Newton OS 2.1
Engineering Documents

IMPORTANT

The information in this document is preliminary, unreviewed, and it is subject to change.

October 6, 1996 Newton Technical Publications Team

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

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

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

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

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

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

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

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

Contents

About This Book

This book describes changes and additions to the Newton operating system for version 2.1.

Important Note

The chapters in this book are at different stages of completion. Some are less complete than others. For all chapters, even the most complete ones, keep in mind that the information is preliminary, unreviewed, subject to change, and may not consistently reflect the latest technical information available. ◆

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 book is the definitive guide to Newton programming, covering Newton OS 2.0. It contains a companion volume, *Newton Programmer's Reference*, on CD-ROM in various electronic formats for quick access.
- *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://devworld.apple.com/dev/newtondev.shtml`) 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 xv.

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.
- `Code 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 Code typeface to distinguish them from regular body text. If you are programming, items that appear in Code typeface 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.

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://devworld.apple.com/dev/newtondev.shtml`

NewtonWorks API ERS

NewtonWorks is a new built-in application that is based on the NewtApp framework. It is shipped with two kinds of installed stationery: a word processor (based on the new protoTXView—see the protoTXView ERS), and a drawing application (see the Draw ERS).

In addition to the standard NewtApp methods, NewtonWorks has other methods and slots. These are listed in the section below.

NewtonWorks allows the registration of different types of dataDefs. These dataDefs appear in the New menu, and their corresponding default viewDefs control the status bar as well as the scroll bars and the viewable area. Each dataDef can have a number of print viewDefs, but only one viewDef that the user sees, the `'default` viewDef. The following sections describe the slots and methods that NewtonWorks expects to find in registered dataDefs and viewDefs.

NewtonWorks-Specific slots

Slot descriptions

`newtAppBase`	Contains the NewtonWorks base view.
`viewDefView`	Contains the current viewDef view. Can be `nil` if no viewDef is currently active, as in the case when the current layout is the overview.

The following method is also provided.

UpdateStatusBar

newtAppBase`:UpdateStatusBar()`

Can be called to recreate the status bar. This will call the `UpdateStatusBar` method of `viewDefView`, if there is a currently active viewDef.

The return value of this method is undefined and you should not rely on it.

DataDef

To register the dataDef, call `RegDataDef` with the symbol and the dataDef frame. As in the NewtApp framework, the symbol of the dataDef is used as the `class` slot of soup entries created by the dataDef.

Note that certain slots are required for all dataDefs, such as `symbol`, `name`, `description`, `superSymbol`, `icon`, `StringExtract`, `TextScript`, `FillNewEntry`, etc. These slots are not documented here unless their values are NewtonWorks-specific. The slots that NewtonWorks expects are as follows.

Slot descriptions

`superSymbol`	Must be the symbol 'newtWorks.

The following methods are also expected in the dataDef.

FindFn

dataDef:`FindFn(`*entry*`,` *what*`,` *offset*`)`

Called when an app-wide Find is performed. Return non-`nil` if the string *what* is found in the document anywhere after *offset*. Return `nil` if not.

Note that if the string is in the soup entry (if `FindStringInFrame` returns `true`) `FindFn` will not be called because NewtonWorks has already identified the entry as matching the Find criteria.

FindSoupExcerpt

ownerView:`FindSoupExcerpt(`*entry*`,` *finderFrame*`)`

Called when the user performs a Find operation, to return a title for the item. This is the standard 2.0 function `FindSoupExcerpt`.

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.

Info Preferences

The dataDef can specify extra preferences that appear in the info button when the user is viewing an entry of that dataDef type. To do so, the dataDef should have a slot called `prefs` which contains a frame with relevant information.

Slot descriptions

`prefs` A frame similar to the frames passed to `PopupMenu`, as documented in the section "Specifying the List of Items for a Popup" (page 6-37) in *Newton Programmer's Guide*. The standard slots in `prefs` (such as `item`, `icon`, etc.) define the appearance of the command in the info button. The frame must also contain a view template in the `prefsTemplate` slot.

`prefsTemplate` A view template for a slip that contains the user interface elements needed for the user to set the dataDef-specific prefs.

When the command is chosen from the info button, NewtonWorks will set the slots `target`, `newtAppBase`, and `viewDefView` appropriately, and then call `BuildContext`(*prefsTemplate*). The template must read the appropriate slots in its `ViewSetupFormScript` method, and write the slots in its `ViewQuitScript` method.

Note that the view must be closed by calling `newtAppBase:RememberedClose(`*base*`)`, or the unused view will use up RAM.

Infobox (Title slip) info

The dataDef can specify extra info that appears in the infobox when the user taps to bring up the title slip. To do so, the dataDef should have a function called `InfoBoxExtract`.

InfoBoxExtract

dataDef`:InfoBoxExtract(`*target*`,` *boxInfo*`,` *viewDefView*`)`

Called when the user brings up the title slip, to get the information to add to the title slip. Return a shape which is the extra info to add to the slip, or `nil` if no extra info.

target The current entry in NewtonWorks.

boxInfo A bounds frame where the text should be placed.

viewDefView The viewDef for the target.

•••Appears to be optional.•••

Viewdefs

To register viewDefs, call `RegisterViewDef` with the dataDef symbol and the viewDef frame. Note that certain slots are required for all viewDefs, such as `symbol`, `type`, `protection`, etc. These slots are not documented here unless their values are NewtonWorks-specific.

Printing requires the registration of a print format as documented in NPG 2.0. NewtonWorks does not expect any slots other than the ones required for all print formats.

For the main viewDef, the slots that NewtonWorks expects are explained in the following sections according to their function.

Find

This section describes the different methods supporting the various find operations. Note that these find operations are document-specific; i.e. they operate on the currently visible document.

The viewDef is responsible for putting the "Find" and "Find Again" commands into the appropriate button in its statusBar. To do this, add `{keyMessage: 'NewtworksFind}` and `{keyMessage: 'NewtworksFindAgain}` to the appropriate popup array. (`NewtworksFind` and `NewtworksFindAgain` are defined in NewtonWorks, so the behavior will be correct as long as these methods get called.)

NewtonWorks assumes that the viewDef defines a `FindChange` method. If this method is not defined, the viewDef should not add the "Find" and "Find Again" commands to any statusBar button, since Find will not operate within the viewDef.

FindChange

viewDef:`FindChange(`*action*`,` *data*`)`

Called when the user performs a Find operation from NewtonWorks Find/Change slip.

action	A symbol indicating the action the user requested: `'find`, `'change`, or `'changeAll`.
data	Varies depending on the value of *action*. See Table 2-1 for details.

Table 2-1 FindChange parameters and actions

action value	*data* value	FindChange method should do this
`'find`	A string to find.	Search for the next string that matches, starting from the current selection. The search should wrap if necessary. Return `true` if the string is found, `nil` if not.
`'change`	A frame with slots `findStr` (string to find) and `changeStr` (string to replace with).	Replace the current selection with `changeStr`. If `findStr` does not match the current selection, this is an error condition. Return `true` if the selection is replaced with `changeStr`, `nil` if not.
`'changeAll`	A frame with slots `findStr` (string to find) and `changeStr` (string to replace with).	Replace all instances of `findStr` with `changeStr`. Return the number of instances replaced as an integer.

ShowFoundItem

viewDef:`ShowFoundItem(`*foundItem*`,` *finderFrame*`)`

Called after the user performs a standard Newton Find operation (from the Find slip that's built into ROM), to show the item that was found. This is the same method documented in *Newton Programmer's Reference*.

There is no specified return value.

Storage

This section describes the methods which support saving and retrieving NewtonWorks documents.

The viewDef should call `StartFlush` when a change has been made. This will cause `SaveData` to be called at idle time, which gives the view a chance to save data to the soup. Even if `StartFlush` is never called, `SaveData` will be called after a longer idle period, so the view gets a periodic chance to save changes that aren't really data changes.

SaveData

viewDef:`SaveData(`*target*`)`

Called when the data is to be saved to the soup.

target The target entry to be saved.

If there is data to be saved, modify *target* to hold the data, and return `true`. If `nil` is returned, the soup entry may still be saved if it has otherwise been modified, e.g. by the NewtApp framework.

To save the data but not mark the entry as changed, for example if the hilite location needs to be saved, return the symbol `'noRealChange`, instead of `true`. This will tell NewtonWorks to save the changes to the soup entry, but not to update the modtime of the entry (as displayed in the title slip).

Scrolling

This section describes the methods necessary to support scrolling. Note that the vertical scrollbar is always visible in NewtonWorks, whereas the horizontal scroller does not appear if the document width doesn't require it. To remove the scrollers altogether, the viewDef should supply a `GetScrollableRect` method that returns `nil`. In this case, no other scroll methods are required.

ViewScrollUpScript

viewDef: `ViewScrollUpScript`

Called when the scroll-up arrow is tapped. If this method is not defined, the default behavior occurs, which is to scroll upwards by one screen in the document.

ViewScrollDownScript

viewDef: `ViewScrollDownScript`

Called when the scroll-down arrow is tapped. If this method is not defined, the default behavior occurs, which is to scroll downwards by one screen in the document.

Scroll

viewDef: `Scroll(`*scrollValues*`)`

Called to scroll the view content (viewBounds minus the ruler) horizontally and vertically as defined in *scrollValues*.

scrollValues	A frame with `x` and `y` slots, to scroll the content by x pixels horizontally and y pixels vertically.

The return value is not used.

Note that you must call `UpdateAllScrollers` from your `Scroll` method to update the corresponding scroller.

GetScrollValues

viewDef: `GetScrollValues()`

Called to get the current scroll values. Return a frame with `x` and `y` slots containing the vertical and horizontal scroll values, as integers.

GetTotalHeight

viewDef: `GetTotalHeight()`

Called to get the total height of the view. Return an integer.

GetTotalWidth

viewDef:`GetTotalWidth()`

Called to get the total width of the view. Return an integer.

GetScrollableRect

viewDef:`GetScrollableRect()`

Called to get a rectangle that describes the scrollable area for the view. Return a frame of integers (`top`, `left`, `bottom`, `right`) describing the rectangle enclosing the view. If `nil` is returned, NewtonWorks removes the scrollbars, and no other scroll methods are required.

UpdateAllScrollers

viewDef:`UpdateAllScrollers(`*view*, *totalHeightChanged*, *scrolledV*, *totalWidthChanged*, *scrolledH*`)`

Unlike the other methods, you call this method. This method is internal to NewtonWorks, and should be called when your view moves its origin, so that the scrollers can be updated appropriately.

view	The view (`self`).
totalHeightChanged	A Boolean value indicating whether or not the maximum value of the vertical scroller has changed.
scrolledV	A Boolean value indicating whether or not the vertical scroller thumb needs to be updated.
totalWidthChanged	A Boolean value indicating whether or not the maximum value of the horizontal scroller has changed.
scrolledH	A Boolean value indicating whether or not the horizontal scroller thumb needs to be updated.

Status Bar

This section describes the status bar information that NewtonWorks expects.

Slot descriptions

`statusLeftButtons`

An array of button frames to put in the left portion of the status bar, after the "New" button. The status bar will be updated with the buttons in this array whenever the viewDef appears. For more details on this array, see the `menuLeftButtons` slot in the `newtStatusBar` proto in *Newton Programmer's Reference* .

`statusRightButtons`

An array of button frames to put in the right portion of the status bar, before the routing and filing buttons. The status bar will be updated with the buttons in this array whenever the viewDef appears. For more details on this array, see the `menuRightButtons` slot in the `newtStatusBar` proto in *Newton Programmer's Reference* .

UpdateStatusBar

viewDef:`UpdateStatusBar()`

This message is sent to the view identified by the `viewDefView` slot, if NewtonWorks receives notification of an auxiliary button change. This can occur if a package installs or removes an auxiliary button for NewtonWorks. When this message is called, the view should update its `statusLeftButtons` and `statusRightButtons` slots. Note that this method is optional.

The return value is not used.

Help

This section describes optional help settings that NewtonWorks will observe.

Slot descriptions

`viewHelpTopic` (optional) A string, used as the string index for the help book to open to.

`helpManual` (optional) A frame defining the help manual to open. This is the same as NewtApplication's `helpManual` slot.

The following help method is also supported.

DoHelp

viewDef:`DoHelp(`*entry*`)`

If this method exists, it will be called when the user chooses Help from the Prefs menu when the viewDef is visible, instead of the default behavior of using the slots `viewHelpTopic` and `helpManual` to open help. If the default behavior is desired, return `'loadHelp` from this method.

Other

This section describes other slots and methods that NewtonWorks expects to find in the viewDef.

Slot descriptions

`symbol` Must be `'default` for the main viewDef.

PrefsChanged

viewDef:`PrefsChanged(`*prefsFrame*`)`

If this method is defined, it is called when the global preferences for NewtonWorks are changed.

prefsFrame A frame containing the following slots:

`metricUnits`
`True` if using centimeters, `nil` if inches.

`internalStore`
`True` if always storing items on the internal store.

ViewChangedScript

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

If this method is defined, it is called when the viewBounds of the view changes as the result of a `SetValue` call.

Registering Tools

This section describes how to add tools to the viewDefs that are installed in NewtonWorks. The individual viewDefs are responsible for displaying the tools installed for that viewDef, and for calling the tool correctly when it's chosen from the Tools popup.

Use the method `RegNewtWorksTool` to register a new tool. Call `UnRegNewtWorksTool` to unregister a tool. You can call `GetNewtWorksTools` to return an array of registered tools.

Within the viewDef or status bar context, you can use the syntax `newtAppBase:`*FunctionName* to call these methods. For example: `newtAppBase:GetNewtWorksTools`.

RegNewtWorksTool

`GetRoot().newtWorks:RegNewtWorksTool(`*toolSym*`,` *toolFrame*`)`

Registers a tool for the viewDef identified by `dataTypeSymbol` slot in the *toolFrame* frame. Returns `nil` if the tool was not successfully registered, non-`nil` if it was.

toolSym	A unique symbol under which to register the tool.
toolFrame	A frame describing the command to appear in the Tools popup. Each command is a frame similar to the frames passed to `PopupMenu`, as documented in *Newton Programmer's Guide*. These standard slots (such as `item`, `icon`, and all other slots supported by `PopupMenu`) define the appearance of the command in the viewDef's Tools button, but each viewDef defines the additional slots it expects to see in this frame. Every *toolFrame* must also contain the slot `dataTypeSymbol` that contains a symbol identifying the dataDef it's registered for. For example, tools for the word processor have the symbol `'paper`.

•••The Draw app may have its own registery. See the Draw ERS.•••

For `'drawPaper` (Draw) or `'paper` (Word Processor) stationery, provide the following additional slots and methods in *toolFrame*:

`CmdFunc(`*viewDefView*`, `*newtAppBase*`)`
: A method called when the command is chosen from the Tools button. *viewDefView* is the main viewDef. *newtAppBase* is the NewtonWorks application.

`keyCommand`
: (optional) A frame with command key information, containing the same slots as the frames used to register command keys, as documented in the Keyboard ERS. To register a command key that activates an item on the Tools button, the `keyCommand` frame must contain the function `KeyFn(`*receiverView*`)`. This function is called when the command key for this item is pressed. This function should perform the action as if the command were chosen from the Tools button. The return value of this function is not used.

UnregNewtWorksTool

`GetRoot().newtWorks:UnregNewtWorksTool(`*toolSym*`)`

Unregisters a tool registered by `RegNewtWorksTool`.

toolSym
: The symbol under which the tool was originally registered.

The return value of this method is undefined and you should not rely on it.

GetNewtWorksTools

`GetRoot().newtWorks:GetNewtWorksTools(`*dataTypeSym*`)`

Returns an array containing the tools identified by *dataTypeSym*. If no tools were found, an empty array is returned.

dataTypeSym
: The `dataTypeSymbol` to look for among the registered tools. This is used to match the `dataTypeSymbol` slot in the *toolFrame* specified when the tool was registered. To get all the tools registered for all data types, pass `nil` for this parameter.

NewtonWorks Word Processor Soup Format

NewtonWorks word processor soup entries have the following slots.

Slot descriptions

`class`	The symbol `'paper`
`version`	Integer, the current version of the entry
`title`	String which is the document title
`timeStamp`	Creation date of the entry
`realModTime`	Date the entry was most recently modified
`saveData`	Frame returned from `protoTXView Externalize` call (see protoTXView ERS)
`hiliteRange`	Frame with the document's hilite range (see protoTXView ERS)
`margins`	Frame with slots `top`, `left`, `bottom`, `right`, which are the document's margins in pixels.

CHAPTER 2

NewtonWorks Draw Application ERS

The NewtonWorks Draw application is built into the Apple eMate™ 300 and runs on any platform providing appropriate hardware (eMate 300 size screen) and software (Newton OS v. 2.1) support. Its features are similar to those of MacDraw v. 1.0 or ClarisWorks v. 2.0. Easy document exchange with desktop draw applications, especially those created in ClarisWorks, is a primary feature.

Overview

This application uses the NewtonScript `DrawShape` function as its drawing engine.

The Draw application is loosely integrated with the NewtonWorks word processor application. The user is able to copy and paste a drawing into a word processing document, but the word processor's ability to edit drawings is limited to the following operations:

- Drawings can be moved to Quick Sketch and scaled.

- Drawings can be moved, cut, copied, pasted, and deleted.
- Text from a text box can change font/size.

The design of the user interface for the Draw application tries to follow the following (sometimes conflicting) goals:

- Present an interface that is not dissonant with ClarisWorks.
  A typical scenario is that students will use the eMate 300 to make a draw document, and then upload the document to a Mac (or PC) and use a desktop drawing program (such as ClarisWorks) if they want to use more advanced features. So there should be some consistency of user interface between the eMate 300 and the desktop. (kmc)
- Provide Newton-like interfaces when appropriate.
  Since the eMate 300 is a Newton product, its Draw application should fit in with other applications on a Newton device. For example, the Draw application should understand gestures that "make sense" for drawings. Also, the pen can be a more convenient input device for drawing than a mouse, and the Draw application should take advantage of that. (kmc)
- Be usable on any Newton platform having an appropriate screen size.
  Because the Apple eMate 300 is a Newton product, any of its applications should in theory be usable on any other Newton product. In particular, the Draw application should be usable without a hardware keyboard and with handwriting recognition enabled.

Feature List

The initial release of the NewtonWorks Draw application provides the features described in this section.

Drawings have fixed width/height/margins and cannot be larger than a single page.

Creating Graphical Objects

The user can create the following graphical objects in the Draw application:

- lines
- squares/rectangles

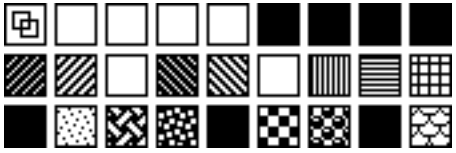
- rounded rectangles
- circles/ovals
- freehand drawing
- polygons (open and closed)
- arcs
- text frames
- shape recognition
- Stamps (bitmaps and PICTs)

Modifying Graphical Objects

The user can perform the following operations on graphical objects:

- drag out/move
- scale (resize)
- cut/copy/paste/clear/duplicate/select all
- undo (single level undo/redo)
- fill color (16 shades of gray) and fill patterns as shown in Figure 3-1. The current plans are for 8 shades of true gray and 18 gray patterns.

Figure 3-1 Fill/pattern palette



- pen size (4 widths)
- pen color (8 shades of gray)

- drag-and-drop (within app and via the clipboard)
- selection
- rotate in 90° increments
- flip horizontal/vertical
- group and ungroup
- constrained drawing (e.g., constrain lines to be horizontal or vertical, rectangles to be squares, ovals to be circles, arcs to be quadrants of circles, images to rotate in increments of 45°, etc.)

