenHeight`
: An integer representing the height of the screen in pixels.

`screenResolutionx`
: An integer representing the number of horizontal pixels per inch. For screens with square pixels, `screenResolutionx` equals `screenResolutiony`. On the MessagePad 120, for example, both `screenResolutionx` and `screenResolutiony` equal 85.

`screenResolutiony`
: An integer representing the number of vertical pixels per inch.

`screenDepth`
: The bit depth of the LCD screen. For the MessagePad 120, the LCD supports a monochrome screen depth of 1.

`patchVersion`
: Returns 0 on an unpatched Newton and nonzero on a patched Newton.

`ROMVersionString`

The user-visible string that identifies the version of the installed ROM and the installed patch, if any.

The first part of the string is a "functionality level" indicating whether the ROM has 1.x or 2.x functionality. All pre-2.x units, except the original MessagePads, have "1.3" as their functionality level. 2.x and later units have "2.0."

The second part of the string is a six-digit number in parentheses that is an encoded representation of ROM and Update information.

Here is an example of code to use to decode the value of the `ROMVersion` slot in the returned frame:

```
global VersionDecode(ROMVersion)
begin
        local minor := BAND(ROMVersion, 0xFFFF);
        local major := BAND(ROMVersion>>16, 0xFFFF);
        [ Floor(StringToNumber(BAND(major>>12, 0xF)
                                & BAND(major>>8, 0xF)
                                & BAND(major>>4, 0xF)
                                & BAND(major, 0xF))),
        Floor(StringToNumber(BAND(minor>>12, 0xF)
                                & BAND(minor>>8, 0xF)
                                & BAND(minor>>4, 0xF)
                                & BAND(minor, 0xF)))];
end;
VersionDecode(Gestalt(0x1000003).ROMVersion);
```

Here is another example of code to test if your Newton is running 2.0. It returns non-`nil` if the major version is 2:

```
global VersionTwo() BAND((Gestalt(0x1000003).ROMVersion)
   >>16, 0xFFFF) = 0x0002;
```

IMPORTANT

Do not assume that if the Newton is running version 2.0 or later that a particular feature exists. You still need to test the Newton to make sure the feature exists. ▲

GetAppName

`GetAppName(`*appSymbol*`)`

Retrieves a user-visible application name for another application. `GetAppName` returns a string that is the name of the application.

appSymbol	A symbol identifying the application whose name you want.

This function looks in several places to find a string that is the application name. Here is how it searches:

1. First, `GetAppName` checks in the application base view for a slot named `appName`, and if found, returns the string found therein.
2. Next, the function looks in the application base view for a `title` slot, and if found, returns the string found therein.
3. Then, the function returns the string used for the application name below its icon in the Extras Drawer.
4. If none of the above attempts succeeds in finding a name, the *appSymbol* is converted to a string and returned.

GetAppParams

`GetAppParams( )`

Returns a frame containing information about the screen size and other system configuration items. The frame returned contains these slots:

`appAreaTop`	The y coordinate of the top-left corner of the screen.
`appAreaLeft`	The x coordinate of the top-left corner of the screen.
`appAreaWidth`	The width of the screen in pixels.
`appAreaHeight`	The height of the screen in pixels.
`buttonBarPosition`	A symbol (`'top`, `'left`, `'bottom`, or `'right`) indicating where the button bar is. This is useful if you want to locate your application flush against the button bar. (The button bar contains the Newton application/scroller icons.)

In order to make your applications compatible with future Newton systems, you must compute the size and location of your application based on the values in this frame.

This frame may be expanded to include additional slots as other system parameters become relevant on future Newton systems.

GetAppPrefs

`GetAppPrefs(`*appSymbol*`,` *defaultFrame*`)`

Retrieves the preferences for an application from the system soup.

appSymbol	A symbol identifying the application whose preferences you want.
defaultFrame	The default frame to use for the application preferences.

This function returns a system soup entry that is the application preferences entry.

The *appSymbol* is stored in the `tag` slot of the entry. If no entry exists for the specified *appSymbol*, a new entry is created, filled in with the contents of the *defaultFrame*, entered into the system soup, and the entry is returned. If *defaultFrame* does not have a `tag` slot, the *appSymbol* is turned into a string and entered into the `tag` slot. The `tag` slot must contain the symbol name of the `appSymbol`; otherwise the function will not work.

GetMemoryItems

`GetMemoryItems(`*memSymbol*`)`

Returns an array of the memory items, suitable for use in a picker, that have been added under *memSymbol*.

memSymbol A symbol used in your memory slot.

For example:

```
self.currentMem := GetMemoryItems('|wiggys:Wiggy:PIEDTS|) ;
   if currentMem AND Length(currentMem) > 0 then
   :PopupMenu(currentMem, nil);
```

GetMemorySlot

`GetMemorySlot(`*memorySlot*`, `*op*`)`

Removes the storage used for the memorized items. You should call this function from the `DeletionScript` function for your application.

memorySlot A symbol that identifies a group of memorized items.

op The symbol, `'remove`.

For example:

```
GetMemorySlot('|wiggys:Wiggy:PIEDTS|, 'remove);
```

Note
Other values of *op* and the return value for this function are undefined and subject to change. ◆

GetPrinterName

`GetPrinterName(`*printerFrame*`) //platform file function`

Retrieves the name of the printer, given a printer frame object.