Arranging Graphical Objects

The user can arrange graphical objects in the following ways:

- front/back/forward/backward, send to front and send to back (kmc)
- align relative to other objects
- align left, center, right
- align top, center, bottom
- distribute objects evenly within selection (vertically, horizontally)
- revert to saved state

Setting Text Attributes

The user can specify the following characteristics for each text object. Only one style per text object.

- font
- size
- style (face)

Setting Paragraph Rulers

Each text object (paragraph) optionally displays the following graphical representations of its text layout characteristics:

- rulers. Support for inches and cm, set in prefs.
- gridlines

ClipArt Stamps (bitmaps)

The Draw toolbar provides an icon that pops up a selection of "clip art" bitmaps. Subsequently tapping in the draw area creates the selected bitmap.

Additional Features

The Draw application also provides the following additional features:

- import from/export to PICT format with embedded information necessary to enable a reasonable level of content preservation when exchanging with ClarisWorks. This import/export functionality may be provided by the Newton device, or as part of the desktop connectivity solution.
- printing (portrait orientation only, single page only)
- beaming and email (mail as attachment, no visible graphics in mail preview or I/O Box)
- API for developers to add tools to the tools palette

Gestures

The Draw application recognizes the following customized gestures:

Selection

The usual Newton hilite gesture (press and hold down pen on any part of an object) selects an object. A second way to select an object is to tap on the object. Tap works in all tool modes except the polygon tool. In the polygon tool, a tap is always interpreted as specifying a point of the polygon.

Tapping on a selected object deselects that object.

Specifying Vertices of Polygons

The user specifies the vertices of a polygon by tapping out the points. After the first tap, a small square is displayed as feedback for the location of the first vertex. After the Nth tap, we draw the edge from the previous point to the current point. To terminate the polygon, the user needs to tap on the first vertex again (to terminate a closed polygon), or tap on the same point twice in a row (to terminate an open polygon), or select another tool in the tools palette, or tap on any button in the status bar. (kmc)

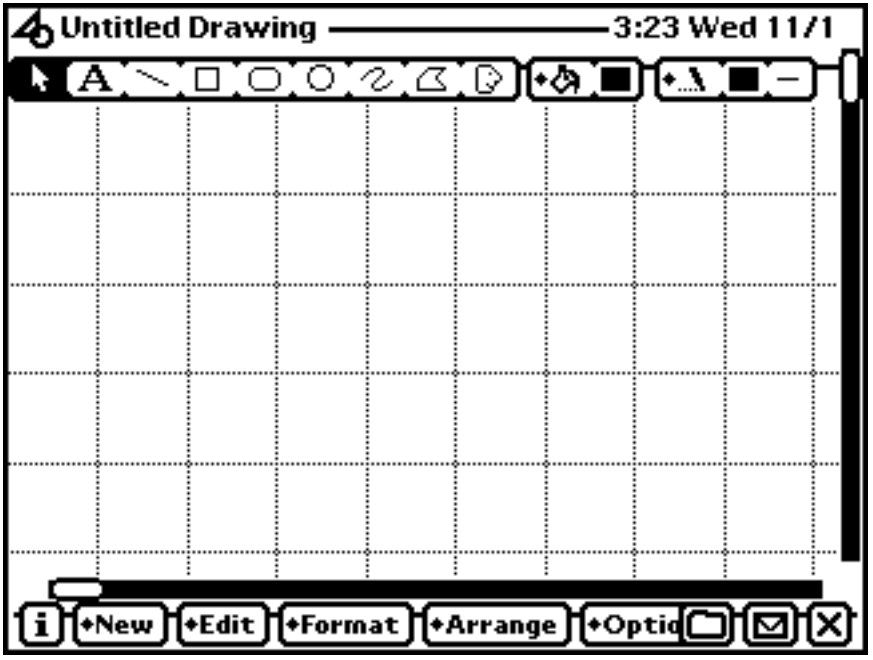
Locking Tools Palette

Tapping a drawing tool in the tools palette automatically locks in that tool (unlike the spring-loaded tools palette in ClarisWorks 4.0) Tapping an existing object will select it and switch to the "arrow" tool.

User Interface

This section provides a preliminary description of the Draw application's user interface. This interface is subject to change; for example, note that the icons for the stamp tool and arc tool are missing from the toolbar depicted in Figure 3-2.

Figure 3-2 Draw application user interface



Tool Bar

The tool bar allows the user to select drawing tools, choose fill patterns and and specify pen styles.

Arrow Tool

For shape selection and scrubbing. Icon definitely subject to change.

Text Tool

Tapping the text tool and then tapping in the drawing canvas places a text insertion caret where the user can type text. Text within a caption is limited to a single font, size, and face.

The text tool creates a text box containing an insertion caret. Box size is determined when the user initially taps/drags the stylus.

Line Tool

Tapping the line tool and then drawing in the canvas drags out a rubber-banded line.

Rectangle Tool

Tapping the rectangle tool and then drawing in the canvas drags out a rectangle.

Rounded Rectangle Tool

Tapping the rounded rectangle tool and then drawing in the canvas drags out a rounded rectangle.

Oval Tool

Tapping the oval tool and then drawing in the canvas drags out an oval.

Arc Tool

Allows user to draw arcs (kmc)

Freehand Tool

Tapping the freehand tool and then drawing in the canvas draws an ink sketch.

Polygon Tool

Tapping the polygon tool puts you in a modal state where each tap in the canvas will layout an edge of a polygon. After tapping at least two points, tapping on the first point again will create a closed polygon. To create an open polygon, tap on the last point twice. Tapping on the original point twice (without tapping any other points) is a no-op, and no polygon is created.

Once a polygon is completed, it is selected and the tool switches to the "Arrow" tool for movement/manipulation.

Stamp Tool

Tapping the stamp tool brings up a popup of stamps to choose from. Once selected, you can use it to create as many stamps as you like.

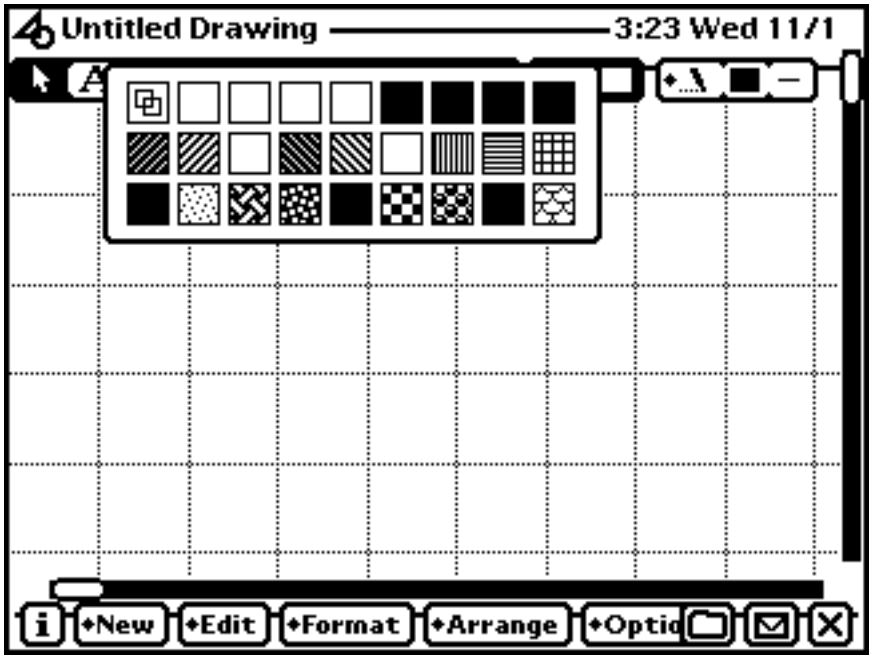
Shapes Tool

Using the Newton Shape recognizer, you can draw the shapes you'd like directly on the screen.

Fill Pattern Palette

The tool bar's fill pattern icon displays a palette from which the user can choose a fill pattern, as shown in Figure 3-3.

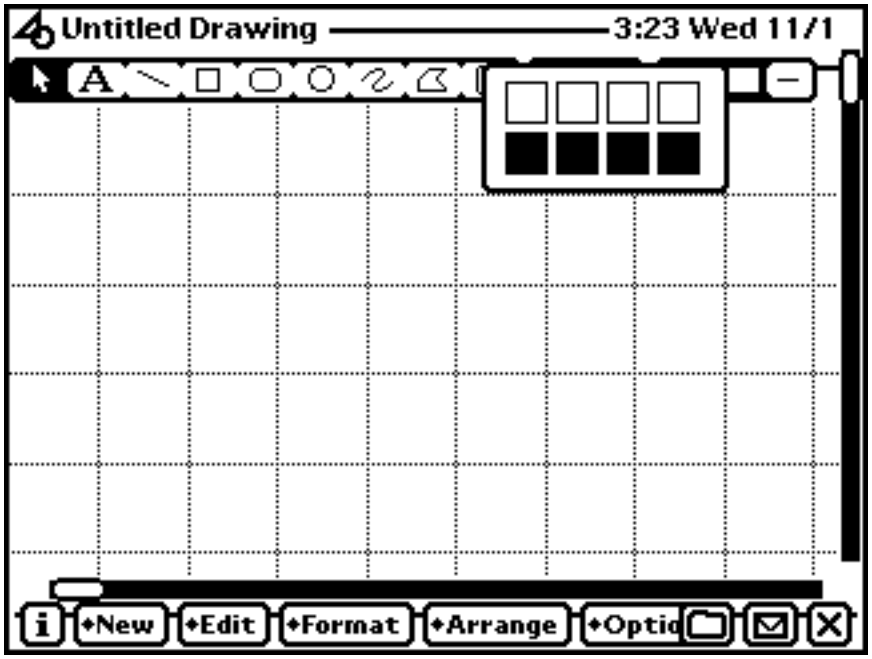
Figure 3-3 Selecting a fill pattern



Pen Pattern Palette

The tool bar's pen pattern icon displays a palette from which the user can choose a pen pattern, as shown in Figure 3-3.

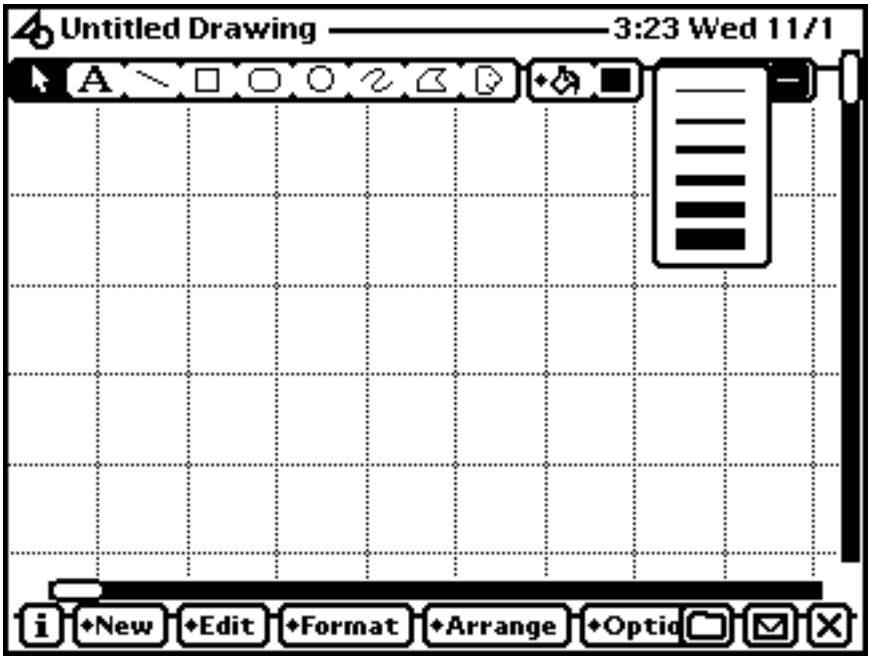
Figure 3-4 Selecting a pen pattern



Pen Size Palette

The tool bar's pen size icon displays a palette from which the user can choose a pen size, as shown in Figure 3-3.

Figure 3-5 Selecting a pen size



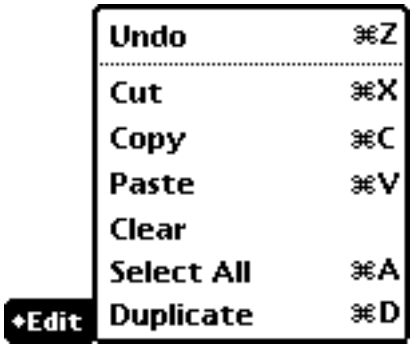
New Button

••• to be supplied •••

Edit Button

The Edit button displays a picker that provides the Undo, Cut, Copy, Paste, Clear, Select All, and Duplicate commands, as shown in Figure 3-6. The Draw application also provides keyboard equivalents for all of the items in this picker except the Clear command.

Figure 3-6 The Edit button collapsed and expanded



Undo

Undo is an undo/redo command. Unlike the Mac, the Newton OS implements a global Undo instead of a per-application Undo. As a result, if the user makes an edit in a draw document, and then makes an edit outside the Draw application, the change in the draw document cannot be undone.

Picking Undo from the menu, typing command-Z, or pressing the Undo key on the eMate 300 keyboard will invoke Undo/Redo. (kmc)

Cut, Copy, Paste, Clear, Duplicate

Cut, Copy, Paste, Clear, and Duplicate do the usual operation on the selected object(s). Clear deletes the selected object without putting it on the clipboard. Duplicate makes a copy , pastes it back into the draw document slightly offset from the original, and selects the copy without changing the clipboard. As discussed in the "eMate 300 Word Processor ERS," Cut and Copy will put the cut/copied object on the visual clipboard.

If there is no selection, dragging the pen over the command does not highlight the command (per the Newton UI Guidelines).

Select All

Select All selects all graphical objects in the active draw document.

Arrange Button

This section describes each of the items in the Arrange picker, which provides items for manipulating the order, alignment, orientation, and grouping of graphical objects. This picker is displayed by the Arrange button, as shown in Figure 3-7.

Figure 3-7 The Arrange button

Move Forward, Move Backward

Move Forward and Move Backward move the selected graphical object(s) one layer forward or backward, respectively, in the stacking order.

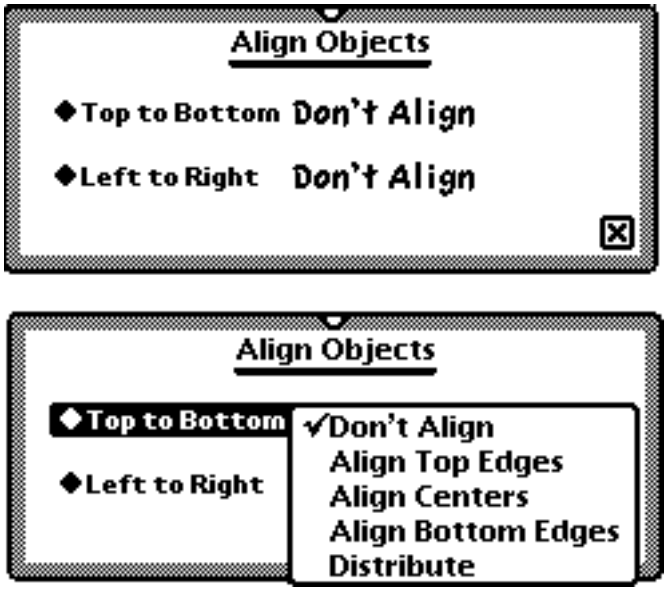
Move to Front, Move to Back

Move to Front and Move to Back move the selected object(s) to the top or bottom, respectively, in the stacking order.

Align Objects

This item displays the slip shown in Figure 3-8. The user can select options in this slip to align the selected objects in relation to each other. Following the usual Newton convention that changes in a slip should take effect immediately, selecting a choice from each of the pickers in this slip changes the alignment of the selected objects immediately. Each time the slip is opened, the initial value of each picker is "Don't Align."

Figure 3-8 The Align Objects slip



Rotate Right/Left

Rotates each selected object in 90° increments clockwise around each object's center.

Flip Horizontally, Flip Vertically

Flip Horizontally flips each selected object over, along a vertical axis through the center of each object. Flip Vertically flips each selected object over, along a horizontal axis through the center of each object.

Group, Ungroup

Group makes the selected objects into a single selectable object. Ungroup turns a group back into individually selectable objects.

Font Button

••• to be supplied •••

Tools Button

••• to be supplied •••

Hide Grid/Show Grid

Hides or shows the graphics grid. The menu command toggles between the two choices.

Hide Tools/Show Tools

Hides or shows the tool bar (which includes the fill and pen buttons). The menu command toggles between the two choices.

Show Rulers/Hide Rulers

Hides or shows the graphics rulers. The menu command toggles between the two choices.

Info Button

••• to be supplied •••

About

••• to be supplied •••

Help

••• to be supplied •••

Prefs

••• to be supplied •••

Developer API

This section describes the NewtonScript interface to the Draw application, including how to register your own tool bar items.

Adding and Removing Custom Drawing Tools

The developer API allows adding addtional tools to the tool bar at the top of the screen. Adding a tool allows the user to create new types of draw objects.

The New button is added by calling a method of the viewDef to register the new button:

`viewDef:RegTool(`*appSym*`,` *drawToolTemplate*`)`

The best way to do this in an install script would be:

```
local viewDef := GetViewDefs('drawPaper).default;
local sym := EnsureInternal('|mySymbol:myCompany|);
if viewDef  then
   viewDef:RegTool(sym, drawToolTemplate)
```

This is normally done in the `InstallScript` function of an auto part. The *drawToolTemplate* must be a template that uses the `protoDrawTool` prototype.

The `RemoveScript` function of the package should call:

```
viewDef:UnRegTool(sym)
```

The draw object created must be an array of shapes which has a class that must match the 'class slot of this tool. This array should have a style frame at the front to make sure you get the drawing environment you want. The last two objects in the array should be an "invisible" style and a rect. The rect should be the size of the shape you are returning and the style should be:

```
{penSize: 0, fillPattern: vfNone, selection: nil }
```

This makes hiliting and hit testing work. This rect at the end of your shapes array will allow the user to hit the "corners" of your shape array for resizing (even if you don't have any real shapes in that corner)

Drawing Tool Template

Your custom drawing tool is defined by a template that has the following slots:

`class`	Required. Symbol which uniquely identifies the tool and the objects created by this tool.
`icon`	Required. Bitmap that represents this tool in the toolbar. Note that the tool border is drawn for you—this bitmap supplys only the interior bits.
`makeObject`	Required. Function object executed when the user drags or taps with this tool in the canvas. This function creates a graphical object having the bounds passed in its arguments. (These values are specified by the user when dragging out a new shape with this tool.) The graphical object that this function returns would normally be an array of shapes. The array must be of the

same class as the tool (so the draw app can call your tool back for edit operations) See comments above.

This function must be of the form

`func (`*left*`,` *top*`,` *right*`,` *bottom*`,` *style*`) begin ... end;`

left	x-coordinate of start point of rectangle user dragged with this tool.
top	y-coordinate of start point of rectangle user dragged with this tool.
right	x-coordinate of end point of rectangle user dragged with this tool.
bottom	y-coordinate of start point of rectangle user dragged with this tool.

Note

If the user drags up and to the left, the values passed as the *right* and *bottom* arguments can be less than those passed as the *left* and *top* arguments. ◆

style	The current user defaults for penSize, penPattern, fillPattern, and font.