IMPORTANT

This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

`call kGetPrinterNameFunc with (`*printerFrame*`);`

▲

printerFrame A printer frame object. The only valid way to obtain a printer frame object is to retrieve it from the system user configuration variables with the `GetUserConfig` function. Do not try to construct the slots of this frame yourself because different types of printer drivers require different slots.

This function returns a string representing the name of the printer associated with *printerFrame*.

Here is an example of some code that retrieves the name of the current printer:

```
printerFrame := GetUserConfig('currentPrinter);
thePrinterName := call kGetPrinterNameFunc with
(printerFrame);
```

MakePhone

`MakePhone (`*phoneFrame*`)`

Takes a phone frame and creates a phone string with the different parts encoded. These phone strings are used in Names and in the rest of the system. To display one of these real phone strings, call `MakeDisplayPhone`.

phoneFrame A frame with optional slots: `areacode`, `phone`, `extension`.

For example, if you call

```
ParsePhone(MakePhone({areacode: "617", phone:
   "965-4322"}))
```

you get

```
back {areacode: "617", phone: "965-4322"}
```

MakeDisplayPhone

`MakeDisplayPhone(`*phoneStr*`)`

Takes a phone string or phone frame (a frame with slots `areaCode`, `phone`, and `extension`), and formats it using the current `phoneFormat` to return another string that is suitable for displaying to the user. The current phoneFormat can be accessed by `GetUserConfig('phoneFormat)` and is a formatting string. For example, `^0/^1x^2`, displays your phone number as `408/555-1212x111`. The default is to display numbers as `408/555-1212x111`.

phoneStr	A phone string or phone frame (a frame with slots `areacode`, `phone`, and `extension`).

MungePhone

rootView`:MungePhone(`*inNum*`,` *country*`)`

Finds the correct dialing sequence to use in your current location. It is a method of the root view which builds from *inNum*, a dialing string based on current user-configuration settings. The returned string prefixes *inNum* with an international access code, country code, and area code as appropriate.

inNum	The phone number to convert. It should generally be of the form *<area-code>* *<phone-number>*. For example: 415 555 1212 —a phone number in San Francisco, USA 81 555 1212— phone number in London, UK

country A string that is the name of the country in which to dial *inNum*. For example:

"USA" for "415 555 1212"

"UK" for "81 555 1212"

Note that the country must be one from the `ROM_countries` frame, discussed below.

If you give `nil` instead of a string, `MungePhone` assumes that the call is within the country specified in the Country setting in the user's personal preferences.

If the user stores a calling card number with *inNum* in the calling options in the Names application, the calling card number is appended to the phone dialing string.

`ROM_countries` is a frame with a slot for each country for which Newton can convert phone numbers. Each slot is a frame, though the only slot you care about is the `name` slot.

You can get an array of the names of all known countries as follows:

```
local countryNames := foreach item in ROM_countries
                                         collect item.name ;
```

`CallOptionsSlip` is the system slip for getting and setting the user call options. This includes the current area code, dialing prefix, long distance access code, and user calling card numbers.

`MungePhone` automatically checks these user configuration items, so they will be figured into the return string.

ParsePhone

`ParsePhone(`*phoneStr*`)`

Takes a phone string and parses it into a frame with the slots `'areacode`, `'phone`, and `'extension`. The slots may be `nil` if there's no corresponding string. You should call `ParsePhone` if you want to get the component parts of the phone number. (If you just want to display the phone number, call `MakeDisplayPhone` instead.)

phoneStr	Phone string to parse into a frame.

PowerOff

`PowerOff(` *reason* `)`

Causes the Newton device to power off. Before powering off, the system calls each function registered with the `RegPowerOff` global function.

reason — The reason for the power off operation, as indicated by one of the following symbols:

`'user`	Used if the user chooses to power off the device through some user interface element.
`'idle`	Used if the Newton has been idle for a period of time.
`'because`	Used when the power-off occurs for any other reason

Ord

`Ord (`*char*`)`

Converts a character to its Unicode decimal integer equivalent.

char	A character.

Here is an example:

```
ord($A)
65
```

RegEmailSystem

`RegEmailSystem(`*classSymbol*`,` *name*`,` *internet*`) // platform file function`

Registers a new type of e-mail system.

IMPORTANT

This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

`call kRegEmailSystemFunc with (`*classSymbol*`,` *name*`,` *internet*`);`

▲

classSymbol — A symbol identifying the class of the e-mail system. This symbol must be a subclass of `'|string.email|` and should include your registered signature. For example:

`'|string.email.prolocommsMailSystem|`.

name — A string that is the name of the e-mail system. This name shows 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 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 is used to convert an e-mail address on this system to an Internet address. The function is 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 is copied into memory by `EnsureInternal`, so take care to ensure that the application that registers the e-mail service can be removed without causing errors.

To unregister an e-mail system registered by `RegEmailSystem`, use the function `UnRegEmailSystem`.

RegPagerType

`RegPagerType(`*classSymbol*`,` *name*`) // platform file function`

Registers a new pager type.

IMPORTANT
This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

`call kRegPagerTypeFunc with (`*classSymbol*`,` *name*`);`

▲

classSymbol A symbol identifying the class of the pager type. This symbol must be a subclass of `'|string.pager|` and should include your registered signature. For example: `'|string.pager.prolocomm:SIG|`.

pagerText A string that is the name of the pager system; for example, "ProLo's Paging System." This name shows up in the pickers listing pager types throughout the system, so it should be short.

Note that none of the arguments to this function is copied into memory by `EnsureInternal`, so take care to ensure that the application that registers the pager types can be removed without causing errors.

To unregister a pager type registered by `RegPagerType`, use the function `UnRegPagerType`.

RegPhoneType

`RegPhoneType(`*classSymbol*`,` *name*`,` *char*`) // platform file function`

Registers a new phone type.

IMPORTANT
This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

`call kRegPhoneTypeFunc with (`*classSymbol*`, `*name*`,`
char`);`

▲

classSymbol	A symbol identifying the class of the phone type. This symbol must be a subclass of `'|string.phone|`, and should include your registered signature. For example: `'|string.phone.prololine:SIG|`.
name	A string that is the name of the phone type. This name shows up in the pickers listing phone types throughout the system, so it should be short.
char	A single character that the application will use to display with the phone number in overview or card views. An example of a valid character is `$C`.

Note that none of the arguments to this function is copied into memory by `EnsureInternal`, so take care to ensure that the application that registers the phone types can be removed without causing errors.

To unregister a phone type registered by `RegPhoneType`, use the function `UnRegPhoneType`.

ShowManual

```
ShowManual()
```

Opens the system-supplied help browser; this has the same effect as tapping "How Do I" in the Assist Drawer.

Sleep

`Sleep(`*ticks*`)`

Puts the Newton to sleep (suspends processing) for the number of ticks given.

ticks An integer giving the number of ticks to sleep. A tick is one-sixtieth of a second.

For example:

```
for i:= 5 to 1 by -1 do
begin
   SetValue(infoView, 'text,
                  "Will restart in " & i & " seconds!");
   RefreshViews();
   Sleep(60);
end;
```

If you leave out the `RefreshViews` call, the view is not updated until after the last iteration, because calling sleep postpones the view event handling.

Notice that the screen is not updated during `sleep`, even if there is a pending update.

IMPORTANT

Do not use the `Sleep` function to put the Newton to sleep for very long. The `Sleep` function suspends the Application task, in which all NewtonScript code runs, so the user can do nothing else during a sleep. Occasionally, particularly during communications, you may need to sleep for several seconds, even half a minute, but, in general, sleeping for more than a fraction of a second is too much. If you need a longer delay, consider `AddDeferredCall` (or a related function) or a `ViewIdleScript` method as alternatives. Those methods return control to the main event loop. ▲

SysBeep

rootView`:SysBeep()`

Plays the system beep sound. This message must be sent to the root view. For example:

```
:SysBeep();
```

UnRegEmailSystem

`UnRegEmailSystem(`*classSymbol*`) // platform file function`

Unregisters an e-mail system registered by `RegEmailSystem`.

IMPORTANT
This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

`call kUnRegEmailSystemFunc with (`*classSymbol*`);`

▲

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.

UnRegPagerType

`UnRegPagerType(`*classSymbol*`) // platform file function`

Removes the registration of a pager class added with `RegPagerType`.

IMPORTANT

This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

```
call kUnRegPagerTypeFunc with (classSymbol);
```

▲

classSymbol A symbol identifying the class of the pager type. This symbol must be a subclass of `'|string.pager|` and should include your registered signature. For example: `'|string.pager.prolocomm:PIEDTS|`.

Note that this function can't be used to unregister pager types that are built-in.

UnRegPhoneType

`UnRegPhoneType(`*classSymbol*`) // platform file function`

Unregisters a phone type added with `RegPhoneType`.

IMPORTANT

This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

```
call kUnRegPhoneTypeFunc with (classSymbol);
```

▲

classSymbol A symbol identifying the class of the phone type. This symbol must be a subclass of `'|string.phone|`, and should include your registered signature. For example: `'|string.phone.prololine:SIG|`.

Error Codes

This appendix lists the exceptions and error codes that the Newton system software generates. These are grouped into the following categories:

- system exceptions
- system errors
- hardware errors
- communications errors
- system services errors
- NewtonScript environment errors
- device driver errors
- other errors

Each of the categories is subdivided into several tables of related error codes to make it easier to find an error. All errors in this appendix are listed in ascending numeric order.

System Exceptions

These are the two main types of exceptions that can be raised by the Newton system software.

Exception symbol	Description
`|evt.ex.fr|`	NewtonScript environment exception
`|evt.ex.comm|`	Communications toolbox exception

System Errors

This section lists the different kinds of Newton system software errors.

Common Errors

These are errors that can occur at almost any time.

Error code	Description
0	No error
–7000	Not enough memory available

Application Errors

These are the application errors.

Error code	Description
–8001	PCMCIA card battery must be replaced
–8002	PCMCIA card battery is running low
–8003	Nothing to undo
–8004	The routing slip is already open
–8005	Close box must be tapped to hang up the modem
–8006	Nothing to print
–8007	Exception not handled
–8008	The length of a styles slot had to be extended
–8009	A length in the read-only styles slot is too short to display the text
–8010	Communications card has been inserted
–8011	Note has too many items
–8012	Note is too large
–8013	Note is too long
–8100	Blank note could not be created
–8101	Item could not be moved

Error code	Description
–8102	Changes could not be saved
–8103	A problem has occurred
–8104	Problem with the PCMCIA card
–8105	Note could not be changed

I/O Box Errors

These are the I/O Box errors.

Error code	Description
–8301	Missing transport
–8302	Missing slip
–8303	Cannot convert

View System Errors

These are the view system errors.

Error code	Description
–8501	Could not create view
–8502	Missing class slot
–8503	Unknown view stationery
–8504	Missing view flags
–8505	Missing view bounds

State Machine Errors

These are the state machine errors.