`setAttribute` Required. Function object executed when a shape is selected (attribute = 'selection) or draw attributes are changed (penSize, penPattern, fillPattern, font). The return value is ignored. This function must be of the form

`func (`*shape*`,` *attribute*`,` *newValue*`) begin ... end;`

`createByTapping` The value `true` specifies that the user can tap to create an object. The default value is `nil`.

`createByDragging` The value `true` specifies that the user can drag out bounds for object. The default value is `true`

`dragARectangle` The value `true` specifies that the user drags out a plain rectangle when creating an object. If the value of this

slot is `nil`, the template's `MakeDragObject` method is invoked. The default value is `true`.

`currentDrawStyle`

Optional. Frame containing the current (default) drawing style.

Developer-Defined Drawing Tool Methods

You can further customize your drawing tool's behavior by supplying in its template the optional methods described in this section as appropriate. The system supplies a default version of any method you don't supply. The default methods are made available to your drawing tool when its template

is registered with the Draw application by means of the `RegTool` method, as described in "Adding and Removing Custom Drawing Tools" (page 2-17).

`makeDragObject` Optional. Function object executed when this template's `dragARectangle` slot is `nil` and the user drags out an object. This function must be of the form

```
func (left, top, right, bottom, style) begin ... end;
```

This function creates a shape to show while the user drags out a bounding box. The default version of this method calls your template's `MakeObject` method.

left	See the `MakeObject` method.
top	See the `MakeObject` method.
right	See the `MakeObject` method.
bottom	See the `MakeObject` method.
style	See the `MakeObject` method.

`AdjustBounds` Optional. Function object executed when user is creating a bounding box or resizing an existing object. This function must be of the form

```
func (shape, style, left, top, right, bottom,
      constrain) begin ... end;
```

The default version of this method limits the bounding box to always be square.

shape	Graphical object being edited.
style	See the `MakeObject` method.
left	See the `MakeObject` method.
top	See the `MakeObject` method.
right	See the `MakeObject` method.
bottom	See the `MakeObject` method.
constrain	The value `true` indicates that the shift key is pressed.

`ScaleShape` Optional. Function object executed when user is resizing an existing object; this method actually resizes the object. This function must be of the form

```
func (shape, oldBounds, newBounds) begin ... end;
```

This function MUST return a shape (either the original or a new one.)The default version of this function calls the global function `ScaleShape(`*shape*`, `*oldBounds*`, `*newBounds*`)` and flips the shape with `MungeShape` as necessary.

`CanvasClickScript`

Optional. Function object executed when user taps in the content area and the tool that defines this method is selected. This function is intended to provide additional control over the creation of shapes. This function must be of the form

```
func (unit) begin ... end;
```

unit Stroke unit passed to this method by the recognition system.

Your version of this method must return one of the following values:

`true` tool has handled click completely

`nil` tool has punted completely, click will turn into tap or hilite.

`'continue` tool wants default shape creation behavior (i.e. tapping or draging out a shape. All of the above methods (such as makeObject) will be called normally)

graphicalObject tool has created a graphical object to be added to drawing. This is the same object returned by the `MakeObject` function that this template defines.

`ToolClickScript` Optional. Function object executed when user taps on this template's tool in the tool bar. This function must be of the form

```
func (Unit) begin ... end;
```

This function lets the tool do its own tracking of the click. For example, the stamps tool uses this method to display a slip containing the stamp bitmaps. It is

recommended that your override method call the inherited `ToolClickScript` method.

unit — Stroke unit passed to this method by the recognition system.

`ToolBegin` — Optional. Function object executed before the `ToolClickScript` method. Your tool can use this method to perform setup tasks before creating shapes.

This function must be of the form

```
func () begin ... end;
```

`ToolEnd` — Optional. Function object executed when the user selects a tool other than the currently active one. Your tool can use this method to perform housekeeping tasks.

This function must be of the form

```
func () begin ... end;
```

Drawing Methods

This section describes methods provided by the Draw application.

GetContents

viewDefView`:GetContents()`

Returns a pointer to all the shapes. This array is always an even length of style/shape pairs. (i.e. Every shape is proceeded by a style).

SetContents

viewDefView`:SetContents(newShapes)`

Replaces the contents of the current drawing with newShapes, an array of shapes. The array can be any combination of styles and shapes.

GetCanvas

viewDefView`:GetCanvas()`

Returns the view which contains and edits the actual style/shape pairs in the document. The following methods/slots all are a part of this view.

AddShape

`canvas:AddShape(`*shape*`, `*style*`, nil)`

Adds *shape* to the document.

style	The style to use for drawing *shape*. If *style* is nil, the default drawing style is used.

The third argument must be `nil`.

AddShapeToSelection

canvas`:AddShapeToSelection(`*shape*`)`

Adds shape to the current selection.

shape	an existing shape in the current document.

ClearSelection

canvas`:ClearSelection()`

Deselects the current selection.

SelectAll

canvas`:SelectAll()`

Selects all shapes. This method's return value is unspecified.

GetSelectedShapes

canvas`:GetSelectedShapes(`*makeCopy*`)`

Returns an array of shapes in the current selection.

makeCopy	True specifies that this method is to return a `DeepClone` of the shapes.

◆ **WARNING**
If you don't work with a copy of the shapes, don't edit them in any way. ◆

DirtyShape

canvas:`DirtyShape(`*shape*`)`

Dirties *canvas* only enough to draw the area around *shape*'

EditGroup

canvas:`EditGroup()`

Groups the current selection

EditUnGroup

canvas:`EditUnGroup()`

Ungroups the current selection

EditCopy

canvas:`EditCopy()`

Copies the current selection to the clipboard.

EditCut

canvas:`EditCut()`

Cuts the current selection and places it on the clipboard.

EditPaste

canvas:`EditPaste()`

Replaces the current selection with the contents of the clipboard.

EditDelete

canvas:`EditDelete()`

Deletes the current selection.

EditDuplicate

canvas:`EditDuplicate()`

Duplicates the current selection.

EditUndo

canvas:EditUndo()

Reverses the effects of the most recent editing operation.

Draw Patterns

To add patterns to the fill pattern palette, use the RegPatterns method of the viewDef from within an InstallScript method, as shown in the following example:

```
local viewDef := GetViewDefs('drawPaper).default;
local sym:= EnsureInternal('|mySymbol:myCompany|);
if viewDef  then
    viewDef:RegPatterns(sym, arrayOfPatterns)
```

The *arrayOfPatterns* argument is an array of any of the QD patterns or constants such as vfBlack, kRGB_Gray5, a graypattern or a dithered Pattern.

Use the corollary method UnRegPatterns to remove patterns from the fill palette:

```
viewDef:UnRegPatterns(sym)
```

Draw Stamps

To add stamps to the stamp palette, use the RegStamps method of the viewDef from within an InstallScript method, as shown in the following example:

```
local viewDef := GetViewDefs('drawPaper).default;
local sym:= EnsureInternal('|mySymbol:myCompany|);
if viewDef  then
    viewDef:RegStamps(sym, arrayOfStamps)
```

The *arrayOfStamps* argument is an array of bitmaps or PICTs as returned by the GetPictAsBits, GetNamedResource, or GetPictAsPixels functions.

Use the corollary method UnRegStamps to remove stamps from the stamp palette:

`viewDef:UnRegStamps(sym)`

Note

It is recommended that each stamp be <1K in size. Stamps groups must not exceed 4x6 arrays of stamps (24 stamps in an array). ◆

NewtonWorks Draw Application ERS

Word Processing View ERS

This document describes how to use word-processing views, which are provided by `protoTXView`. This view supports the editing of large amounts of text. The word processor packaged with the eMate 300 uses this view to provide word processing operations.

About protoTXView And the View System

You implement word-processing views in your applications with `protoTXView`. This section describes some of the non-standard view features of `protoTXView`. The remainder of this document provides a reference manual for this proto.

Application-defined Methods

The protoTXView proto supports some, but not all, of the standard Newton view system application-defined methods. Table 4-1 shows the status of application-defined methods with protoTXView.

Table 4-1 Use of application-defined method in protoTXView

Method	Use in protoTXView
ViewGestureScript	Supported.
ViewClickScript	Supported.
ViewStrokeScript	Supported.
ViewWordScript	Supported.
ViewKeyDownScript	Called if you specify vSingleKeystrokes in the textFlags slot.
ViewKeyUpScript	Called if you specify vSingleKeystrokes in the textFlags slot.
ViewScrollUpScript	Use protoTXView:Scrollfor scrolling. Note, however, that the view system does send this message to protoTXView.
ViewScrollDownScript	Use protoTXView:Scrollfor scrolling. Note, however, that the view system does send this message to protoTXView.
ViewOverviewScript	Use protoTXView:Scrollfor scrolling. Note, however, that the view system does send this message to protoTXView.
ViewGetDropTypesScript	Supported.
ViewGetDropDataScript	Supported.
ViewHiliteScript	protoTXView does its own scrolling. Note, however, that the view system does send this message to protoTXView.

View Slots

Some of the standard view slots are used differently for `protoTXView`, as shown in Table 4-2.

Table 4-2 Use of standard view system slots in `protoTXView`

Slot name	**Use in** `protoTXView`
`recConfig`	Ignored.
`_keyboard`	Ignored.
`textFlags`	Ignored except for the `vSingleKeystrokes` flag.
`viewFont`	Used to determine the default font in your view. If this slot is missing, the `userFont` is used as the default font.
`viewJustify`	Can be used to specify sibling and parent-relative alignment, but is not used with the `protoTXView` view.
`viewFormat`	Ignored.
`viewLineSpacing`	Ignored.

Other View Features

This section describes several characteristics of `protoTXViews`.

- No child views are allowed within protoTXViews. If a child is added to a protoTXView, it will not update correctly, since the protoTXView assumes it has control of the entire drawing area.
- No word or text recognition is done in protoTXView views.
- The drag and drop hooks are all supported. Like `editViews`, `protoTXViews` support the movement of `'text`, `'ink`, `'picture` and `'polygon` content; in addition, protoTXViews also `'shape` types as well.

  The formats of the first four types are the same as for `editViews`, and are

documented in *Newton Programmer's Guide*. The shape format consists of a shape array or shape frame, which can be drawn with `DrawShape`. You can override drag/drop behavior by implementing any of the view system application-defined methods, including `ViewGetDropTypesScript` and `ViewGetDropDataScript`.

Terminology

The word-processing view refers to paragraph settings as a **ruler**. Each paragraph has a ruler, which defines its margins, line spacing, justification, and tab settings.

There are also **style runs** in word-processing documents. Each style run specifies font appearance attributes that apply to a range of data in the document.

Common Parameter Descriptions

This section describes the parameter values that are used by a number of the word-processing view methods.

The Range Frame

Several of the word-processing methods use a range parameter to specify a range of characters in the view.

range — A frame with two slots: `'first` and `'last`. This frame defines a text range from `'first` to `'last`, inclusive. Each slot is required. The value of each slot must be a positive integer value.

The Font Specification Frame

Several of the word-processing methods use a font specification parameter to specify the font to use for an operation.

fontSpec A font specification, used exactly as for paragraph views. This can be an integer, frame, or binary object specification of a font.

The Graphics Specification Frame

Several of the word-processing methods use a graphics specification parameter to specify a shape object to use for an operation.

graphicsSpec A frame with two slots: class `'graphics` and `'shape`, which is a shape object as described in *Newton Programmer's Guide*.

The Ruler Information Frame

Several of the word-processing methods use a ruler information parameter to specify style information for a range of characters in the view.

rulerInfo A frame describing the attributes of a ruler. This frame contains the following slots:

justification Optional. One of the symbols: `'left`, `'right`, `'center` or `'full`.

indent Optional. The indentation of the first line of the paragraph, expressed as an integer number of pixels measured from the left edge of the text bounds.

leftMargin Optional. The left margin of the paragraph, expresses as an integer number of pixels measured from the left edge of the text bounds.

rightMargin Optional. The right margin of the paragraph, expresses as an integer

	number of pixels measured from the right edge of the text bounds.
lineSpacing	Optional. The number of lines between text lines in the paragraph, expressed as an integer number of lines. A value of 2 indicates double spacing.
tabs	Optional. An array of tab frames, as described in the next section.

Tab Frames

Each tab setting in a ruler is specified by a frame with two slots:

kind	Required. One of the symbols: `'left`, `'right`, `'center` or `'decimalPoint`.
value	The tab value expressed as an integer number of pixels measured from the left edge of the text bounds.

Initialization

This section describes the methods that you can call to set the different storage and geometry characteristics of your word-processing views. You can call these from your `ViewSetupForm` script.

SetStore

protoTXView`:SetStore(`*store*`)`

Specifies that the text in your word-processing view is to be stored as a virtual binary object (VBO). The default is to store the text as a RAM-based object.

store	The store object to contain the text. This is usually the same as your soup entry's store.

Storing text as a VBO allows the text to be as large as the amount of free space on the store. Text is swapped in and out of memory as required.

IMPORTANT

You cannot change the storage-type of your text once your `ViewSetupDoneScript` has executed. ◆

SetGeometry

protoTXView:`SetGeometry(`*isPaged*, *width*, *height*, *margins*`)`

Changes the geometrical characteristics of the view.

isPaged — Required. Specifies whether the text is laid out in many pages, with text flowing from one page to another, or bounded by one box. Use `true` to indicate that the text is paged, or `nil` to indicate that all text is contained in one box. The default value is `nil`.

A page break inserted into a non-paged view has no effect. However, the page break will have an effect if the text is pasted into a paged view.

width — Required. The width of the text bounds, or the page width if *isPaged* is true. This value is expressed as an integer number of pixels, and includes the left and right margins. The default value is the width of the view (as specified in the `viewBounds` rectangle).

height — Required. The height of the text bounds, or the page height if *isPaged* is true. This value is expressed as an integer number of pixels, and includes the top and bottom margins. The default value is the width of the view (as specified in the `viewBounds` rectangle).

margins — Required. A rectangle that specifies the margins of the page or text box. The rectangle is specified as a frame with four slots:

top	The top edge of the rectangle.
left	The left edge of the rectangle.
bottom	The bottom edge of the rectangle.
right	The right edge of the rectangle.

The default margins is a rectangle in which all four values are `0`.

You can call this method at any time to change the geometrical characteristics of your word-processing view.

IMPORTANT

You cannot change the *isPaged* characteristic once the `ViewSetupDoneScript` has executed. If you attempt to change isPaged after that time, the new setting is ignored. ◆

Getting Info

This section describes the methods you can use to retrieve information about the content of a word-processing view.

GetRangeData

protoTXView`:GetRangeData(range, which)`

Returns a certain kind of data (text or styles) for the specified range in a word-processing view.

range — A frame with two slots: `'first` and `'last`. This frame defines a text range from `'first` to `'last`, inclusive. Each slot is required. The value of each slot must be a positive integer value.

which — Specifies the kind of data to retrieve. You can specify one of the following values:

- `'text` — A string allocated from the NS heap.
- `'styles` — An array with two entries for each style run in the range. The first entry specifies the number of characters, and the second is either a `fontSpec` or a `graphicsSpec` frame. The run length is always 1 for `graphicsSpec` objects.

  See "The Font Specification Frame" (page 3-5) for a description of `fontSpecs` and "The Graphics Specification Frame"

	(page 3-5) for a description of `graphicsSpecs`.
`'rulers`	An array with two entries for each ruler. The first entry specifies the number of characters for the ruler, and the second contains a rulerInfo frame. See "The Ruler Information Frame" (page 3-5) for a description of the `rulerInfo` frames.
`'all`	Returns all information in a frame that contains three slots: `'text`, `'styles`, and `'rulers`.

The `GetRangeData` methods returns data for a range of text within a word-processing view.

GetCountCharacters

protoTXView`:GetCountCharacters()`

Returns the number of characters in a word-processing view.

This cannot be called before `protoTXView`'s `viewSetupDoneScript` gets called, since the document won't have been initialized yet.

FindString

protoTXView`:FindString(`*str, startOffset, options*`)`

Searches for matching text in a word-processing view.

str	The string to be searched.
startOffset	The offset at which the search should start.
options	Must be `nil`. Currently the search is not case sensitive.

The `FindString` method searches in a word-processing view for a sequence of characters that matches *str*. The search begins at *startOffset* from the beginning of the view and continues until a match it made or the end of the text is reached.

The `FindString` method returns the offset of the matching string in the view. If no match is found, `FindString` returns `nil`.

GetWordRange

protoTXView`:GetWordRange(`*offset*`)`

Returns a range frame that specifies the starting and ending offsets of the word that follows the specified *offset* in a word-processing view.

offset	The offset from the beginning of the text in the word-processing view.

The `GetWordRange` method searches forward in the text to discover the first character and last character of the word that follows *offset*. This method considers a word to consist of alphanumeric characters delimited by white space (tabs, returns, spaces, and graphic runs). Hyphenated words are considered single words.

The `GetWordRange` method returns a range frame, as described in "The Range Frame" (page 3-4). The `GetWordRange` method returns nil if it does not find a word after *offset*.

CharToPoint

protoTXView`:CharToPoint(`*offset*`)`

Returns a frame that specifies the coordinates of the character at the specified *offset* in a word-processing view.

offset	The offset from the beginning of the text in the word-processing view.

The `CharToPoint` method returns a frame with three slots:

x	The horizontal coordinate of the top-left corner of the rectangle enclosing the character at offset.
y	The vertical coordinate of the top-left corner of the rectangle enclosing the character at offset.
lineHeight	The line height of the line that contains the character.

The value of each of the slots is in global coordinates, relative to the top-left of the screen. This means that the y-value can be negative if the view has been scrolled down.

PointToChar

protoTXView:`PointToChar(point)`

Returns a range frame for the character at the specified point.

point	A point frame containing two slots:	
	x	The horizontal coordinate, as a global coordinate values, relative to the top-left of the screen.
	y	The vertical coordinate, as a global coordinate values, relative to the top-left of the screen.

The `PointToChar` method returns a range frame for the character at the specified point. Range frames are described in "The Range Frame" (page 3-4).

The values of the slots in the returned range frame are as follows:

- If *point* is inside of a text run, the `'first` and `'last` slots will have the same value.
- If *point* is inside of a graphics run, the value of the `'last` slot will be `1` greater than the value of the `'first` slot.

GetLineRange

protoTXView:`GetLineRange(offset)`

Returns a range frame for the span of the line that contains the specified *offset*.

offset	The offset from the beginning of the text in the word-processing view.

The `GetLineRange` method returns a range frame corresponding to the span of the line that contains *offset*. The returned range includes the trailing carriage return if it exists. If the *offset* is on a carriage return, the range returned is the one preceding the carriage return and including it. If the *offset* is after a carriage return, the next run is returned. Page breaks are treated the same as carriage returns.

Editing

This section describes the methods that you can use to perform editing operations in your word-processing views. Many of the editing operations can be undone by the user without any coding effort on your part.

Note that when an editing operation crosses paragraph boundaries, the ruler of the first paragraph is used.

Cut

protoTXView:`Cut()`

Removes the highlighted range text and copies it to the clipboard.

Copy

protoTXView:`Copy()`

Copies the highlighted range text to the clipboard.

Paste

protoTXView:`Paste()`

Replaces the highlighted range text with the clipboard content.

Clear

protoTXView:`Clear()`

Removes the highlighted range text and any included graphics, styles, and rulers. The clipboard is not changed.

ChangeRangeRuns

protoTXView:`ChangeRangeRuns(`*range*, *fontSpec*, *toggleFace*, *undoable*`)`

Changes the font attributes for a range of text in a word-processing view.

range	A range frame defining the text range that you want to change. See "The Range Frame" (page 3-4).
fontSpec	A font specification frame that can contain nil slots. Any non-`nil` slots in this frame indicate new text attributes for the range. Use `nil` slots to indicate that the corresponding attribute is not to change. See "The Font Specification Frame" (page 3-5).
toggleFace	If this value is `nil`, the font face attribute used for all text in the *range* is changed to the value of the `face` slot in *fontSpec*, unless the face slot is `nil`. If this value is non-`nil`, the font face used for all text in the *range* is toggled: if one of the font face values specified in the `face` slot in *fontSpec* is used across the entire range, `ChangeRangeRuns` turns off that attribute. Otherwise, that attribute is applied to all of the text in *range*.
undoable	If non-`nil`, the operation is undoable.

ChangeRangeRulers

protoTXView:`ChangeRangeRulers(`*range*, *ruler*, *undoable*`)`

Changes the attributes of the rulers in a range of text.

range	The rulers in this *range* are changed. Note that the *range* is grown to enclose entire paragraphs.
ruler	A rulerInfo frame. Any non-`nil` slots in this frame indicate new text attributes for the *range*. Use `nil` slots to indicate that the corresponding attribute is not to change. See "The Ruler Information Frame" (page 3-5).
undoable	If non-`nil`, the operation is undoable.

Replace

protoTXView:Replace(range, data, undoable)

Replaces the data inside of the specified range with the specified data. You can replace text and/or graphics with this method.

range A range frame defining the text range that you want to change. See "The Range Frame" (page 3-4).

data A frame describing the new data. This can be a graphics specification frame, as described in "The Graphics Specification Frame" (page 3-5). Or *data* can be a frame with the following slots:

- 'text — If this slot is nil, style runs are replaced, but the text remains the same. If non-nil, this is the new text.
- 'styles — If this slot is nil, the new text uses the style attributes at the start of the *range*.

  If non-nil, this is an array with two entries for each style run in the range. The first entry specifies the number of characters, and the second is either a fontSpec or a graphicsSpec frame. The run length is always 1 for graphicsSpec objects.

  See "The Font Specification Frame" (page 3-5) for a description of fontSpecs and "The Graphics Specification Frame" (page 3-5) for a description of graphicsSpecs.

undoable If non-nil, the operation is undoable.

The following call to the Replace method changes the first ten characters to the word "any" using the system font, bold face, point size 9:

```
myTxView:Replace( {first:0, last:10},
       {text:"any", styles: [3, tsSystem+tsSize(10)+tsBold]} );
```

The following call to the `Replace` method changes the first ten characters of the range to a rounded rectangle:

```
myShape:= makeroundrect(0, 0, 50, 50, 16);
myTxView:Replace({first:0, last:10},
                 {class: 'graphics, shape: myShape})
```

ReplaceAll

protoTXView`:ReplaceAll(str, startOffset, options, data)`

Searches the text in the view, starting at the specified *startOffset*, and replaces all occurrences of *str* with *data*.

str	The string to be replaced
startOffset	The starting offset of the search in the text.
options	Must be `nil`.
data	A frame describing the new data. This can be a graphics specification frame, as described in "The Graphics Specification Frame" (page 3-5). Or *data* can be a frame with the following slots:

`'text`	If this slot is `nil`, style runs are replaced, but the text remains the same. If non-`nil`, this is the new text.
`'styles`	If this slot is `nil`, the new text uses the style attributes at the start of the *range*. If non-`nil`, this is an array with two entries for each style run in the range. The first entry specifies the number of characters, and the second is either a `fontSpec` or a `graphicsSpec` frame. The run length is always 1 for `graphicsSpec` objects. See "The Font Specification Frame" (page 3-5) for a description of `fontSpecs` and "The Graphics Specification Frame"

(page 3-5) for a description of `graphicsSpecs`.

The `ReplaceAll` method returns a count of the replacements that it made.

WARNING

This operation can not be undone. ◆

Storage

This section describes the methods that you can use to save and retrieve word-processing documents. What you normally do with these documents is to put the `protoTXView` data frame in a soup entry slot and then use the standard soup methods to store or modify it.

Externalize

protoTXView`:Externalize()`

Returns a reference to an object that contains all of the data for the document, including the text, style runs, and rulers. The object is allocated from the NS heap.

Note

The object referenced by this method is intended for use as a block box, which means that you can only use it for two purposes: to save it into a soup or as input to the `Internalize` method. ◆

Internalize

protoTXView`:Internalize(object)`

Replaces the current content of the view with the data in *object*.

object A reference to an object, as returned by the `Externalize` method.

The Internalize method replaces the contents of a `protoTXView` view with the data retrieved by a previous call to the `Externalize` method and resets the highlight range to `(0,0)`.

IsModified

protoTXView`:IsModified()`

Returns a non-nil value if the contents of the view have changed since the last call to `Externalize` or `Internalize`.

Note

For improved performance, you should only call the `Externalize` method when `IsModified` returns true. ◆

Scrolling

This section describes the methods that you can use for scrolling your word-processing views.

Note

The protoTXView has built-in automatic scrolling. You usually only have to call the methods described in this section if you have scrollers attached to your view. ◆

Scroll

protoTXView`:Scroll(scrollValues)`

Scrolls the content of the view vertically and/or horizontally, as defined in *scrollValues*.

scrollValues — A frame with `'x` and `'y` slots. The content is scrolled by `x` pixels horizontally and `y` pixels vertically.

For example, to scroll from page 1 to page 2, use the following:

```
Scroll({x:0, y:pageHt});
```

GetScrollValues

protoTXView:`GetScrollValues()`

Returns a frame with `'x` and `'y` slots containing the current scroll values. You can use call this method and then use the returned values to position your the thumbs on your scrollers.

GetTotalHeight

protoTXView:`GetTotalHeight()`

Returns the current total text height. You can use this value to set the maximum value of a vertical scroller.

If the view is non-paged, `GetTotalHeight` returns the height of the entire view, as set with the `SetGeometry` method. If the view is paged, `GetTotalHeight` returns the height of the entire view (as specified in `SetGeometry`) multiplied by the total number of pages.

GetTotalWidth

protoTXView:`GetTotalWidth()`

Returns the current total text width. You can use this value to set the maximum value of a horizontal scroller.

The `GetTotalWidth` method returns the width of the view, as set with the `SetGeometry` method.

GetScrollableRect

protoTXView:`GetScrollableRect()`

Returns a rectangle frame describing the global coordinates of the rectangle in which text is displayed. This is normally equal to the view bounds minus the ruler area. The returned frame has four slots: `'top`, `'left`, `'bottom`, and `'right`.

ViewUpdateScrollersScript

protoTXView:`ViewUpdateScrollersScript(`*updateMaxVal*, *scrolled*`)`

Is called to notify you that the scrollers need to be updated.

updateMaxVal	A Boolean value indicating whether you need to update the maximum value of the vertical scroller.
scrolled	A Boolean value indicating whether you need to update the scroller thumb values (of both the horizontal and vertical scroller).

Highlighting

This section describes the methods that you can use to work with the currently highlighted (selected) range of text in your word-processing view.

Note that `protoTXView` does not support discontinous highlights.

If there is currently an insertion point, the highlight range has a length of `0`.

GetHiliteRange

protoTXView:`GetHiliteRange()`

Returns a range frame describing the current highlight range. Range frames are described in "The Range Frame" (page 3-4).

SetHiliteRange

protoTXView:`SetHiliteRange(`*newRange*, *showHilite*, *setKeyView*`)`

Changes the current highlight range to *newRange*.

newRange	A range frame that specifies the new highlight range. Range frames are described in "The Range Frame" (page 3-4).
showHilite	A Boolean value. If this is `true`, the content of the view is scrolled as necessary to display the new range. If the

range is larger than the screen, the start of the range is displayed.

setKeyView A Boolean value. If this is `true`, the view becomes the current key view, which activates the view for keyboard input. If this is `nil`, the highight range is shown as an inactive selection.

GetContinuousRun

protoTXView`:GetContinuousRun()`

Returns a frame that specifies the style run containing the current highlight range.

If the current highlight range contains only one graphics object, GetContinuousRun returns a graphics specification frame, as described in "The Graphics Specification Frame" (page 3-5) .

Otherwise, GetContinuousRun returns a font specification frame, as described in "The Font Specification Frame" (page 3-5). Any non-nil values in the font specification frame indicate values that are continous for the highlight range.

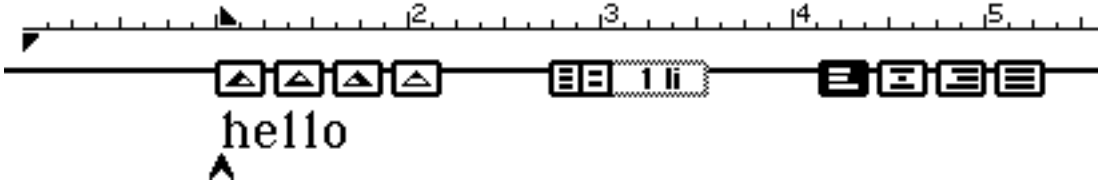
Note

You can use this method to check the appropriate items in the style menu(s). ◆

Ruler User Interface

This section describes the methods you can use to work with the ruler user interface that is built into `protoTXView`. When the ruler is displayed, it always appears at the top of the view, as shown in Figure 4-1.

Figure 4-1 The displayed ruler



Whether or not the ruler is shown affects the values returned by the `GetTextViewRect` method, since the operational area for text editing is smaller if the ruler takes up space.

ShowRuler

protoTXView:`ShowRuler(rulerSettings)`

Shows the ruler if it is not currently shown. The ruler is hidden by default.

rulerSettings A frame with one slot, `'type`. The value of this slot can be either `'metric` or `'inches`. If this parameter is nil, `'inches` is used.

HideRuler

protoTXView:`HideRuler()`

Hides the ruler if it is currently shown.

IsRulerShown

protoTXView:`IsRulerShown()`

Returns `nil` if the ruler is currently hidden, non-`nil` if the ruler is currently visible.

UpdateRulerInfo

protoTXView:`UpdateRulerInfo(rulerSettings)`

Changes the ruler display settings and updates the ruler display.

rulerSettings A frame with one slot, `'type`. The value of this slot can be either `'metric` or `'inches`.

Pages

This section describes the methods you can use to work with pages in your word-processing view.

GetCountPages

protoTXView:`GetCountPages()`

Returns the number of pages. `GetCountPages` returns `0` if the view is not using a paged layout.

InsertPageBreak

protoTXView:`InsertPageBreak(range)`

Replaces the text inside of the specified range with a page break.

range A range frame, as described in "The Range Frame" (page 3-4).

The page break is a real character with character code `$\u000A`. This character can be copied, pasted, and searched for (with the `FindString` method).

Printing

This section describes the `SetDrawOrigin` method, which you can use to reconfigure your view for printing.

fferent methods that support printing. You can call SetGeometry to reconfigure your view when printing, but it's nicer to keep the current margin settings so the view stays WYSIWYG.

SetDrawOrigin

protoTXView:`SetDrawOrigin(origin):`

Reconfigures a view for printing.

origin	A point frame with the following two slots:	
	x	The horizontal margin, in pixels.
	y	The vertical margin, in pixels.

For paged layouts, the margins are used for each page. The default margin is `(0, 0)`.

Note

You can call `SetGeometry` to reconfigure your view for printing; however, doing so changes your margin settings. If you use `SetDrawOrigin`, your margin settings are not affected. ◆

For example, to print the page number *n* for a view with page height *h* (including margins) in your `PrintNextPageScript`, you can call `SetDrawOrigin` as follows and the draw the view:

```
SetDrawOrigin( {x:0, y:n*h} )
```

protoTXViewFinder

This section describes the methods of `protoTXViewFinder`. You can use the `protoTXViewFinder` to search a `protoTXView` document in a soup without incurring the overhead of creating a word-processing view. This proto allows you to search the data returned by the `Externalize` method of `protoTXView`.

Note

At this time, you cannot use NTK to create a `protoTXViewFinder`; instead, you can create an instance of this proto with the following call:

```
txFinder := {_proto:@827};
```

◆

FindString

protoTXViewFinder`:FindString(`*object*, *str*, *startOffset*, *options*`)`

Searches for matching text in the data object returned by a call to the `protoTXView:Externalize` method.

object	A data object returned from the `Externalize` method.
str	The string to be searched.
startOffset	The offset at which the search should start.
options	Must be `nil`. Currently the search is not case sensitive.

The `FindString` method searches in a word-processing view for a sequence of characters that matches *str*. The search begins at *startOffset* from the beginning of the view and continues until a match it made or the end of the text is reached.

The `FindString` method returns the offset of the matching string in the view. If no match is found, `FindString` returns `nil`.

Note

This method is the same as the `protoTXView:FindString` method, except that the `protoTXViewFinder` version has an additional parameter: *object*. ◆

GetCountCharacters

protoTXViewFinder`:GetCountCharacters()`

Returns the total number of characters.

IMPORTANT

You must call the `protoTXViewFinder:FindString` method before calling `protoTXViewFinder:GetCountCharacters`. ◆

GetRangeText

protoTXViewFinder`:GetRangeText(`*range*`)`

Returns a string containing the characters in *range*.

range	A range frame, as described in "The Range Frame" (page 3-4).

IMPORTANT

You must call the `protoTXViewFinder:FindString` method before calling `protoTXViewFinder:GetRangeText`. ◆

Word Processing View ERS

KeyboardEnhancementsERS

This document covers the expanded toolbox support that is required for the user-level keyboard features for the built-in keyboard on the eMate 300 and the attached keyboard on the MessagePad 2000. This document focuses on new facilities for defining, processing, and displaying keyboard commands and shortcuts. These enhancements include:

- command keys
- return/enter for default actions (default buttons)
- keyboard-based selection and navigation of text
- keyboard-based selection and navigation of menu items
- a context-sensitive popup slip that lists all available key commands

This document does not address hardware issues, nor does it address all application-specific issues; for example, there is a command key proposal for built-in applications at the conclusion of the User Interface section. All features and issues described in this document apply to all forthcoming Newton-based keyboards.

Terminology

This section provides definitions for terms used in this document. There are two terms defined here:

- **Modifier keys** are keys that affect the functioning of the alphanumeric keys. The modifier keys are the shift, command, control, option, and caps lock keys.
- The **key-view** is the view that receives user key strokes. You can get the current key-view by calling the global function `GetKeyView`(). The key-view is the view that owns the caret and is either a `clParagraphView` or a `clEditView`.
- **Command keys** associate a message (an action) with a key combination. The **key combination** consists of a character typed on the keyboard in combination with some number of modifier keys.
- **Keystroke events** are events generated by the system when the user interacts with a keyboard. Keystroke events include the key-up, key-down, and key-repeat events.

User Interface

This section describes the specific user commands that are implemented using the new keyboard facilities for specific applications.

General Usage

The following key combinations are applicable to general usage:

- Pressing the control key in combination with a letter produces the appropriate results, as per the ASCII standard. Built-in apps (anything that uses paragraph views) will ignore control keys, which means that you cannot insert a control-C into a note. However, applications such as a

terminal emulator can make use of them. The control key will not be used for commands.

- Pressing the command key in combination with an alphanumeric key can be used to trigger a system or application-defined command. Commands can also be executed by pressing a combination of command, option, and shift keys along with an alphanumeric key.

Text entry and editing

The following keyboard usage is applicable to text entry and text editing:

- The arrow and tab keys can be used to move between fields.
- The cut (cmd-X), copy (cmd-C), and paste (cmd-V) commands all function as they do on the Macintosh.
- The select-all (cmd-A) command is also available for text editing.

Slips, windows, and buttons:

As described in "Terminology" (page 4-2), the key-view is generally the view that contains the caret and receives and processes keyboard commands. The slip containing the key-view is drawn specially when a keyboard is connected. Figure 5-1 shows how the find slip looks when it is not the key-view.

Figure 5-1 The find slip when it is not the key view

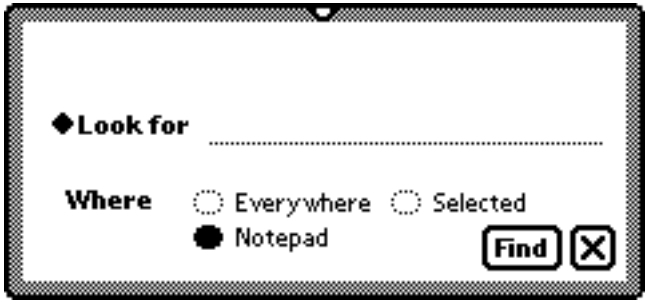


Figure 5-2 shows how the Find slip looks when it is the key view.

Figure 5-2 The Find slip when it is the key view

When the slip is the key-view, its border is thicker, and the "default button," which is triggered by pressing return, is marked with lines above and below the button. These appearance features are only applied when a keyboard is connected. When a keyboard is not connected, the default button looks just like any other button and the slip containing the caret is drawn exactly like other slips.

Note
The default button is highlighted with lines above and below to keeps its width the same whether or not a keyboard is connected. ◆

As soon as the user taps to move the caret, the border and the default button change to match the new caret location. Note that since the caret can be placed in a slip that is not the frontmost slip, the key slip is not necessarily the same as the frontmost slip.

Older applications do not have default buttons. However, the borders of slips drawn in older applications are highlighted as shown above when they contain the caret.

The user can close the frontmost window or slip by pressing the cmd-W key combination, the Cmd-period key combination, or the "Close" key on the

eMate 300 keyboard. This does work with older applications. Also note that, unlike other keyboard commands, the close button is applied to the frontmost slip regardless of whether that slip contains the caret.

Menus

This section describes keyboard usage with Newton menus.

Application-defined keys or key combinations can be used to pop up menus. Some system-wide standards are defined (such as command-N for New and command-R for the routing menu, among others), but application developers can override these definitions if desired.

While a menu is displayed, the user can change the highlighted item by pressing the up-arrow and down-arrow keys. If necessary, the menu will scroll up or down. The right-arrow and left-arrow keys are also supported for menus that contain two-dimensional grids.

When a menu is opened by way of a keyboard command, the first item of the menu is initially highlighted. If the menu is opened in some other way, no item is initially highlighted; in this case, the user can highlight the top item by pressing the down-arrow key, or the user can highlight the bottom item by pressing the up-arrow key.

Pressing the return key selects the highlighted item. This is the same as tapping on that item.

Pressing a letter or sequence of letters “type-selects” menu items, as in the Macintosh Finder and standard file dialogs. If necessary, the menu scrolls to reveal the type-selected item.

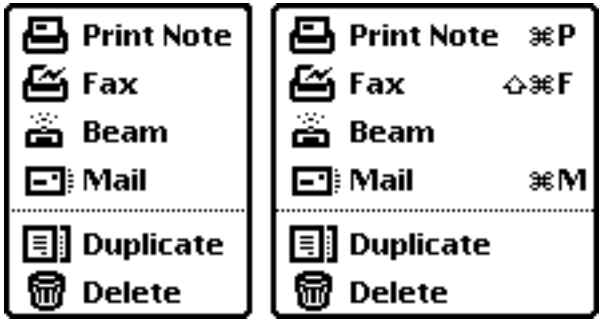
Menu items can have keyboard equivalents that are displayed to the right of the item. These are only displayed when a keyboard is actually connected.

Note

Only printable characters are displayed on menus as key equivalents. ◆

The key equivalents can be used when the receiving application contains the caret or when the menu is open. Figure 5-3 shows the basic appearance of a menu with and without its keyboard equivalents displayed.