Error code	Description
–8601	Invalid state
–8602	No state
–8603	No wait state
–8604	No polling routine
–8605	Polling timed out
–8606	Aborted
–8607	No reentrance
–8608	Invalid mode

Operating System Errors

These are the operating system errors.

Error code	Description
–10000	Bad domain object ID
–10001	Bad physical page object ID
–10002	Unexpected object type
–10003	No page table
–10004	Allocation on an uninitialized heap
–10005	Call not implemented
–10006	Bad parameters
–10007	Not enough memory
–10008	Item not found
–10009	Could not create object
–10010	Must use a remote procedure call
–10011	Bad object
–10012	Not a user call

Error code	Description
–10013	Task does not exist
–10014	Unexpected end of message
–10015	Bad object ID
–10016	Bad message object ID
–10017	Message already posted
–10018	Cannot cash token
–10019	Port no longer exists
–10020	No message waiting
–10021	Communications problem (message timed out)
–10022	Bad semaphore group ID
–10023	Bad semaphore operation list ID
–10024	Semaphore group no longer exists
–10025	Semaphore would cause blocking
–10026	Task no longer exists
–10027	Task aborted
–10028	Cannot suspend blocked task
–10029	Bad register number
–10030	Bad monitor function
–10031	No such monitor
–10032	Not a monitor
–10033	Size too large in shared memory call
–10034	Shared memory mode violation
–10035	Object not owned by task
–10036	Object not assigned to task
–10037	Total confusion
–10038	Another task already blocking
–10039	Cancelled

Error code	Description
–10040	Object already initialized
–10041	Nested collection
–10042	Shared memory message no longer exists
–10043	Receiver did not perform remote procedure call
–10044	Copy aborted
–10045	Bad signature
–10046	Call not in progress
–10047	Token expected
–10048	Receiver object no longer exists
–10049	Monitor is not suspended
–10050	Not a fault monitor
–10051	No available page
–10052	Interrupt not enabled
–10053	Interrupt not implemented
–10054	Tric interrupt not enabled
–10055	Tric interrupt not implemented
–10056	Unresolved fault
–10057	Call already in progress
–10058	Offset beyond data
–10059	Bus access
–10060	Access permission
–10061	Permission violation
–10062	Duplicate object
–10063	Ill formed domain
–10064	Out of domains
–10065	Write protected
–10066	Timer expired

Error code	Description
–10067	Not registered
–10068	Already registered
–10069	System restarted due to a power fault
–10070	System restarted because the battery was dead
–10072	System restarted because a PCMCIA card was removed while in use.
–10073	RAM table is full
–10074	Unable to satisfy request
–10075	System error
–10076	System failure
–10077	New system software
–10078	Resource is claimed
–10079	Resource is unclaimed

Stack Errors

These are the stack errors.

Error code	Description
–10200	Stack too small
–10201	No room for heap
–10202	Stack is corrupted
–10203	Stack overflow
–10204	Stack underflow
–10205	Address out of range
–10206	Bad domain

Package Errors

These are the package errors.

Error code	Description
–10401	Bad package
–10402	Package already exists
–10403	Bad package version
–10404	Unexpected end of package
–10405	Unexpected end of package part
–10406	Part type is already registered
–10407	Part type is not registered
–10408	No such package exists
–10409	Newer package already exists
–10410	Newer version of application already installed

Newton Hardware Errors

This section lists the different kinds of Newton hardware errors.

PCMCIA Card Errors

These are the PCMCIA card errors.

Error code	Description
–10501	Unrecognized card
–10502	Card not ready
–10503	Bad power on card
–10504	Unexpected card error
–10505	Card reset
–10506	Card is not initialized
–10507	Card service is not installed

Error code	Description
–10508	Card service is not suspended
–10509	Card service has not been resumed
–10510	No usable configurations on card
–10511	Card could not be formatted
–10512	Card could not be formatted because it is write-protected
–10520	Bad CIS parser procedure pointer
–10521	Unknown tuple in CIS
–10522	Unknown subtuple in CIS
–10523	CIS tuple order is bad
–10524	CIS tuple size is bad
–10525	CIS tuple specified as no link has a link
–10526	CIS tuple specified with a link has no link
–10527	CIS tuple link target is bad
–10528	Bad CIS tuple version 1
–10529	Bad CIS tuple version 2
–10530	Bad CIS JEDEC tuple
–10531	Bad CIS checksum
–10532	Missing CIS
–10533	Blank CIS
–10534	Bad CIS
–10535	Bad link target

Flash Card Errors

These are the flash card errors.

Error code	Description
–10551	Flash card is busy
–10552	Flash card is not erasing
–10553	Flash card erase is not suspended
–10554	Flash card suspend erase error
–10555	Flash card erase failed
–10556	Flash card write failed
–10557	Flash card Vpp is low
–10558	Flash card error in sleep
–10559	Flash card does not have enough power

Card Store Errors

These are the card store errors.

Error code	Description
–10600	Attempt to read or write outside of object bounds
–10601	Bad buffer pointer
–10602	Bad card access
–10603	Bad storage type
–10604	Store not found
–10605	The store has been write-protected by the user
–10606	Object not found
–10607	Flash card block is full
–10608	Flash card is not virgin
–10609	Write error (one or more bits failed to assert)
–10610	No more objects
–10611	Flash card erase in progress

Error code	Description
–10612	Card is full
–10613	No more blocks left in search on flash card
–10614	Flash card log is full
–10615	Card needs to be formatted
–10616	Bad or unknown PSSID
–10617	Card memory is full
–10618	Missing or low battery on SRAM card
–10619	Attempt to modify store without a transaction in effect
–10620	Transaction aborted
–10621	Card needs recovery, but it is write-protected
–10622	Object too large for store

DMA Errors

These are the DMA errors.