Figure 5-3 A menu with and without its keyboard equivalents displayed:



Note
The keyboard equivalents shown in Figure 5-3 are not the actual key-combinations for these commands. ◆

The Command-key Combination Pop-up Help Slip

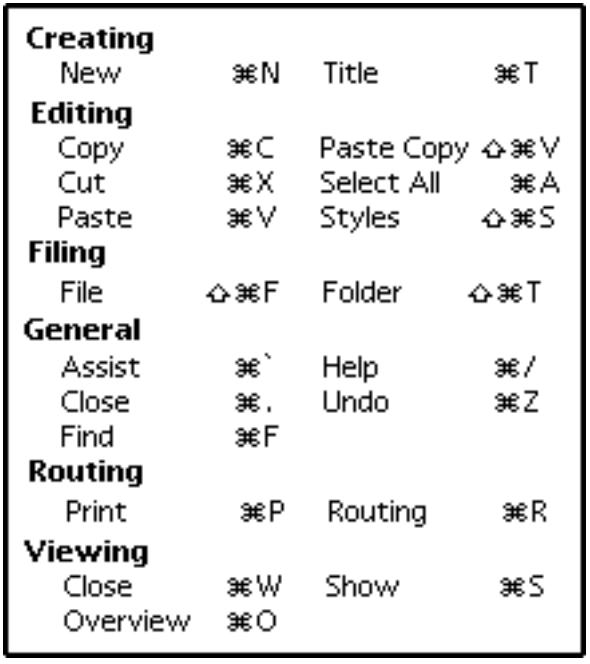
Keyboard equivalents are not displayed for buttons. Instead, a pop-up help slip appears after the user holds down the command key for approximately 1.5 seconds. The pop-up help slip displays all of the valid command keys for the current context.

The pop-up help slip is produced automatically by the system without any direct assistance from the application. The application registers its command key combinations, as described in "The Command-Key Mapping Frame" beginning on page 4-23, and the system constructs the help slip based on which command keys are available.

The pop-up help slip displays the available commands ordered by category. The categories are displayed in alphabetical order. Applications can add to the standard categories or can define new categories, as described in "Command Key Handling" beginning on page 4-22.

Figure 5-4 shows a version of the command-key combination pop-up slip.

Figure 5-4 Command-key combination slip.



The pop-up help slip does not display function key equivalents, which are permanently labeled on the eMate 300 keyboard.

The pop-up help slip closes automatically when the user releases the command key, or when the user presses a key on the keyboard.

If the number of commands defined exceeds the maximum number that can be displayed at once, the help slip gets a scroller on the right. The view can be scrolled using the up-arrow and down-arrow keys (while holding down the command key), or by tapping on the scroller's buttons. During this process, the command key must be held down, or the help slip will close.

Not every keyboard command has to appear in the pop-up help slip. If a command lacks a name ("copy," "find," etc), the command does not appear in the list. This is true of the standard command-arrow combinations for

navigation. If a key-combination command has no category, but does have a name, it is automatically placed in the "Other" category.

Applications can completely override this slip by providing a different help-type slip, or can use the information presented in the default slip in their own way, as described in "Command Key Handling" beginning on page 4-22.

System and Built-in App Command Key Assignments

This section describes the command-key combinations for system-level and built-in application operations. Table 5-1 shows the system-level command-key combinations.

Table 5-1 System-level key assignments

Command-key combination	Behavior
Cmd-A	Selects all text in a note or the current view
Cmd-C	Copies selected text to clipboard
Cmd-E	Opens the title slip
Cmd-F	Opens the Find slip
Cmd-N	Opens the New button popup
Cmd-O	Opens the overview, toggles to close overview
Cmd-P	Opens the print slip
Cmd-R	Opens the routing popup, with first item hilited
Cmd-S	Opens the Show button popup
Cmd-V	Pastes selected text at cursor
Cmd-W	Closes the open window/slip
Cmd-X	Cuts selected text
Cmd-Z	Undo/redo

Table 5-1 System-level key assignments (continued)

Command-key combination	Behavior
Cmd-return	Opens popup
Cmd-` (tilde)	Opens assist
Cmd-. (period)	Cancels action, closes window or slip
Cmd-=	Activates spellcheck
Cmd-shift -F	Opens the filing popup, tab/arrow supported
Cmd-shift S	Opens the styles slip
Cmd-shift T	Opens the folder tab
Cmd-?	Opens help file
Cmd-up/ down arrows	Scrolls up and down, except in the Word Processor, in which it scrolls to the beginning or end of the document.
up/down arrows	Moves highlight up or down in overview or popup
Return	Selects highlighted item from overview or popup

Notes

The system-level key assignments do not include combinations for accessing Prefs, the Info button, or the Edit Folders button.

When local scrollers are present, the Cmd-up-arrow and Cmd-down-arrow key combinations can affect the local scrollers. ◆

Command-key Assignments for The NotePad Application

Table 5-2 shows the command-key assignments for the Notepad checklist and outline stationery.

Table 5-2 Notepad checklist and outline stationery command keys

Command Keys	Behavior
Cmd-]	Creates a new right bulleted item
Cmd- [	Creates a new left bulleted item
Cmd-return	Checks/unchecks an item
Cmd-=	Creates a bulleted item (same level)

Note
There is no support for demoting/promoting or expanding/collapsing already created items. The user can only perform these operations with the stylus. ◆

Command-Key Assignments for The Names Application

Table 5-3 shows the command-key assignments for the Names application.

Table 5-3 Names application command keys

Command Keys	Behavior
Cmd- +	Opens the Add popup. The user can choose the menu item with the arrow and return keys.

Note
New users will often press Return or Enter after they fill in the first of the entry screens, rather than the preferred Cmd-W. ◆

Command-Key Assignments for The Dates Application

Table 5-4 shows the command key assignments for the Dates application.

Table 5-4 Dates application command keys

Command Keys	Behavior
Cmd-+	Opens the Add popup. The user can choose the menu picker item with the arrow and return keys.

Command-Key Assignments for The In/Out Box

Table 5-5 shows the command key assignments for the In/Out box

Table 5-5 In/Out box command keys

Command Keys	Behavior
Cmd-left-arrow	Opens In Box
Cmd-right-arrow	Opens Out Box
Cmd-E	Taps the send button when in an item; taps the send or receive when in box view
Cmd-G	Taps the tag button

Command-Key Assignments for The Call Log

Table 5-6 shows the command-key assignments for the call log.

Table 5-6 Call log command keys

Command Keys	Behavior
Cmd- +	Taps the Add to Names button
Cmd-D	Taps the Call button
Cmd-H	Taps the Hang-up button; arrows then navigate popup

Command-Key Assignments for the BookPlayer

Table 5-7 shows the command-key assignments for the BookPlayer application.

Table 5-7 BookPlayer command keys

Command Keys	Behavior
Cmd- B	Taps the Bookmark button
Cmd-G	Taps the Page Number button
Cmd-M	Taps the Markup button

Keyboard Reference

This section describes the functions, methods, and data structures for keyboard handling in Newton applications. There are two main areas of keyboard handling that you need to understand:

- How to handle keystrokes from a keyboard, as described in "Keystroke Handling" beginning on page 4-13.
- How to work with command keys, as described in "Command Key Handling" beginning on page 4-22.

Keystroke Handling

This section describes the functions and methods that you can use to handle keystrokes in your Newton applications.

You can handle keystrokes in an application in the following ways:

- You can define keyboard commands that the system will match and execute where appropriate. This is the usual mechanism for defining keyboard equivalents and is described in detail later in this section.
- When you need to apply custom handling to keyboard events, you can intercept those events, including the key-down, key-up, and key-repeat events.
- Sometimes the system groups multiple keystrokes together into "keyStrings" in order to improve performance. You can also intercept these strings, which were implemented to improve user response time. Note that keystrokes in all views are grouped together into strings unless you set the `vSingleKeystrokes` text flag in the view.

  For example, if the user quickly types the word "something" into a long paragraph view, each change to the view's contents results in an insertion and redisplay. Grouping the keys into a single insertion and redisplay operation produces a much faster response than responding individually to the entry of each character.

Table 5-8 shows the functions and methods that you can use to handle keystrokes.

Table 5-8 Summary of keyboard methods and functions

Function/Method	Description
`IsCommandKeystroke`	Determines if a keystroke is a command key combination.
`ViewKeyDownScript`	Sent to the key-view when the user presses down on a key.
`ViewKeyUpScript`	Sent to the key-view when the user releases a key.
`ViewKeyStringScript`	Sent to the key-view when a group of keystrokes needs to be processed (when single keystroke handling is not applied to the key-view).
`ViewKeyRepeatScript`	Sent to the key-view while the user holds a key down.

Intercepting Keystrokes Directly

To intercept keystrokes directly, you need to respond to the key-down, key-repeat, and key-up events. To guarantee that the scripts for these keystroke events are called in your view, you must set the `vSingleKeystrokes` text flag. This is true regardless of the view class of your view.

Note
Setting the `vSingleKeystrokes` text flag in a paragraph view results in a substantial reduction in typing performance for the user. This is because the system processes each keystroke individually, rather than batching a set of keystrokes into a string, as described in "Intercepting Grouped Keystrokes" (page 4-19). ◆

The system software calls a script for each of the keyboard events. To understand how these scripts are invoked, read the next section, "Keystroke Event Sequencing" beginning on page 4-17. If the script returns true (to indicate that the system should continue processing the key event), the system next checks the key to determine if it is a command key. If the event is not a command key, the system hands the key event to the appropriate view for default handling.

There is one global function for keystroke handling and four keystroke event scripts. One of the event scripts, `ViewKeyStringScript`, is described in "Intercepting Grouped Keystrokes" (page 4-19). The other three are described here.

IsCommandKeystroke

`IsCommandKeystroke(`*char*, *flags*`)`

Returns `true` if the keystroke is a command-key combination, and `nil` if not.

char The character that was entered on the keyboard. Note that if a modifier key is the only key pressed (for example, the Shift key), this value will be `0`.

flags A 30-bit integer that specifies which modifier keys were pressed and other additional information. The use of the individual bits in this value are shown in Table 5-9.

ViewKeyDownScript

`ViewKeyDownScript(`*char*, *flags*`)`

Is sent by the system when a user presses down on a keyboard key.

char The character that was entered on the keyboard. Note that if a modifier key is the only key pressed (for example, the Shift key), this value will be `0`.

flags A 30-bit integer that specifies which modifier keys were pressed and other additional information. The use of the individual bits in this value are shown in Table 5-9.

Your implementation must return `nil` if you want the system to continue processing the keystroke.

Table 5-9 shows how the bits in the *flags* parameter are used for the key event scripts.

Table 5-9 Key event-processing script flags

Bits	Description
0 to 7	The keycode.
8 to 23	The 16-bit character that would be inserted if none of the modifier keys were pressed.
24	Indicates whether the key was delivered from an on-screen keyboard. (`kIsSoftKeyboard`)
25	Indicates that the Command key was down. (`kCommandModifier`)
26	Indicates that the Shift key was down. (`kShiftModifier`)
27	Indicates that the Caps Lock key was down. (`kCapsLockModifier`)
28	Indicates that the Option key was down. (`kOptionsModifier`)
29	Indicates that the Control key was down. (`kControlModifier`)

ViewKeyUpScript

`ViewKeyUpScript(`*char*, *flags*`)`

Is sent by the system when the user releases a keyboard key.

char — The character that was entered on the keyboard. Note that if a modifier key is the only key pressed (for example, the Shift key), this value will be `0`.

flags — A 30-bit integer that specifies which modifier keys were pressed and other additional information. The use of the individual bits in this value are shown in Table 5-9 (page 4-16).

Your implementation must return `nil` if you want the system to continue processing the keystroke.

ViewKeyRepeatScript

`ViewKeyRepeatScript(`*char*, *flags*`)`

Is sent by the system repeatedly while the user holds down a keyboard key.

char	The character that was entered on the keyboard. Note that if a modifier key is the only key pressed (for example, the Shift key), this value will be `0`.
flags	A 30-bit integer that specifies which modifier keys were pressed and other additional information. The use of the individual bits in this value are shown in Table 5-9 (page 4-16).

Your implementation must return `nil` if you want the system to continue processing the keystroke.

Keystroke Event Sequencing

This section describes the sequence of events that is generated when the user presses, holds down, and then releases a key on the keyboard.

Key-down Events

The following list describes the processing sequence when the user presses down on a keyboard key and a key-down event is generated:

1. If the `vSingleKeystrokes` text flag is set in the key-view, the view system looks for a `ViewKeyDownScript` in the key-view (proto inheritance only) and calls it. Note that the `ViewKeyDownScript` is also called when there are no other pending, unprocessed keystrokes.
2. If the `ViewKeyDownScript` returns a non-`nil` value, handling for the key-down event is complete.
3. Otherwise, the system checks for a command key. This is described in detail in "Command Key Handling" beginning on page 4-22.

4. If the key is not a command key, the default view class handling occurs. If the key-view is a `clEditView`, a new paragraph is created at the caret location. If the key-view is a `clParagraphView`, the default handling is to insert the appropriate character at the caret, unless the key is a backspace or arrow key, in which case the expected action occurs.

Key-repeat Events

The following list describes the processing sequence when the user holds down on a keyboard key and key-repeat events are generated:

1. After a brief delay, the system starts issuing key-repeat events.
2. For each key-repeat event, if `vSingleKeystrokes` is set in the key-view, the `ViewKeyRepeatScript` (proto inheritance only) is called.
3. If the view does not have a `ViewKeyRepeatScript`, the `ViewKeyDownScript` is called instead.
4. If this script returns non-`nil`, the repeated keystroke is considered to have been handled.
5. Otherwise, the system checks for a command key , as described later in this section. Commands can specify whether or not they are executed with repeated keys.
6. If the key is not a command key, the default view class handling occurs.

Key-release Events

The following list describes the processing sequence when the user releases a keyboard key and a key-up event is generated:

1. If `vSingleKeystrokes` is set in the key-view, the view system the `ViewKeyUpScript` (proto inheritance only) is called.
2. If the `ViewKeyUpScript` returns a non-`nil` value, the key-up event handling is complete.
3. No command check occurs with key-up events.
4. Otherwise, the default viewClass handling occurs. Ordinarily, this is nothing at all — all characters are inserted at key-down time, and arrows and tabs are handled then as well. The exception is the backspace key

when the last paragraph of a paragraph has been deleted — the default key-up handler for a `clParagraphView` will remove the view from its parent if that view is inside a `clEditView`.

Intercepting Grouped Keystrokes

If you have not set the `vSingleKeystrokes` flag for the key-view, the system groups together a set of keystrokes for batch processing and sends the `ViewKeyStringScript` message when a group of keystrokes is ready to be handled.

ViewKeyStringScript

`ViewKeyStringScript(`*string*`)`

Is sent by the system when a batched group of keystrokes is ready to be processed.

string	The batched string of characters as a null-terminated string. These are not keycodes.

You must return `true` if your script handles the *string* and `nil` if not.

Note that function keys and command-key combinations never appear in *string*. These keys are always processed individually.

Typing Without a Caret

The key-view must be established before a keystroke is posted. This means that when the user types and there is not an active caret (the key-view is `nil`), the system has to set the key-view. However, different actions need to be taken, depending on whether the keystroke is a command key or an insertable character key.

When a key is pressed and the key-view is `nil`, the system looks for the frontmost view that can handle it. This view may vary, depending on whether or not the key pressed is a command key.

Non read-only paragraph views, edit views, and text editing views all accept both command keys and normal keys (for insertion). For other views, there

are two additional textFlags that you can use to specify the kinds of keystrokes you want to handle, as shown in Table 5-10.

Table 5-10 Text flags to specify the kind of keystrokes a view accepts

Text Flag	Description
`vTakesCommandKeys`	The view accepts command keys.
`vTakesAllKeys`	The view accepts all keys, including command keys.

Note:
You do not use the text flags shown in Table 5-10 for edit views, paragraph views, or textEdit views, all of which always act as if both flags are on (as long as they are not read-only views). ◆

These flags are also significant when a normal key is typed. If a normal key is typed when the key-view accepts only command keys, the key-view is switched to the frontmost view that accepts normal keys.

You can determine which view is the frontmost view that accepts normal keystroke by calling `GetView` as follows:

```
view := GetView('viewfrontkey);
```

You can determine which view is the frontmost view that accepts command keystrokes by calling `GetView` as follows:

```
view := GetView('viewfrontcommandkey);
```

The Caret Stack and Caret Activation

The system maintains a stack of key-views, which allows the current key-view to be reset to the previous one when a key-view is closed. For example, when the user opens the Find slip while the caret is in the notepad, the caret is moved from the notepad to the input line in the Find slip. Then, when the user closes the find slip, the caret is returned to the notepad in its former location.

The system attempts to preserve selections in this process. If the user selects a word in the notepad, then opens the find slip and closes it, the word in the notepad will be re-selected.

The caret stack mechanism is largely invisible to applications. When a view becomes the key-view (either through a user action or through restoration from the caret stack) or when a view loses the key-view, the following message is sent to the view that is losing the caret:

```
oldkeyview:ViewCaretActivateScript(nil);
```

Immediately thereafter, the following message is sent to the view that is getting the caret:

```
newkeyview:ViewCaretActivateScript(true);
```

You can use the `ViewCaretActivateScript` to trigger actions when your view becomes the key-view or is no longer the key-view. The return value is ignored.

Obtaining The State Of a Key

You can use the `IsKeyDown` function to determine if a specified key is down on the keyboard.

IsKeyDown

`isDown := IsKeyDown(`*keyCode*, *isHardKeyboard*`);`

Returns true if the specified key is down on the keyboard.

keyCode	The keycode that you want to test.
isHardKeyboard	`True` if you want the hardware keyboard tested. A value of `nil` means that the on-screen keyboard is tested.

This function works for both on-screen and hardware keyboards.

Note that the system maintains two separate key maps, one for all on-screen keyboards, and one for the connected hardware keyboard.

Simulating Keystrokes

You can use the `HandleKeyEvents` function to post key events and have the system handle them as if they had been typed on a hardware keyboard. This is useful for testing purposes, or for playing back keyboard macros.

HandleKeyEvents

`HandleKeyEvents(`*keyEvents*`);`

Posts key events as if they were typed on a hardware keyboard.

keyEvents	An array of integers. Each integer specifies a single key-down or key-up event. The least significant seven bits of each integer specificy a keyCode value, and the eighth bit indicates whether or not the event is a key-down event. Add 128 to the key code value to simulate a key-down event.

The state of the hardware keyboard (its keymap) is saved and restored before and after the events are handled so that inconsistencies are avoided (i.e., if the shift key is down on the actual keyboard, it had better be down in the hardware keyboard's `keyMap`).

Note

You cannot simulate key-repeat events with the `HandleKeyEvents` function. ◆

Command Key Handling

This section describes the data structures, methods, and functions that you can use to work with command keys in your Newton applications. You define key commands in `keyCommand` frames. Each `keyCommand` frame associates a key combination with a message and other information.

Each view in your application has a set of key commands associated with it. You can use the functions and methods shown in Table 5-11 to work with key commands.

Table 5-11 Summary of command key methods and functions

Function/Method	Description
`view:AddKeyCommand`	Adds a key command to a view.
`view:AddKeyCommands`	Adds an array of key commands to a view.
`view:BlockKeyCommand`	Blocks a key command from being associated with a view.
`view:ClearKeyCommands`	Removes all key commands from a view.
`SendKeyMessage`	Sends a key message as if a key command had been typed on the keyboard.
`FindKeyCommand`	Finds the key command that matches a command-key combination.
`GatherKeyCommands`	Returns an array of the command keys associated with a view.

The Command-Key Mapping Frame

The mapping between keystrokes and commands is defined by `keyCommand` frames, which are used for the following purposes:

- keyboard command dispatch and execution
- menu display
- display on the key equivalent help slip

The `keyCommand` frame contains five slots, as shown here:

```
keyCommand := {
    char:           $a,
    modifiers:      kCommandModifier,
    keyMessage:     '_SelectAll,
    name:           "Select All",
```

```
    category:       "Editing"
};
```

Slot descriptions

char The unmodified character of the keypress. Required.

modifiers The required modifiers. This slot can be absent or nil, in which case no modifiers are required. See below for additional details on specifying key commands that do not require the command key to be held down.

This slot can also be used to specify other flags related to the command:

`kRepeatable`
: The command is to be executed on key-repeat events as well as key-down events.

`kWorksInAllModals`
: Only applies to system-wide (root view) commands. When set, the command is available in modal dialogs.

`kWorksInAppModals`
: Only applies to system-wide (root view) commands. When set, the command is modal dialogs whose `vApplication` bit is set.

keyMessage A symbol. Required. This is the message that is sent when the `keyCommand` is matched. You must supply a method of this name that takes a single parameter (the current key-view) somewhere in the key-view chain. The method is called when the system matches the key command.

name A string. The name of the command that appears on menus and the command key pop-up help slip. If this slot is `nil` or absent, the key equivalent is not displayed on the pop-up help slip.

category A string. The name of the category to which the command belongs on the command key pop-up help

slip. If this slot is absent or nil, but there is a *name* slot, the command is placed in the "Other" category on the command key pop-up help slip.

showChar — A character. Optional. If present, this character is shown on popup menus and in the pop-up help slip instead of the character in the *char* slot. This is useful for presenting a more user-friendly key combination to the user than the actual combination. For example, you can define the Cmd-/ combination and present it as Cmd-? by defining this slot with the '?' character.

Table 5-12 shows the key codes for special (non-printing) keyboard keys. You can use these values in the *char* slot of your `keyCommand` frame.

Table 5-12 Key codes for special keys

Constant	Value
`kTabKey`	`$\u0009`
`kBackspaceKey`	`$\u0008`
`kReturnKey`	`$\u000D`
`kEnterKey`	`$\u0003`
`kEscKey`	`$\u001B`
`kLeftArrowKey`	`$\u001C`
`kRightArrowKey`	`$\u001D`
`kUpArrowKey`	`$\u001E`
`kDownArrowKey`	`$\u001F`
`kF1Key`	`$\uF721`
`kF2Key`	`$\uF722`
`kF3Key`	`$\uF723`
`kF4Key`	`$\uF724`
`kF5Key`	`$\uF725`

Table 5-12 Key codes for special keys (continued)

Constant	Value
kF6Key	$\uF726
kF7Key	$\uF727
kF8Key	$\uF728
kF9Key	$\uF729
kF10Key	$\uF72A
kF11Key	$\uF72B
kF12Key	$\uF72C
kF13Key	$\uF72D
kF14Key	$\uF72E
kF15Key	$\uF72F

Note
The function keys (KF1Key through kF15Key) are only available on the eMate 300 keyboard. ◆

Searching for Key Commands

If your viewKeyDownScript or viewKeyRepeatScript returns nil, the system tests for a key command by searching for a keyCommand frame that matches the entered key(s). By default, the system searches for a keyCommand frame when the key pressed is one of the following:

- a function key
- the escape key
- any key pressed while the command key is held down

However, it is also possible to force the system to search for keyCommands with every keystroke, regardless of whether the command key is down. To do this, set the text flag vAlwaysTryKeyCommands in the key-view.

The following list describes how the system searches for a `keyCommand` frame when the user enters a potential command key:

1. The command search starts at the view that owns the caret. See below for a description of what happens when the key-view is `nil`.
2. The system looks in the `_keyCommands` slot of the current view for a `keyCommand` that matches the pressed key.
3. If the system finds a match, the search is complete.
4. Otherwise, the system looks for a slot named `_nextKeyView` in the current view. If this slot is present, its contents are used as the next view in which to search.
5. If the `_nextKeyView` slot is not found, the system moves up to the current view's parent and uses that as the next view to search. This continues until the command is found or the root view has been searched.

The search for a command key is analagous to the parent inheritance chain. You can link a slip to your base view, rather than its parent, which is normally the root view. This allows a key command defined in your base view to be available in the slip.

Key commands can be global (available regardless of the context), specific to a certain application, or specific to a slip within an application. Some commands may even be specific to a certain input field of a certain slip. In any case, each key command is associated with a certain view. Table 5-13 shows how commands are associated with views.

Table 5-13 Command definition views

Command Type	Associated view
Global commands	Root view
Application commands	Application's base view
Slip commands	Slip's base view
Field commands	Field's view

KeyMessage Definition and Invocation

You specify the method associated with a command key in the `keyMessage` slot of the `keyCommand` frame. The method need not be implemented in the same view as the command key.

Once the system matches a `keyCommand`, the system searches the same chain (starting at the key-view and follwing either `'_parent` or `'_nextKeyView` slots) until the method has been found. The method is called with a single parameter: the current key-view. For example:

```
view:_SelectAll(currentKeyView)
```

You can examine the current key-view in your implementation to decide which actions you want to take.

Adding, Blocking, and Removing Key Commands

You can use the methods described in this section to associate a command key with a specific view or to remove (temporarily or permanently) a command key from a view.

Note that the most recently added key command for a specific key combination takes precedence. If, for example, your application defines a command-F equivalent and adds it using `AddKeyCommand`, and then adds another command-F equivalent, the last one added will be the only one seen by the system.

AddKeyCommand

`view:AddKeyCommand(`*keyCommandFrame*`);`

Associates a key command with the view.

keyCommandFrame

A key command frame, as described in "The Command-Key Mapping Frame" (page 4-23).

You can call this method from your `viewSetupDoneScript`.

AddKeyCommands

```
view:AddKeyCommands(arrayOfKeyCommandFrames);
```

Associates a collection of key commands with the view.

arrayOfKeyCommandFrames

An array of key command frames, as described in "The Command-Key Mapping Frame" (page 4-23).

You can call this method from your `viewSetupDoneScript`.

This method is efficient for adding an array of `keyCommands` at once. A minimum of cloning is performed; in the case that a view already has one or more `keyCommands` defined, some cloning is performed.

BlockKeyCommand

```
view:BlockKeyCommand(keyMessageSymbol);
```

Hides a key command that would ordinarily be accessible in the view.

keyMessageSymbol

A symbol that names the command message. This must be the same message as you specified in the `keyMessage` slot of the `keyCommand` frame.

The `BlockKeyCommand` method makes any key command that matches *keyMessageSymbol* unavailable from the view. The key command no longer displays on the command key pop-up help slip in the view.

IMPORTANT

Calling the `AddKeyCommand()` and `BlockKeyCommand` methods grow a RAM-based array, so you must be careful to not overuse these methods. ◆

ClearKeyCommands

```
view:ClearKeyCommands();
```

Removes all key commands from the view.

The `ClearKeyCommands` method removes all key commands that are defined in the view. This method does not, however, remove key commands that are available in the view but defined elsewhere.

Looking Up KeyCommands

This section describes several functions you can use to work with the key command system software.

SendKeyMessage

`SendKeyMessage(`*keyView*, *keyMessage*`);`

Sends a message to a view as if the user had typed a key command.

keyView	The view to which the message gets sent.
keyMessage	A symbol that names the command message. This must be the same message as you specified in the `keyMessage` slot of the `keyCommand` frame.

The `SendKeyMessage` function sends the message using the same lookup rules that are used when a key command is being handled by the system. This function always returns `nil`.

FindKeyCommand

`keyCommand FindKeyCommand(`*startView*, *char*, *flags*`);`

Searches for and returns the key command frame that matches a key combination.

startView	The view in which to start searching for the command key.
char	The command key character. Note that if a modifier key is the only key pressed (for example, the Shift key), this value will be `0`.
flags	A 30-bit integer that specifies which modifier keys were pressed and other additional information. The use of the individual bits in this value are shown in Table 5-9 (page 4-16).

The `FindKeyCommand` function function starts at the view *startView* and looks for a `keyCommand` frame that matches the keypress described by *key* and *flags*. The function returns a matching `keyCommand` or `nil` if none was found.

FindKeyCommand

```
keyCommandArray GatherKeyCommands(startView);
```

Returns an array of all key commands available in the view.

startView The view in which you are interested.

This function returns an array of all the key commands available to the view *startView*.

CategorizeKeyCommands

```
keyCategories CategorizeKeyCommands(keyCommandArray);
```

Categorizes an array of key command frames.

keyCommandArray An array of `keyCommand` frames.

This function sorts the `keyCommandArray` by category. Within each category, the `keyCommand` frames are sorted by name. `CategorizeKeyCommands` returns an array of frames that describe each category and its key commands. For example:

```
{ category: "myName", keyCommands: [ kc1, kc2, kc3...] }
```

You can use this function to create your own popup command key help slip.

Displaying the Pop-up Command Key Help Slip

When the user has held down the command key for a certain period of time (1.5 seconds), the system makes the following call:

```
SendKeyMessage(keyview, '__keyHelpOpenScript);
```

When the command key is released, the following call is made:

```
SendKeyMessage(keyview, '__keyHelpCloseScript);
```

The standard implementation of this command, which is in the `rootView`, dynamically builds and displays the standard “Keyboard Commands” pop-up help slip, according to the available keyCommands. You can provide your own versions of these scripts to modify or override the default pop-up help slip.

Faking a Button Press

A common use of a `keyCommand` is to tap an on-screen button. To perform this action, call the `PressButton` function.

PressButton

`PressButton(`*buttonView*`);`

Causes the button to act as if it had been tapped by the user.

buttonView The button you want tapped.

The `ViewClickScript` in *buttonView* is not called and thus does not need to be defined. All other button-related scripts are called as if the button had been tapped with the pen.

Designating the Default Button In a Slip

In order to designate a button as a view's default button, you need to set a the `'_defaultButton` slot in the slip. This slot must contain a reference to the view that is the default button, which you can do by declaring the button view as `'_defaultButton` in the slip). The system automatically applies the highlighting graphical treatment to the default button.

You need to use a view that protos to the new `protoContainerView`. This allows the button to be tapped when the user presses the Return key. Note that protoApplication, `protoDragger`, and many other built-in protos are based on `protoContainerView`.

Paragraph views that have `oneLineOnly` viewJustification automatically send the keyMessage `'_DoDefaultButton`, which results in the default button being tapped. `protoContainerView` does the same thing in a `ViewKeyDownScript`, and implements the `_DoDefaultButton` method, which calls `PressButton()` for the view declared as `'_defaultButton`.

Designating a Slip's Close Box

You need to let the system know which button is the close button in a slip by declaring the button in the slip as `'_closeBox`. All of the supplied close box protos do this automatically; if you implement your own close box, you need to ensure that the button is declared properly in the slip.

When a slip is closed via the keyboard, the system simulates a tap on the close box (in the same way the default button is pressed when the user presses return).

Default and Close Buttons in Confirm Slips

Four new default button lists are now available for use in confirm calls. This makes it easy to add keyboard support to all of your confirm slips. Table 5-14 shows the new default button lists.

Table 5-14 New default button lists

Button list	Default	Other
`'okCancel`	Ok	Cancel means close
`'okCancelDefaultCancel`	Cancel	No keyboard equivalent for OK
`'yesNoDefaultYes`	Yes	No means close
`'yesNoDefaultNo`	No	No keyboard equivalent for Yes

If you are creating your own button list, you can add a slot to the `buttonFrame` named `'buttonValue` (**NOTE**: this might be `'keyValue`, not `'buttonValue`). The value of this slot can be either `nil`, `'default`, or `'close`. The confirm slip will associate the appropriate keystroke with each button value. A value of `nil` means no key association.

A final note: the root version of the `Confirm` method (`:Confirm()`) previously used the `'okCancel` button list; it now uses the `'okCancelDefaultOk` button list instead.

KeyCommands and Popup Menus

As described in “Menus” beginning on page 4-5, command key equivalents are shown on popup menus. Recall that a popup menu is defined using an array of items. For example:

```
[   "one",
    {icon: i, item: "two"},
    {icon: ii, mark: $-, item: "three"} ]
```

This array, which could be passed to `:PopupMenu`, defines a three-item menu. The second item contains an icon, and the third contains an icon and a "mark."

In a similar way, items in popups can map to `keyCommands`. You specify the `keyMessage` for a given item, and when the menu is displayed, the system performs a `keyCommand` lookup, starting at the button that was pressed to invoke the menu. When the system finds a matching `keyCommand`, it uses the name, character, and required modifiers from the `keyCommand` frame. For example:

```
[   "one",
    {keyMessage: _DoSomething},
    {keyMessage: _AnotherThing} ]
```

In the above example, the names for the second and third items will come from the corresponding `keyCommands`. An item that is defined using a `keyMessage` slot can still have `'icon` and `'mark` slots.

If you want to use a name other than the name specified in the `keyCommand` frame, you can specify a `keyMessage` and an `item` slot in the item frame. For example:

```
{keyMessage: _DoSomething, item: "something"}
```

In this case, the name displayed on the menu will be "something," regardless of the contents of the `keyCommand` that matched the `_DoSomething` keyMessage.

Note

Grid items are not permitted to have key equivalents. ◆

Normally, when a user choses a menu item that is defined through a `keyMessage`, the action is triggered by sending the `keyMessage` to the button's view and following the key-view chain (`_parent`/`_nextKeyView`) rather than through the normal process of calling the `PickActionScript`. If you want the `PickActionScript` to be called regardless of how your menu items are defined, you need to define the following slot in your button:

```
alwaysCallPickActionScript: true,
```

This forces the `PickActionScript` to be called for any menu item.

Compatibility

This section documents keyboard-related compatibility issues for older applications.

Default Buttons

Default buttons in slips will only appear and function in applications which were designed with them in mind.

Possible Key-view Compatibility Problem

Prior to this release, you could not set the key-view to anything other than a `clParagraphView` (using `Setkeyview`) or a `clEditView` (using `SetCaretInfo`). In this latest release, you can designate any view as a key-view. This could be a problem for some older applications.

For example, the following code fragment worked fine for older versions of the system software, but is now erroneous.

```
kv := GetKeyView();
if band(kv.viewClass,0x7FFF) = clParagraphView then
      // do something specific for a paragraph view
else
      // no good: cannot assume the view is an editView
```

Newton OS 2.1 Grayscale Imaging ERS

The implementation of imaging in Newon OS 2.1 allows for 1, 2, and 4 bit output on systems that provide appropriate hardware support, allowing developers to write applications that

- draw gray shapes, text, and patterns.
- display color PICTs as grayscale images on the Newton screen.

The imaging system also supports anti-aliasing for pixelmaps (i.e. received faxes). The imaging system does not support anti-aliasing for lines, curves, or text.

The use of grayscale imaging on hardware that predates the MessagePad 2000 or eMate 300 is not supported.

Overview of Features

The software buttons, icons, and other pictures used by the system are stored as monochrome pictures. The soft button bar and Extras Drawer now

contain some gray icons. Gray is also used in the card slot icons when opening the card icon, and in the eMate 300 Word Processor and Draw stationery. (Specifically, in scrollers.)

If an application contains color pictures, the RGB values contained within the image are not modified. However, if the user creates a new picture from the original, the new picture is defined in terms of grayscale values.

For example, the user views color images in a Newton application on his desktop computer. He then downloads the application, including the color image, to the Newton. When viewing the image on the LCD, the image is converted from RGB values to gray tones. If the user then resizes the image, and writes a new picture out to the store, gray tones are written out, not RGB values. If the user subsequently uploads the new resized image to the desktop, the user will see the original color picture and a new gray resized picture.

PICTs are highlighted as a "photographic negative." Ink and ink text are monochrome.

Terminology

PICT	A Macintosh file structure containing a series of opcodes and related data.
bitmap	A data structure containing several descriptive fields followed by raster data.
multi-bit depth	The bit depth of an image determines how many tones can be represented by each pixel. The number of tones is 2 raised to the power of the bit depth. For example, a 4 bit depth results in 16 possible tones, including white and black.
direct pixel format	A format for PixMaps where each pixel is composed of an RGB triplet. This pixel format can be 16 bit (5 bits for

each red, green, and blue value, and one unused bit) or 32 bit (8 bits for each value, and 8 unused bits).

indexed pixel format
: A format for PixMaps where each pixel is an index into a color table of RGB triplets. This pixel format supports images with 1, 2, 4, or 8 bit depths.

RGB to gray conversion
: The following luminance weights are used to convert RGB values to gray tones:

```
gray =    (.30 * red value) +
          (.59 * green value) +
          (.11 blue value)
```

antialiasing
: A technique that uses gray tones in the areas of transition between white and black as a means of avoiding the jagged lines caused by aliasing in 1-bit images.

Developer API

This section describes the developer interface to grayscale imaging; it assumes familiarity with the contents of the “Drawing and Graphics” chapter of *Newton Programmer’s Guide*.

Color definitions in graphic shapes' style frame

The `penPattern` and `fillPattern` slots in the styles frame accept a constant representing 4-bit gray tones ranging from black to white:

```
kRGB_Gray0
kRGB_Gray1
kRGB_Gray2
...
kRGB_Gray15
```

kRGB_Gray0 is black, and is also defined by the kRGB_Black constant. kRGB_Gray15 is white, and is also defined by the kRGB_White constant.

You can use the new 'graypattern class to define a gray pattern in a binary object. The pattern can be as large as 8 pixels high by 8 pixels wide, with each pixel represented by an RGB triplet contained within the binary object. Each component of the triplet is a 16-bit value ranging from 0 to 65535. In the event the developer defines less than the number of bytes needed for an 8 x 8 pattern, the pattern triplets are repeated to complete the pattern.

For example, a fillPattern in the styles frame can be defined as in the following code fragment:

```
fillPattern: SetClass("\u999999999999555555555555",
                          'grayPattern)}
```

If less than 8 RGB triplets are defined, the defined triplets are repeated until a full 8 pixel line is completed. This line is then duplicated an additional 8 times to complete the 8 x 8 pattern.

If more than 8 RGB triplets are defined, the triplets are duplicated using a resolution of 8 pixel multiples. In other words, if 16 triplets are defined, the 16 triplets are duplicated 4 times to complete the pattern. If 15 triplets are defined, the last 7 triplets are ignored, and the first 8 triplets are duplicated 8 times to complete the pattern.

Dithered Patterns

You can define a dithered pattern as a frame having the following slots:

```
{class:'ditherPattern,
pattern: vfBlack,   // B/W QD pattern const (vfBlack, vfGray, etc.)
                    // or
          binaryObj,// Binary object of type 'pattern (BW 8x8 pattern)
foreground: kRGB_Gray0, //  kRGB_Gray0 thru kRGB_Gray15
background: kRGB_Gray15, //  kRGB_Gray0 thru kRGB_Gray15
}
```

This pattern uses the BW pattern with the "black" bits drawn in the foreground color and the "white" bits drawn in the background color. This

allows for a wide range of pattern possibilities without taking up much space.

Text shapes

The new `'textPattern` slot in the style frame is used to render text shapes in gray tones. This slot makes it possible to use the same style frame for an array containing both non-text shapes and text rendered with different tones.

Text is drawn with a single shade of gray. If a `'grayPattern` binary object defines a complex pattern using more than one shade of gray, the results are undefined.

Color PICTs and bitmaps

Color PICTs can be imported through the use of a variation on the `GetPictAsBits` function. The format of the PICT files must be indexed pixels with 1, 2, 4, or 8 bit depth.

The existing `GetPictAsBits` function converts the PICT to a bitmap frame containing the following slots:

`bounds`	A rectangle describing the bounds of the image.
`bits`	A binary object containing the image data. The binary data is in the form of a "FramBitmap": bytes 0-3 unused baseAddr bytes 4-5 rowBytes bytes 6-7 unused pad generated by ARM compiler bytes 8-11 bounds bytes 12-end image data
`mask`	An optional slot containing a binary object in the same format as `bits`. The data is a mask that is drawn with modeBic.