Error code	Description
–10800	DMA mode
–10801	DMA bus access
–10802	DMA buffer doesn't exist
–10803	DMA address word alignment
–10804	DMA count word alignment
–10805	DMA count size
–10806	DMA offset size
–10820	DMA PCMCIA ready
–10821	DMA PCMCIA input acknowledgment
–10822	DMA PCMCIA write protect
–10823	DMA PCMCIA time out

Heap Errors

These are the heap errors.

Error code	Description
–10900	Heap odd block size
–10901	Heap block out of range
–10902	Heap preferred free not found
–10903	Heap free accounting error
–10904	Heap accounting error
–10905	Heap block too big
–10906	Heap bad prior pointer
–10907	Heap bad last pointer in prior
–10908	Heap bad last pointer in last

Communications Errors

This section lists the different kinds of Newton communications errors.

Generic AppleTalk Errors

These are the generic AppleTalk errors.

Error code	Description
–12001	Buffer too small or corrupted
–12002	Event is pending
–12003	Cancelled
–12004	Attempt to cancel failed
–12005	No handler for cancel
–12006	Unknown message receiver
–12007	Cannot create AppleTalk port
–12008	Cannot create AppleTalk task

Error code	Description
–12009	Not implemented
–12010	Data length error
–12011	No such subject available to open
–12012	Not opened
–12014	AppleTalk is already open
–12015	Duration is too small
–12016	Duration is too large

LAP Protocol Errors

These are the LAP protocol errors.

Error code	Description
–12100	LAP read link failed
–12101	LAP all protocols in use
–12102	No protocol handler
–12103	No such command
–12104	Bad link

DDP Protocol Errors

These are the DDP protocol errors.

Error code	Description
–12200	No such DDP command
–12201	Invalid socket
–12202	Not in static socket range
–12203	Not in dynamic socket range
–12204	Socket is already open
–12205	Socket not open

Error code	Description
–12206	Socket internal socket
–12207	Socket is in use
–12208	Unknown LAP type
–12209	DDP back check sum
–12210	Bad packet size
–12211	No listener for socket
–12212	No such protocol type known
–12213	External client timed out

NBP Protocol Errors

These are the NBP protocol errors.

Error code	Description
–12300	Bad form
–12301	Name is already registered
–12302	Too many names
–12303	Name is not registered
–12304	Too many names requested
–12305	Too many lookups are pending
–12306	Not a NBP packet DDP type
–12307	Unknown NBP function
–12308	Unknown NBP lookup reply
–12309	Too many tuples in lookup request
–12311	NBP index out of range
–12312	NBP lookup aborted
–12313	No such command
–12314	No names found

AEP Protocol Errors

These are the AEP protocol errors.

Error code	Description
–12400	No such command
–12401	Not an echo packet DDP type
–12402	AEP packet size is zero
–12403	AEP function not requested

RTMP Protocol Errors

These are the RTMP protocol errors.

Error code	Description
–12500	No such command
–12502	Packet size is zero
–12503	RTMP routed
–12504	RTMP address unresolved
–12505	RTMP no router available

ATP Protocol Errors

These are the ATP protocol errors.

Error code	Description
–12600	No such command
–12601	No ATP packet DDP type
–12602	Unknown ATP function
–12603	ATP request data length is zero
–12604	Expected responses are out of range
–12605	Response buffer is too small
–12606	ATP retry duration too small

Error code	Description
–12607	ATP transaction timed out
–12608	Responding socket already open
–12609	Responding socket not open
–12610	Response packet length bad
–12611	Bad number of response packets
–12612	Socket already has a request on autorequest

PAP Protocol Errors

These are the PAP protocol errors.

Error code	Description
–12700	No such command
–12701	Unexpected connection ID
–12702	Invalid connection ID
–12703	Invalid responder socket
–12704	Unexpected function
–12705	Printer is busy
–12706	Unexpected connection open result
–12707	Bad flow quantum requested
–12708	Connection timed out
–12709	EOF sent
–12710	PAP flushed
–12711	Printer terminated connection
–12712	Printer not found
–12713	No status available
–12714	No data available
–12715	The buffer that was passed is too small
–12716	Put data operation timed out

ZIP Protocol Errors

These are the ZIP protocol errors.

Error code	Description
–12800	No zones

ADSP Protocol Errors

These are the ADSP protocol errors.

Error code	Description
–12900	Too many ADSP connections
–12901	ADSP mode invalid
–12902	ADSP packet size bad
–12903	ADSP control type bad
–12904	Remote end disconnected

Utility Class Errors

These are the utility class errors.

Error code	Description
–14001	Not implemented
–14002	Out of memory
–14003	Bad position
–14004	Already initialized
–14005	Invalid size
–14006	Overflow
–14007	Underflow
–14008	Range check failed
–14009	Element sizes do not match
–14010	Not initialized
–14011	Pointer is `nil`

Communications Tool Errors

These are the communications tool errors.

Error code	Description
–16001	Command in progress
–16002	Bad communication tool command
–16003	Tool already has maximum requests pending
–16004	Buffer overflow
–16005	Request canceled or connection disconnected
–16006	Bad parameter in request
–16007	Connection end has not been created yet
–16008	Invalid call when connected
–16009	Phone connection was cut off, or invalid call when not connected
–16010	Connection negotiation failed because remote end is not compatible with local end configuration
–16011	Connection terminated or failed due to retransmission limit of data or connect packet
–16012	No data available for `TCommToolGetRequest` when `fNonBlocking` is true.
–16013	Request canceled or connection disconnected
–16014	Call not supported by tool
–16015	Request not pending
–16016	Event not pending
–16017	Time-out waiting for connection
–16018	Connection end is already bound
–16019	Connection end was not bound before use
–16020	Connection end is being released
–16021	No phone number was provided

Error code	Description
–16022	Operation failed because a resource was not available
–16023	Call failed because the option passed is not supported
–16024	The method is not implemented

Serial Tool Errors

These are the serial tool errors.

Error code	Description
–18000	Serial channel is in use
–18001	Memory error
–18002	Not current owner of the serial port
–18003	Framing or parity overrun, or bad connection
–18004	CRC error on input framing
–18005	An internal error has occurred
–18006	Packet size too large or too small in an output request
–18007	Unexpected packet length
–18008	EOF not found
–18009	Overrun bit was set
–18010	Too many collisions when sending packet
–18011	Too many deferrals when sending packet
–18012	Timed out waiting for an event
–18013	Serial tool is not active or ready

MNP Tool Errors

These are the MNP tool errors.

Error code	Description
–20001	Connection parameter negotiation failed
–20002	Acceptor of connect request timed out
–20003	Not connected
–20004	Request aborted by disconnect request
–20005	Link attention service is not enabled
–20006	Request retry limit of connect initiator reached
–20007	Command already in progress
–20008	Connection already established
–20009	Connection failed due to incompatible protocol levels
–20010	Connection handshake failed
–20011	Memory for MNP not allocated

FAX Tool Errors

These are the FAX tool errors.

Error code	Description
–22001	Lost connection while sending or receiving FAX
–22002	FAX machine is not compatible
–22003	Transmission error
–22005	FAX machine had a problem sending some pages
–22006	Transmission error
–22007	Transmission error

Modem Tool Errors

These are the modem tool errors.

Error code	Description
–24000	No modem is connected
–24001	There is no dial tone
–24002	There is no answer
–24003	The phone number is busy
–24004	There is no answer
–24005	The modem is not responding properly
–24006	FAX carrier error
–24007	The modem is not responding properly
–24008	The modem connected to the serial port does not support cellular connection
–24009	The AT+FRH command timed out when receiving flags

Communications Manager Errors

These are the Communications Manager errors.

Error code	Description
–26000	Service already initialized
–26001	Unknown command
–26002	Unknown service
–26003	Service already exists
–26004	No service specified in the options array
–26005	There is no registered service matching the type specified in the options array
–26006	No endpoint exists; this is usually because `CMStartService` has not been called
–26007	No public port exists; this is usually because `CMGetEndPoint` has not been called

Error code	Description
–26008	No known last connected device
–26009	A tuple has been received, but no the device ID tuple
–26010	A service information response tuple was expected
–26011	Unsupported service; can only load packages
–26012	An SCP load is in progress and another cannot be issued
–26013	The SCP load call is not supported on this machine
–26014	Cannot process this speed
–26015	The SCP loader did not previously load a package

Docker Errors

These are the docker errors.

Error code	Description
–28001	Invalid store signature
–28002	Invalid entry
–28003	Aborted
–28004	Invalid query
–28005	Read entry error
–28006	Invalid current soup
–28007	Invalid command length
–28008	Entry not found
–28009	Bad connection
–28010	File not found
–28011	Incompatible protocol
–28012	Protocol error
–28013	Docking canceled
–28014	Store not found
–28015	Soup not found

Error code	Description
–28016	Invalid header
–28017	Out of memory
–28018	Newton version too new
–28019	Package cannot load
–28020	Protocol already registered
–28021	Remote import error
–28022	Bad password error
–28023	Password retry
–28024	Idle too long
–28025	Out of power
–28026	Invalid cursor
–28027	Already busy
–28028	Desktop error
–28029	Cannot connect to modem
–28030	Disconnected
–28031	Access denied
–28100	Disconnect during read
–28101	Read failed
–28102	Communications tool not found
–28103	Invalid modem tool version
–28104	Card not installed
–28105	Browser File Not Found
–28106	Browser Volume Not Found
–28107	Browser Path Not Found

Docker Import and Export Errors

These are the docker import and export errors.

Error code	Description
–28200	Syntax error
–28201	Invalid version
–28202	Could not open temporary store
–28203	Could not convert
–28204	Invalid criteria
–28205	Error applying script
–28206	Missing meta data
–28207	Unknown error
–28208	Scanner overflow error
–28209	Data Viz translator error
–28210	Invalid type

Docker Disk Errors

These are the docker disk errors.

Error code	Description
–28300	Disk full
–28301	File not found
–28302	File is write protected
–28303	Duplicate file name
–28304	Too many files open

Docker Desktop DIL Errors

These are the docker desktop DIL errors.

Error code	Description
–28700	No Error
–28701	Out of memory
–28702	Invalid pipe state
–28703	Exception error
–28704	Queue full
–28705	Pipe not initialized
–28706	Invalid parameter
–28707	Pipe not ready
–28800	No Error
–28801	Out of object heap memory
–28802	Out of temporary memory
–28803	Unknown slot
–28804	Slot size exceeded
–28805	Slot size required

System Services Errors

This section lists the different kinds of Newton system services errors.