The new function provides a frame containing the following slots:

`bounds`	A rectangle describing the bounds of the image.
`bits`	Optional. A binary object containing 1-bit image data. May be nil if the image has multiple bit depths.
`mask`	Optional. A 1-bit mask that is drawn with modeBic.
`colorData`	Optional. A frame or an array of frames. This slot allow you to provide images that are customized for different bit depths. Each frame contains the parameters and data for a particular bit depth, and has the following slots:

`cBits`	A binary object containing color image data. The binary data has the same format as the bits, although the image data will be indices into a color table.If this slot exists, the bits slot should be set to nil.
`bitDepth`	The bitDepth of the image. Valid values are 1, 2, 4, and 8. An invalid bit depth will throw an exception. If this slot is empty, a 1 bit depth is assumed.
`colorTable`	A table representing the RGB triplets, conforming to the format described in “Color definitions in graphic shapes' style frame” (page 5-3). If the table is not defined, a default table is used by the system that has 2 *bitDepth* colors, ranging from white to black.

You must use the GetNamedResource global function to load 16- or 32-bit PICT files.

Manipulating Bitmap Data

The `GrayShrink` function performs scaling, gray tone conversion, and antialiasing operations on monochrome bitmap data.

GrayShrink

`GrayShrink(`*bitmap*`, `*style*`);`

Converts monochrome source data to grays before sending it to the screen. This function can be used to antialias a monochrome bitmap. In order for the antialiasing to occur, the height or width of the source data must be smaller than the corresponding dimensions of the destination view.

bitmap — The bitmap source data, as created by any of the global functions `MakeBitmap`, `GetPictAsBits` or `GetPictAsPixels`.

style — A style array containing a transform represented by a source rect and destination rect. This array conforms to the DrawShape interface described in Newton Programmer's Reference, although the transform's source rect is ignored because the bitmap's bounds are used for the source rect. The transform's destination rect is the desired size for the entire picture. If transform is `nil`, the view bounds is used.

If the bitmap is not monochrome data, the data is displayed as it would if `DrawBitmap` was called rather than `GrayShrink`.

Gray Text in clEditView Views

The `font` frame used by `clEditView` views has 3 slots: 'family, 'face, and 'size. You can add a `'color` slot to the font frame to cause an edit view to render text in gray tones by. The `'color` slot uses the constants described in "Color definitions in graphic shapes' style frame" (page 5-3).

Gray viewFormat

The `viewFormat` can take advantage of gray colors and patterns. This is done by selecting and defining a custom color for frame, fill, or lines. The gray color can be defined as in "Color definitions in graphic shapes' style frame" (page 5-3).

Utility Functions

PackRGB

`PackRGB(`*red*`,` *green*`,` *blue*`)`

Returns a packed integer in a format that can be used when drawing shapes or text.

red	A 16-bit integer, ranging from 0 to 65535
green	A 16-bit integer, ranging from 0 to 65535
blue	A 16-bit integer, ranging from 0 to 65535

GetRed

`GetRed (`*packedRGB*`)`

Returns the value of the *packedRGB* argument's red component as a 16-bit integer.

packedRGB	A packed integer representation of an RGB color

GetGreen

`GetGreen (`*packedRGB*`)`

Returns the value of the *packedRGB* argument's green component as a 16-bit integer.

packedRGB	A packed integer representation of an RGB color

GetBlue

`GetBlue (`*packedRGB*`)`

Returns the value of the *packedRGB* argument's blue component as a 16-bit integer.

packedRGB	A packed integer representation of an RGB color

GetTone

`GetTone(`*packedRGB*`)`

Returns an integer representing the 4-bit gray tone corresponding to the *packedRGB* value.

packedRGB A packed integer representation of an RGB color

IsEqualTone

`IsEqualTone(`*packedRGB1*`,` *packedRGB2*`)`

Returns `true` when the values of its arguments map to the same gray tone.

packedRGB1 A packed integer representation of an RGB color

packedRGB2 A packed integer representation of an RGB color

PtInCPicture

`PtInCPicture(`*x*`,` *y*`,` *picture*`)`

PtInPicture returns a Boolean `false` when the pixel specified by the x and y arguments is white or outside the bounds of *picture*. (Note: picture is a bitmap, not a PICT.)

This function is similar to the existing `PtInPicture` function. `PtInCpicture` returns the value of the specified pixel. This value is based on the bit depth of the bitmap.

For bitmaps with a bit depth of 1, 2, 4, and 8, the pixel is an index. For example, if the bitmap has a bit depth of 4, the value returned by the function would range from 0 to 15.

If the bitmap has a bit depth of 16 or 32, the pixels have a direct format, and the function will return the direct RGB pixel value.

When the pixel specified by the *x* and *y* coordinates lies outside the bounds of the *picture* bitmap, this function returns `-1`.

Compatibility Information

The goal of this implementation is to have complete backward compatibility with software written for version 2.0 of the Newton operating system. If an application contains a monochrome PICT or bitmap, it is displayed properly.

New applications that take advantage of the grayscale imaging capabilities will not be compatible with version 2.0 of the Newton operating system. If an application contains a color PICT, the data is thrown away by the 2.0 imaging subsystem. If an application contains a color bitmap, the 2.0 operating system displays the image incorrectly and the result is undefined. Application that contains references to the new textColor slot causes a frames exception on a 2.0-based system.

Note that if a 2.1-based application beams or mails a Notepad note containing a frame that holds a `'textColor` slot, the 2.0 application ignores the and 'textColor slot. A 2.1 application that tries to use the `'textColor` symbol on a 2.0 ROM causes a frames exception.

Newton OS 2.1 Graphics Shapes ERS

This document describes changes to the shape-based graphics model in the Newton 2.1 OS. This is the primary graphics model used by applications to draw custom items.

Data Structures

The style frame has changed in Newton 2.1 OS.

Modifications to the Style Frame

The following three changes have been made to style frames:

- Text shapes can be rendered with gray tones by using the new `textPattern` slot in the style frame. This slot can take the same values as the style frame slot `penPattern`. This slot makes it possible to use the same style frame for an array containing both non-text shapes and text rendered with different tones.

- In Newton 2.0 OS, text was drawn with a dithered pattern if the `penPattern` slot was set to `vfGray`. In Newton 2.1 OS, lines, text, and ink use `penPattern` to define their rendered tone. The geometric frames around shapes (rect, roundrects, ovals) are treated as lines. The interior of shapes are defined with `fillPattern`.
- Style frames can now include a `selection` slot with a numeric value. If present, "resize handles" (like you see in most drawing programs) will be drawn in the four corners of the shape bounds. The value is the size of the handles; an even number is recommended as it centers the handles best over the corners. The `FindShape` (page 6-3) function supports hit testing of these handles.

Functions

This section describes functions which are new to the Newton 2.1 OS, and older functions which have been changed in this OS release.

New Functions

The following functions are new to Newton 2.1 OS.

ConvertDropToShape

`ConvertDropToShape(`*dropType*`, `*dropFrame*`)`

Converts clipboard-format data into shape data.

dropType	On of the following symbols: `'polygon`, `'ink`, `'picture`, `'shape`, or `'text`.
dropFrame	Clipboard data that comes into the `ViewDropScript`.

It returns a shape or a shape array, depending on whether there's a style that needs specifying.

FindShape

`FindShape(`*shape*`,`*x*`,`*y*`,`*style*`)`

Indicates whether the point (x,y) lies in the specified shape. This function works like the `HitShape` function, however, the following behavior is different:

- Shapes are found from front to back instead of back to front. If two shapes overlap and you call `HitShape` with a point that was inside both shapes, the backmost shape was returned. `FindShape` returns the frontmost shape.
- The new ink shapes are supported; see "MakeInk" (page 6-4).
- If a shape has no fill pattern the click "falls through" and will miss, or hit a shape below it.
- There is "slop" built into the hit testing so clicks a few pixels from a shape will still hit the shape.
- If a shape is drawn with a selection style, you can "hit" these hilited corners; see both "Modifications to the Style Frame" (page 6-1), and the description of the `vertex` slot of the frame returned by this function.
- It now takes a style frame as an argument so you can use it the same way you do `DrawShape`.

shape	The shape to test.
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.
style	A style frame. The effect of this parameter is to have this function return `nil` or otherwise depending on whether the point (*x*,*y*) would have been in *shape*, if *shape* had been drawn with this style frame.

This function returns nil if there was no hit. Otherwise, it returns a frame with the following slots:

Slot Descriptions

vertex Either nil or one of the integers: 0, 1, 2, or 3. The value nil is returned if the hit was not in the corner of a shape; see "Modifications to the Style Frame" (page 6-1) for more information. Otherwise, if the hit was in a corner, the integers returned are to be interpreted as follows:

0	The northwest corner.
1	The northeast corner.
2	The southeast corner.
3	The southwest corner.

This slot will only be non-`nil` if you hit the corner of a shape and the selection style is a numeric value.

path A path expression to the shape that was hit.

GetPointsArrayXY

`GetPointsArrayXY(`*unit*`)`

Returns an array of points extracted from the specified unit. The original `GetPointsArray` function returned an array of points in y,x order. This routine does the same thing but returns them in x,y order.

unit A unit passed to the `ViewWordScript`, `ViewShapeScript`, `ViewStrokeScript`, and `ViewGestureScript` methods.

MakeInk

`MakeInk(`*inkdata*`,`*left*`, `*top*`, `*right*`, `*bottom*`)`

Creates an ink shape in the specified bounds box.

inkdata The ink data object.

left The left boundary.

top The top boundary.

right The right boundary.

bottom The bottom boundary.

This ink shape will work fine when passed to the various shape methods: `DrawShape`, `OffsetShape`, `IsPrimShape`, `ScaleShape`, etc. `HitShape` however does not work on these new objects. The new and improved `FindShape` (page 6-3) does however.

The bounds slot doesn't scale the ink, it just tells `DrawShape` where to place the upper left corner; it doesn't clip to the box.

MungeShape

`MungeShape(`*shape*`, `*action*`, `*style*`)`

Flips a shape, or rotates it to the right. It modifies the shape in-place, unless the shape is unmodifiable. In that case, it will create a bitmap that looks like the shape, and operate on the new bitmap.

shape	A shape or shape array.
action	On of the following symbols: `'rotateRight`, `'flipHorizontal`, or `'flipVertical`.
style	Style to use if shape must be converted to a bitmap.

This function returns a shape or shape array.

PictToShape

`PictToShape(`*pict*`)`

Takes a pict binary object and returns an array of shapes that will produce the same bits on the screen.

pict	The pict binary object to convert.

Changed Functions

The following changes have been made to functions which existed in earlier releases.

MakeShape now handles bitmaps with masks

`MakeShape` will now create bitmap shapes with an additional slot, `mask`, if the bitmap also has a mask. This mask will not be used unless it is drawn with the `transferMode` set to `modeMask`.

GetStrokePointsArray filters more points and swaps point coordinates

In previous release this function accepted an integer *format* parameter. It can now accept a frame of the following form as well as an integer:

Slot descriptions

`format`	An integer, this integer has the same interpretation as the *format* parameter of previous releases.
`distance`	Optional. An integer, the minimum distance between points.The default is 0. This significantly reduces the size of the array you get back from this function. This is handy as you don't have to munge through so much data if you want to analyze a stroke yourself from scripting. You can also create polygon views with reasonable fidelity using significantly fewer points.
`order`	Optional. One of the following symbols: `'xy` or `'yx`. The default is `'yx`. This slot controls the xy ordering of the points. Normally, `GetStrokePointsArray` returns the points in (y,x) order. If this slot has the value `'xy`, the points are returned in (x,y) order.

Newton OS 2.1 Sound ERS

This document describes the developer API for the Newton OS 2.1 Sound Manager, including the sound input interface, and improvements to playback and sound compression.

Terminology

channel	A virtual connection to specific piece of sound hardware.
sample	Sound consisting of sound (ADC) samples. The Runt-based products use the same sample format as Macintosh (8 bit unsigned). Voyager-based products use 16-bit linear (signed) samples.
volume	A value used to specify the loudness for a sound. The 1.x interfaces support integer volumes 0 through 4, (0 = quiet; 4 = maximum volume). The 2.0 interfaces support real volumes in the volume slot of a sound frame, which correspond to dB levels. A large negative value corresponds to silence, and 0.0 corresponds to full volume. The Newton OS 2.1 interfaces add a dB based system volume (Get/SetSystemVolume).

User Interface

Other than support for hardware volume control, there is no direct user interface for sounds, apart from the sound and alarms prefs panel, and the volume control in the Extras Drawer.

Developer Interface

This section reprises the 2.0 sound interface and its predecessors, then describes the Newton OS 2.1 enhancements and extensions to the 2.0 interface.

Sound Interface in 1.x Systems

The following functions are supported as documented in *Newton Programmer's Reference 2.0*:

```
PlaySound(soundFrame);
PlaySoundSync(soundFrame);
GetVolume();
SetVolume(volume);
```

Note

`PlaySound` honors user preferences settings for pen, alarm and action sound effects by comparing the sound frame passed as its argument to the "typical" sound effects; for example, `PlaySound(ROM_click)` is silent when "pen sound effects" are turned off. This behavior can produce unfortunate side-effects, the most notable being that `GetRoot():SysBeep()` does nothing when `beepSound` is the same as `alarmSound` and alarm sound effects are disabled. ◆

Sound Interface in 2.0 Systems

The functions described in this section can be used to overcome the limitations described in "Sound Interface in 1.x Systems" (page 7-2). These functions allow you to play sounds at specific volumes easily, or to play system sounds without regard to the current user preference settings that govern system sound effects.

PlaySoundAtVolume

```
PlaySoundAtVolume(soundFrame, volume)
```

Plays the soundFrame at the specified volume. If volume is nil, it uses the current sound volume. This function calls PlaySound, so it is subject to the same limitations as PlaySound (see Note: above).

PlaySoundIrregardless

```
PlaySoundIrregardless(soundFrame)
```

Plays the specified sound frame, ignoring the user sound prefs settings, thus always using the system volume.

PlaySoundIrregardlessAtVolume

```
PlaySoundIrregardlessAtVolume(soundFrame, volume);
```

As PlaySoundIrregardless at the specified volume. If volume is nil, it uses the current system volume. This function does not call PlaySound.

PlaySoundEffect

```
PlaySoundEffect(soundFrame, volume, type);
```

Plays the sound specified by soundFrame at the volume specified by volume if user prefs allow the sound type. Type can be one of 'pen, 'alarm, or 'action, checked against userConfig slots 'penSoundEffects, 'alarmSoundEffects, 'actionSoundEffects respectively. If volume is NIL, the system volume is used. This function is preferred to all other versions of PlaySound for playing sound effects.

If type is not 'pen, 'alarm, or 'action, then the sound is always played at the specified volume (same as PlaySoundIrregardlessAtVolume).

Clicker

```
Clicker()
```

Plays various "click" sounds. Use this for pen sounds instead of PlaySound(ROM_click).

Sound Interface for Newton OS 2.1

Support for hardware volume control has been added to the system. When a particular system has hardware volume control, the volume parameters for all sound calls are ignored.

GetSystemVolume

```
GetSystemVolume()
```

Returns the system volume in decibels.

SetSystemVolume

```
SetSystemVolume(volume)
```

Sets the system sound output level as specified and returns the output level that was actually set. Most hardware supports only a few distinct volume levels. This call sets system volume to the level nearest that supported by the current sound driver. If the hardware supports a hardware volume control, this function returns the hardware volume control's current setting. (Basically it turns into a `GetSystemVolume(..)` call.

volume Sound output level to set, expressed in decibels.

For more information, see the descriptions of new slots added for Voyager sound support, as described in "Sound Frame Format" (page 7-7).

For information on sound input, see "Sound Input" (page 7-12).

GetSoundFrame

```
GetSoundFrame(name)
```

GetSoundFrame is a new build-time function available for retrieving sounds. The calling convention is exactly the same as GetSound and GetSound11. GetSoundFrame returns a sound frame in whatever format is

specified in the source sound resource. The old routines hard-wired the sound frame to a specific sample size and sample rate. GetSoundFrame(...) is capable of loading sounds other than 22khz and 11khz sounds. It can theoretically load any type of sound source. The Newton, of course, will only play certain kinds of sounds. This call should be added to the platform file.

protoSoundChannel

The `protoSoundChannel` system prototype supplies the methods described in this section.

Open

soundChannel`:Open()`

Open the sound channel. This method allocates the TUSoundChannel for this channel. Throws `|evt.ex.fr|` if an error occurs. Returns nil.

soundChannel A frame having a `_proto` slot that references the `protoSoundChannel` system prototype.

Close

soundChannel`:Close()`

Closes the sound channel. This method deletes the TUSoundChannel allocated for this channel. Throws `|evt.ex.fr|` if an error occurs. Returns nil.

soundChannel A valid sound channel object created by the `Open` method of the `protoSoundChannel` system prototype.

◆ **WARNING**
The memory allocated for a sound channel is not released when the *soundChannel* frame is GC'd. You MUST call the `Close` method explicitly to avoid memory leaks. ◆

Schedule

soundChannel:`Schedule(`*soundFrame*`)`

Queues the *soundFrame* for play. Throws |evt.ex.fr| if an error occurs. Returns nil. If the soundFrame defines a callback function, the sound channel sends the callback message to the *soundFrame* when play completes.

Start

soundChannel:`Start(`*async*`)`

Starts the sound channel. The channel will begin playing sound frames in the order they were scheduled (see below). Throws |evt.ex.fr| if an error occurs. Returns nil.

async Pass non-nil to play sounds asynchronously. If this value is nil, control does not return until the entire play queue is empty (all scheduled sounds complete).

Stop

soundChannel:`Stop()`

Stops playing the current sound, if any, and sends a callback message (state = 1 (kSoundAborted)), if defined, to each sound frame in the specified channel's queue. When this method returns, all scheduled sounds will have received a callback message, and the queue will be empty. Throws |evt.ex.fr| if an error occurs. Returns a sound result frame (page 7-9) indicating which sound frame was stopped, or nil if no sound was currently playing.

Pause

soundChannel:`Pause()`

Temporarily suspends play in progress or resumes play already paused. Throws |evt.ex.fr| if an error occurs. Returns a sound result frame (page 7-9) indicating which sound frame was stopped, or nil if no sound was playing when the `Pause` message was sent to *soundChannel*. Scheduled sounds are unaffected.

SetVolume

soundChannel:`SetVolume(`*volume*`)`

Sets playback level for the specified channel.

volume	An integer or NIL. Values 0, 1, 2, 3, or 4 correspond to decibel levels -∞ dB, -18 dB, -6 dB, -3 dB, or 0 dB (unity gain) respectively. If volume is NIL, then the "master" volume is set (the driver's preferred volume).

GetVolume

soundChannel:`GetVolume()`

Returns the specified channel's current playback volume, expressed in decibels. The integer return values 0, 1, 2, 3, and 4 correspond to decibel levels -∞ dB, -18 dB, -6 dB, -3 dB, and 0 dB (unity gain) respectively. This mehod returns nil when the channel does not specify its own volume but instead inherits it from user preference settings (the default behavior).

IsPaused

soundChannel:`IsPaused()`

Returns TRUE if the channel is paused, nil otherwise.

IsActive

soundChannel:`IsActive()`

Returns TRUE if the channel is active (playing, recording or paused), nil otherwise.