Sound Errors

These are the sound errors.

Error code	Description
–30000	Generic sound error
–30001	Not enough memory available
–30002	Invalid message

Error code	Description
–30003	Sound was not played
–30004	No channel decompressor
–30005	Destination buffer too small
–30006	Sound player busy
–30007	Sound recorder busy
–30008	No samples provided
–30009	Unsupported sound configuration
–30010	Sound channel closed
–30011	Sound cancelled
–30012	The sound volume is set to zero

Compression Errors

These are the compression errors.

Error code	Description
–32001	Cannot compress in place
–32002	Parsing error
–32003	Invalid type
–32004	Compression not achieved
–32005	Key not found
–32006	Compression index error
–32007	Cannot decompress in place
–32008	Decompression not achieved
–32009	Unexpected end of source
–32100	Buffer overflow
–32101	Buffer underflow

Memory Errors

These are the memory errors.

Error code	Description
-34000	Not free, direct or indirect
-34001	Pointer not aligned to 4-byte boundary
-34002	Pointer to outside of heap
-34003	Unknown infrastructure type
-34004	Free block where there shouldn't be one
-34005	Free list pointer points outside of heap
-34006	Free-list pointer doesn’t point at a free block
-34007	Invalid block size
-34008	Forbidden bits set in block size
-34009	Less than minimum size for heap block
-34010	Heap block tool large
-34011	Total free space is more than space for entire heap
-34012	Nil pointer where not allowed
-34013	Actual free space does not match tracked free space
-34014	Linked free space does not match tracked free space
-34015	Master pointer doesn’t point back to a handle block
-34016	Invalid block size adjustment
-34017	Internal block may be mangled
-34018	The heap is invalid
-34019	Caught an exception while checking the heap
-34020	Invalid heap header

Communications Transport Errors

These are the communications transport errors.

Error code	Description
–36001	Incorrect address format
–36002	Incorrect option format
–36003	Cancel is in progress
–36005	Could not allocated address
–36006	Operation not supported in the current tool state
–36008	System error
–36012	Flow control problem
–36018	Unsupported primitive
–36019	State change is in process
–36030	There's already a synchronous call pending

Sharp IR Errors

These are the Sharp infrared errors.

Error code	Description
–38001	No response - protocol time out
–38002	Cancelled - remote side cancelled operation
–38003	Protocol error
–38004	Data checksum failed
–38005	Remote side receive failed
–38006	Bad connection - allowed number of retries exceeded
–38007	SCC data errors on receive
–38008	Unspecified beaming error

Online Service Errors

These are the online service errors.

Error code	Description
–40102	Lost connection to host
–40103	Lost connection to host
–40104	The host is not responding
–40105	There is a problem reading from the host
–40106	Failed to connect to local access number

Printing Errors

These are the printing errors.

Error code	Description
–44000	Printer problem
–44001	Newton is unable to print
–44002	No printer is connected
–44003	Printer busy
–44004	Printing stopped
–44005	Lost contact with the printer
–44006	Image too complex for printer
–44100	The next sheet of paper must be inserted
–44101	The phone number must be dialed now
–44102	There is no paper tray
–44103	The wrong paper tray is attached
–44104	The printer has no paper
–44105	The printer has no ink
–44106	The printer is jammed
–44107	The printer door is open
–44108	The printer is off-line

Newton Connection Errors

These are the Newton connection errors.

Error code	Description
–46001	Connection initialization failed
–46002	Timer error
–46003	Connection request was denied by the remote
–46004	Unable to connect because there are no endpoints available
–46005	A connect request was received but no service name was given

NewtonScript Environment Errors

This section lists the different kinds of NewtonScript error codes.

Store and Soup Errors

These errors are related to stores and soups.

Error code	Description
–48001	The PCMCIA card is not a data storage card
–48002	Store format is too old to understand
–48003	Store format is too new to understand
–48004	Store is corrupted, can't recover
–48005	Single object is corrupted, can't recover
–48006	Object stream has unknown format version
–48007	Fault block is invalid
–48008	Not a fault block
–48009	Not a soup entry
–48010	Tried to remove a store that was not registered
–48011	Soup index has an unknown type
–48012	Soup index has an unknown key structure

Error code	Description
–48013	Soup index does not exist
–48014	A soup with this name already exists
–48015	Tried to `CopyEntries` to a union soup
–48016	Soup is invalid (probably from a removed store)
–48017	Soup is invalid (probably from a removed store)
–48018	Entry is invalid (probably from a removed store)
–48019	Key does not have the type specified in the index
–48020	Store is in ROM
–48021	Soup already has an index with this path
–48022	Internal error–something unexpected happened
–48023	Tried to call `RemoveIndex` on the `_uniqueID` index
–48024	Query type missing or unknown
–48025	Discovered index inconsistency
–48026	Maximum number of soup tags reached
–48027	Soup does not have a tags index
–48028	Invalid tags specification in the query
–48029	Store cannot handle the feature (for example, large objects)
–48030	Unknown sorting table
–48031	Cannot do union soup because of different sorting tables
–48032	Invalid index description
–48033	Cannot use virtual objects for soup entry keys

Object System Errors

These errors are related to the object system.

Error code	Description
–48200	Expected a frame, array, or binary object
–48201	Invalid magic pointer
–48202	Empty path
–48203	Invalid segment in path expression
–48204	Path failed
–48205	Index out of bounds (string or array)
–48206	Source and destination must be different objects
–48207	Long out of range
–48210	Bad arguments
–48211	String too big
–48212	Expected a frame, array, or binary object
–48213	Expected a frame, array, or binary object
–48214	Object is read-only
–48216	Out of heap memory
–48217	Invalid attempted use of magic pointer
–48218	Cannot create or change an object to negative size
–48219	Value out of range
–48220	Could not resize locked object
–48221	Reference to deactivated package
–48222	Exception is not a subexception of `|evt.ex|`

Bad Type Errors

These errors are caused by data of the wrong type.

Error code	Description
–48400	Expected a frame
–48401	Expected an array
–48402	Expected a string
–48403	Expected a frame, array, or binary object
–48404	Expected a number
–48405	Expected a real
–48406	Expected an integer
–48407	Expected a character
–48408	Expected a binary object
–48409	Expected a path expression (or a symbol or integer)
–48410	Expected a symbol
–48411	Expected a function
–48412	Expected a frame or an array
–48413	Expected an array or `nil`
–48414	Expected a string or `nil`
–48415	Expected a binary object or `nil`
–48416	Unexpected frame
–48417	Unexpected binary object
–48418	Unexpected immediate
–48419	Expected an array or string
–48420	Expected a virtual binary object
–48421	Expected a package
–48422	Expected `nil`
–48423	Expected `nil` or a symbol
–48424	Expected `nil` or `true`
–48425	Expected an integer or an array

Compiler Errors

These errors are generated by the compiler.

Error code	Description
–48600	Could not open a listener window
–48601	Syntax error
–48603	Cannot assign to a constant
–48604	Cannot test for subscript existence; use length
–48605	Global variables not allowed in applications
–48606	Cannot have a global variable and a global constant with the same name
–48607	Cannot redefine a constant
–48608	Cannot have a variable and a constant with the same name in the same scope
–48609	Non-literal expression for constant initializer
–48610	End of input inside a string
–48611	Odd number of digits between `\\u`'s
–48612	No escapes but `\\u` are allowed after `\\u`
–48613	Invalid hex character in `\\u` string
–48617	Two-digit hex number required after `$\\` escape
–48618	Four-digit hex number required after `$\\u`
–48619	Illegal character `'%c'`
–48620	Invalid hexadecimal integer: `%s` (out of range)
–48621	Invalid real number (out of range)
–48622	Invalid decimal integer: `%s` (out of range)
–48626	`#xxxx` not allowed from NTK
–48627	Not a constant
–48628	Decimal digit required after `@`

Interpreter Errors

These are interpreter errors.

Error code	Description
–48800	Not in a break loop
–48803	Wrong number of arguments
–48804	FOR loop BY expression has value zero
–48806	No current exception
–48807	Undefined variable
–48808	Undefined global function
–48809	Undefined method
–48810	No `_proto` for inherited send
–48811	Tried to access slot of `nil`
–48814	Local variables and FOR/WITH loops not allowed at top level
–48815	The operation would make the rich string invalid

Communications Endpoint Errors

These are the communications endpoint errors.

Error code	Description
–54000	An active input spec is required
–54001	Error in the `form` slot of an input spec
–54002	Trying to send zero-length data
–54003	An input spec is required
–54004	The option you tried to set was missing
–54005	Error in the `endSequence` slot of an input spec
–54006	Used the `Partial` method with a bad input spec, or unable to do a partial input
–54007	Error in `termination` slot of input spec

Error code	Description
–54008	Error in `target` slot of input spec
–54009	Error in `filter` slot of input spec
–54010	Attempted to receive binary data with no target object specified
–54011	Attempted to send or receive template data without a template specified
–54012	Tried to set an input spec when one was already active
–54013	Invalid value in filter proxy of input spec
–54014	Endpoint object is missing
–54015	Method not supported, or called inappropriately
–54016	The character specified in the filter proxy of the input spec is more than a single byte
–54021	Option failed
–54022	Option set, but set value is different from requested value
–54023	Set attempted on read-only option
–54024	Option not supported
–54025	Invalid option `opcode`
–54026	Option not found
–54027	One or more requested options missing

Device Driver Errors

This section lists the device driver error codes.

Tablet Driver Errors

These are the tablet errors.

Error code	Description
–56001	Attempted to call the tablet driver before it was loaded
–56002	Attempted to create a tablet driver a second time
–56003	Creation of tablet driver failed
–56004	Unable to enter bypass mode
–56005	Not in bypass mode
–56006	Cannot add sample to buffer
–56007	No new data since last polling time
–56008	Unsupported function
–56101	Timeout when calibrating
–56102	Calibration aborted

Battery Driver Errors

These are the battery driver errors

Error code	Description
–56201	Could not find battery driver
–56202	Battery error
–56203	Invalid battery selector

Other Services Errors

This section lists the error codes for other services.

Alien Store Errors

These are the alien store errors

Error code	Description
–58001	Oversize page
–58002	No such page
–58003	Cannot repage ID
–58004	No more for that page
–58005	Store is damaged

Index

A

B

C

E

F

G

H

I

J

K

L

M

O

P

S

T

U

V

W, X, Y

Z

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

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

LEAD WRITER
Christopher Bey

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

PROJECT LEADER
Christopher Bey

ILLUSTRATOR
Peggy Kunz

EDITORS
Linda Ackerman, David Schneider, Anne Szabla

PRODUCTION EDITOR
Rex Wolf

ELECTRONIC MEDIA TEAM
Gary Hillerson, Gerry Kane, Christopher Bey

PROJECT MANAGER
Gerry Kane

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