Sound Frame Format

A sound frame may have the following slots:

Slot descriptions

`sndFrameType`	Symbol specifying the type of sound frame. The `'simpleSound` symbol indicates a standard sound, while the `'codec` symbol indicates a compressed sound.
`samples`	Required. Binary of class 'samples. Contains the sound samples.
`samplingRate`	Optional. Real or Integer value describing the sampling rate of data in the `samples` slot. (8000.0, 11013.21586, and 22026.43172 are common values). If missing, the sound channel assumes 22026.43172. For more information, see the description of the `recordingQuality` slot in the system's user configuration data.
`compressionType`	Optional. Integer. Encoding format of samples. If present, it must be `kSampleStandard` (0), `kSampleLinear` (6), or `kSampleMuLaw` (1). If missing, `kSampleStandard` is assumed.
`dataType`	Optional. Integer value specifying the size of samples in bits. If present, it must be 8 (k8Bit) or 16 (k16Bit). If missing, k8Bit is assumed.

Note

Older versions of NTK generate sound frames having the value 0 (zero) in the dataType slot. The Sound Manager assumes 0 (zero) is the same as 8 (k8Bit). ◆

`volume`	Optional. Integer or real value specifying the volume level at which to play this sound. If missing, the channel's volume setting is assumed. Note that if volume is an integer it must have the value 0, 1, 2, 3, or 4 corresponding to decibel levels -∞ dB, -18 dB, -6 dB, -3 dB, or 0 dB (unity gain) respectively. If volume is real, it is treated as the actual dB level, and should be negative. This value overrides the system volume and the channel volume—including values set by functions such as `PlaySoundAtVolume` and sound channel methods such

as `SetVolume`. If the system has a hardware volume control, the hardware control overrides everything else.

`start` Optional. Integer value that is the index of the first sample toplay. When this value is missing, 0 is assumed. Omit this slot in sound frames used for recording.

`count` Integer that is the number of samples to play. When this value is missing, `Length(samples) / (dataType/8)` is assumed. Omit this slot in sound frames used for recording.

`loops` Optional. Integer that is the number of times to repeat the sound. (i.e., setting `loops` to `3` causes the sound to play a total of four times). When this value is missing, 0 is assumed. There is no way to specify continuous play. Omit this slot in sound frames used for recording.

`callback` Optional function object executed when the sound frame completes. This function must be of the form

`func (`*state*`, `*result*`) begin ... end;`

state The state of the sound channel when the callback was executed. Values are:

`0 = kSoundCompleted`

`1 = kSoundAborted`

`2 = kSoundPaused`

result An integer error code, if present. For a listing of possible values, see "Sound Manager Error Codes" (page 7-11).

Sound Result Frame Format

A sound result frame holds the following slots:

Slot descriptions

`sound` The sound frame that was paused, stopped, or completed.

`index` Integer value that is a zero-based index into the `sample` slot. This value describes where in the sample data the sound channel was paused or stopped. The following inequality is always true:

```
soundFrame.start > soundFrame.index < (soundFrame.start + soundFrame.count).
```

Sound Manager Error Codes

This section lists error codes returned by the Sound Manager and describes possible causes for each.

`kError_Snd_Generic`

Error scheduling node

Codec channel aborted

No driver found

Channel open; no sound port

Attempt to do synchronous input

`kError_Snd_Bad_Msg`

Internal sound manager error

`kError_Snd_Not_Played`

All Sound stopped (power off or other problem)

`kError_Snd_Busy`

Internal resource conflict

`kError_Snd_Record_Busy`

Internal resource conflict

`kError_No_Samples_Given`

Start called with nothing scheduled

Cancel called for nonexistent node

Codec called with zero-length samples

`kError_Unsupported_Snd_Configuration`

Unable to create codec

Bad value for `sndFrameType` slot

Bad value for `compressionType` slot

Bad value for `dataType` slot

Invalid `samplingRate` slot

Input not implemented on target hardware

Driver doesn't support output

	Driver doesn't support input
`kError_Snd_Channel_Closed`	
	Channel aborted (`kChannelAborted` returned to frame callback)
	Channel stopped (returned to frame callback)
	Attempt to make call to closed channel
	Unable to start channel
	Unable to pause channel
	Unable to schedule block
	Unable to cancel channel
	Unable to stop channel
`kError_Snd_Cancelled`	
	Channel cancelled (frame callback)
	Channel aborted (frame callback)

Compatibility Information

No problems or issues expected. Eveything that worked on 1.x and 2.0 should work on Newton OS 2.1. Newton OS 2.1 sound frames are backwards compatible, although sounds created using the Voyager data format (16-bit linear) will not play on Runt-based units.

Sound Input

There are four main components to sound input: the NewtonScript API for recording and storing sound; the Sound Manager and driver support that implements the API; SoundRecorder, the UI element that lets you control recording and playback; and SoundPaper, the UI element that lets you store and organize pieces of recorded sound.

NewtonScript Interface

The interfaces for recording are nearly identical to the sound channel interfaces for playback. The protoSoundChannel frame and protoSoundFrame have both been extended to support recording.

Because recording requires some UI to start and stop the recording process, there is no simple interface for recording equivalent to PlaySound(). You must create a new sound channel to do recording, and call open, schedule, start and stop.

Generally, here is how you do a recording using a sound channel. First create a new sound channel, using newRecording. Next, open the sound channel by sending it the open message. Create a new block of memory for it to record into by sending the newInputBlock message, which returns a soundFrame, and then schedule it using the schedule message. Once a soundFrame is scheduled, you can start the recording by sending the start message to the sound channel.

After you have sent the start message, there will be about a 1/2 second delay while the sound hardware powers up. Eventually we will probably play a record beep during that interval.

The sound manager then records sound into the scheduled soundFrame. Once the provided buffer is full (default size 64K), it looks to see if there is another scheduled soundFrame to continue recording into. It calls the callback provided by the first soundFrame, then it continues recording into the next soundFrame. It is the application's job to ensure that enough soundFrames are scheduled to keep up with it. In the soundRecorder, after starting the sound channel, I schedule the second soundFrame immediately, and then schedule one additional new one when an old one is filled up.

It is the application's responsibility to keep track of which sound frames have been created and scheduled, and to update the size of the final sound block when the stop button is pressed (see the return value of the stop function).

Devices and Channels

We now support multiple output devices for playback, and multiple input devices for recording. The system uses a default output device which can be overridden by the channel. The system also uses a default input device which can similarly be overridden by the channel.

Despite this flexibility, the hardware cannot send two simultaneous sounds to different devices: all concurrent or overlapping output must go to the same device(s). This means that if you have a sound playing on the external speaker, and you request a sound on the internal speaker, it will be routed to the external speaker instead. The same is true of input: all simultaneously active input channels will have the same source.

The number of simultaneous sounds that can be played is limited by the processing power of the CPU. On a 710, the number is about 4, depending on the exact nature of the sound being played.

The current release supports the following output devices:

```
kDefaultDevice      0x00   // the internal speaker
kInternalSpeaker    0x01   // the internal speaker
kLineOut            0x08   // line out on the interconnect
```

and the following input devices:

```
kDefaultDevice   0x00// the internal microphone
kInternalMic     0x04// the internal microphone
kLineIn          0x10// line in on the interconnect
```

Notice that each device is represented by a single bit, so that the devices can theoretically be OR'ed together. In the current release, we do not support more than one input or output device at a time.

It is possible to have multiple recording sessions running simultaneously.

Input Gain

While recording, you can specify an input gain, which is an amplification applied to the incoming signal.

The input gain can have a value ranging from 0 to 255. If the input gain is 0, then the incoming signal will not be amplified at all; if the input gain is 255, then the signal will be amplified by an amount that the driver has determined to be a maximum desirable amount. The middle value of 128 is considered to be an "optimal" setting for normal use.

The behavior of the input gain can seem non-intuitive. The signal that comes from the internal microphone is a very weak one, and the system relies on the input gain to boost it to a level that you can hear. Thus, when you are using the internal microphone and you set the input gain to 0, the recording will be silent. The signal that comes from the line-in jack, however, is much stronger. When the input gain is set to 0, the recording is quite loud, which you might not expect if you think about it as an input volume instead of an input gain. In both cases, the default value of 128 instructs the device to amplify the signal to an "optimal" level.

We do not support a self-adjusting input gain.

ProtoSoundChannel

ProtoSoundChannel has been modified to support input. This section describes the new slots and methods that have been added to protoSoundChannel.

The following new slots have been added:

Slot descriptions

`direction` Indicates whether the soundChannel is to be used for input or output. It can have two valid values: 'record or 'play. This is set to 'record by the :newRecording() function. If this slot is invalid or nil, 'play is assumed.

`outputDevice` Specifies the device that is to be used for playback. If nil, then playback is done on the device specified by the userConfiguration slot of the same name. If that is nil, then the internal speaker is used. The default

	outputDevice in protoSoundChannel is nil. Device definitions are give above in the section on Devices and Channels.
`inputDevice`	Specifies the device that is to be used for recording. If nil, then recording is done on the device specified by the userConfiguration slot of the same name. If that is nil, then the internal microphone is used. The default value of inputDevice in protoSoundChannel is nil. Device definitions are give above in the section on Devices and Channels.
`inputGain`	A value that specifies how much the input signal is to be amplified before it is recorded. It can have a value between 0 and 255, as described in the section on Input Gain, above. If nil, then the inputGain specified by the userConfiguration frame is used. If that is nil, then a suitable default value is used. This value is used to determine an initial value for input gain. Although the SetInputGain functions (global and local) can change the input gain for the channel, they will not change the value of the inputGain slot.
inputBlockSize	The allocation size used by `newInputBlock` when it creates the VBO to record into. The default value is 65536. You can change the size, but the timing of the sound manager may not work for sizes that are too small.

The following new methods have been added to `protoSoundChannel`:

IsOpen

soundChannel`:IsOpen()`

IsOpen tests to see whether the sound channel is open, and returns true if it is.

NewRecording

soundChannel:`NewRecording()`

Creates a new protoSoundChannel frame that is properly initialized for sound input. It is simply a protoSoundChannel that has its direction slot set to 'record.

NewInputBlock

soundChannel:`newInputBlock(`*callback*`)`

Creates a new soundFrame, that has all its slots set up for recording speech at an optimal combination of fidelity and compression. It allocates a VBO, and stores it in the 'samples slot. This is where the recorded data is stored. The size of the VBO is determined by the value of the inputBlockSize slot in the channel, which defaults to 65536.

A newly created input block looks like this:

```
{
_proto: protoSoundFrame,
samples: vbo,           // size determined by inputBlockSize
callback: callback,     // specified by caller
dataType: k8Bit,        // eight bits per sample
compressionType: kSampleMuLaw,// MuLaw encoded
samplingRate: 10800,    // 10K samples per second
}
```

This is a reasonable default setting for voice recording. However, you should not rely on these specific values. If you require specific settings, you should override them yourself.

SetInputGain

soundChannel:`setInputGain(`*gain*`)`

Sets the amplification applied to the signal coming from the input device. SetInputGain can only be called on open channels, or an error is thrown.

This method changes the input gain, in real time, of the current input channel (and because the hardware supports only one input source at a time, all other active input channels).

This method does not change the value of the inputGain slot in the channel. This value is assumed to be the default value used when the channel is started.

gain	The amplification to be applied to the input signal. For information on values, see “Input Gain” (page 7-15).

GetInputGain

soundChannel:`getInputGain()`

Returns the gain currently used by the specified channel. The channel must be open. For information on the meaning of values this method returns, see “Input Gain” (page 7-15).

ProtoSoundFrame

ProtoSoundFrame is newly created for Newton OS 2.1. Not only does it contain all the slots described as valid for 2.0 soundFrames, but it now has some functions that make it easier to work with. Note that sound frames that are not derived from ProtoSoundFrame will not support these functions.

GetSampleCount

soundFrame:`GetSampleCount()`

GetSampleCount returns the number of samples in the soundFrame. This is based on the value of the length slot, if non-nil, or on the size of the samples object otherwise.

GetSampleSize

soundFrame:`GetSampleSize()`

Returns the size of each sample, in bytes.

GetSamplingRate

soundFrame:`GetSamplingRate()`

Returns the rate at which the data in the soundFrame was (or will be) recorded.

GetPlayingTime

soundFrame:`GetPlayingTime()`

Returns the playing time of the sound, in seconds, as a floating point number. This is useful for creating progress indicators for the sound.

SetRecordingLength

soundFrame:`SetRecordingLength(`*numSamples*`,` *callback*`)`

SetRecordingLength should be called after the user presses the stop button. It sets the length slot to the indicated value, then it adds a deferred call to actually set the length of the VBO. Since the functions above rely on the length slot, and not the size, both GetPlayingTime and GetSampleCount will both return correct values immediately after SetRecordingLength has been called.

numSamples	An integer expressing the number of samples to include in the sound frame.
callback	Callback function object executed when the `SetLength` is actually completed. This may be necessary for an interface in which the user can switch rapidly from recording to playback. This slot may hold the value `nil` or a callback function of the form `func () begin ... end;`

MuLaw Compression

In addition to kSampleStandard (0) and kSampleLinear (6), a third compressionType has been added: kSampleMuLaw (1). This compressionType is used to compress voice data from 16 bits (returned by the sound hardware) to 8 bits. When you specify kSampleMuLaw, you must set dataType to k8Bit.

Setting Global Sound Preferences

When optional override values are not specified in the sound channel, default values are read from slots in the system's user configuration data. This data is soup-based, and therefore persists across application launches and soft resets.

This section describes how to use the GetUserConfig and SetUserConfig functions to access sound-related user configuration slots.

Getting and Setting Input Gain

To retrieve the global input gain value, pass the `'inputGain` symbol to the `GetUserConfig` function, as in the following example

```
GetUserConfig('inputGain)
```

The value returned is an integer between 0 and 255 that sound channels used for input take as the default value of their inputGain slot. The default value is 128.

To set the global input gain value, pass the `'inputGain` symbol and the new gain value to the `SetUserConfig` function, as in the following example

```
SetUserConfig('inputGain, gain)
```

This code sets the value used as the default value of `inputGain` slot in sound channels used for input. If there are any active input channels, their input gain will immediately be set to the specified value. The gain parameter must hold an integer value between 0 and 255.

This call changes the value of the `inputGain` slot in the system's user configuration data, but not the value of the `inputGain` slots in any active channels.

Getting or Setting Default Input or Output Devices

`GetUserConfig('inputDevice)`

This code returns a constant representing the current default input device. This default can be overridden by putting an inputDevice slot in your protoSoundChannel. Device definitions are give above in the section on Devices and Channels.

`SetUserConfig('inputDevice,` *value*`)`

This code sets a global that indicates the default input device used during recording. It does not change the device for a currently recording input channel.

`GetUserConfig('outputDevice)`

This code returns a constant representing the current default output device. This default can be overridden by putting an outputDevice slot in your protoSoundChannel. Device definitions are give above in the section on Devices and Channels.

`SetUserConfig('outputDevice,` *value*`)`

This code sets a global that indicates the default output device used during playback. It does not change the device for a currently playing output channel.

UserConfiguration Slots

`soundVolumeDb`	The current system sound volume, in decibels.
`alarmVolumeDb`	The current system alarm volume, in decibels.

Rathe than accessing these slots directly, use the `SetUserConfig` function to set their values. This function calls other functions necessary to make the system use the new value.

Sound Recorder and Player

The user controls recording using an on-screen recorder slip described in this section.

The `soundRecorder` is a root child, so you can access it by calling

```
sr := getroot().soundRecorder;
```

If you want to open it to do a recording, call:

```
sr:OpenRecord(callback);
```

This sets up the recorder slip to do recording, and opens it on the screen. When the user presses the close box to close the recorder, the callback is called with a single argument, the array of soundframes that were allocated as a result of recording.

To play a sound via the sound recorder slip, call:

```
sr:OpenPlay(soundFrame);
```

This sets up the recorder to play the specified *soundFrame*, and opens it on the screen.

After opening the recorder with either `openPlay` or `openRecord`, the following slots in the base view can be changed to override the values in an associated `soundChannel` or `soundFrame`.

`fSoundFrameSlots` may hold a frame that contains slots to be copied to the sound frames for recording. For example:

```
fSoundFrameSlots:
            {sndFrameType: 'simpleSound,
             samplingRate: 10800,
             compressionType: kSampleMuLaw,
             dataType: k8Bit
            },
```

defines a frame that overrides the default values with themselves. This frame can contain any slots that you want copied to the sound frame.

`fInputGain` Sets the `inputGain` slot in the `soundChannel`

`fInputDevice` Sets the `inputDevice` slot in the `soundChannel`

`fOutputDevice` Sets the `outputDevice` slot in the `soundChannel`

ProtoRecorderEngine

••• to be supplied •••

This section is to contain a description of the protoRecorderEngine. This is a base class that can be used to create custom recorders.

ProtoRecorderView

••• to be supplied •••

This section is to contain a description of the protoRecorderView. This proto can be used to put an embedded recorder view within another view, as used by SoundPaper.

SoundPaper

A new type of stationery called SoundPaper provides a standard way for the user to store and manipulate sounds. The user will create a new sheet of SoundPaper by tapping the "New" button, and picking "Recording". This sheet of paper displays a small microphone and some informational text, and it also contains recorder controls. The user uses the recorder to record a sound. The record button should be used to start a recording; the play button is used to start playback; and the stop button is used to stop either recording or playback. Once you start recording or playing, the record/play button changes to a pause button. You can pause and resume recording/playback using this button.

.

Dial-In Networks ERS

The dial-in network application program interface (API) allows you to add dial-in networks to augment the built-in SprintNet and ConcertNet networks already in the system. A dial-in network basically provides phone numbers for an application (or transport) to call to get access to the network.

For example, a CompuServe mail client would need to register a CompuServe dial-in network to supply numbers for connecting to the CompuServe network.

The primary function of a dial-in network is to supply phone numbers to call given a particular location. It supplies these phone numbers by providing a function to be called by elements such as the connection slip and the Internet Enabler. This function returns the possible numbers.

Dial-in networks are stored in a registry in the system. To register a dial-in network with the system, a developer must put a dial-in network into this registry. A developer does this by calling the registration function `RegDialinNetwork`, passing in a network frame that describes the dial-in network; see "Network Frame" (page 8-2).

Data Structures

Two data structures are used with this API: access frames (page 8-2) and network frames (page 8-2).

Access Frame

An access frame contains the following slots:

Slot descriptions

`mailNewtwork`	A symbol for the network.
`mailPhone`	A string for the phone number.
`baud`	An integer indicating the baud rate.

Network Frame

A network frame contains the following slots:

Slot descriptions

`title`	A string describing the network, such as `"SprintNet"` or `"ConcertNet"`.
`id`	A symbol uniquely identifying the network,
`GetAccessNumbers`	A function described in "GetAccessNumbers" (page 8-2).

GetAccessNumbers

networkFrame`:GetAccessNumbers(`*worksiteFrame*`,` *cityFrame*`)`

Called to retrieve an array of access numbers for a given worksite or city.

worksiteFrame	A frame of the format of a Names worksite soup entry; see "Worksite Entries" (page 16-24) in Chapter 16,

	"Built-in Applications and System Data Reference," in *Newton Programmer's Reference* .
cityFrame	A frame with the same format as the frames returned by the `GetCityEntry` function; see "GetCityEntry" (page 16-82) in Chapter 16, "Built-in Applications and System Data Reference," in *Newton Programmer's Reference* .

This function should return either an array of access frames, or nil if no numbers are available; access frames are described in "Access Frame" (page 8-2). You should never, however, return the empty array (`[]`).

It is up to you to implement a mechanism to store and retrieve these access numbers. One possible implementation is to store a frame containing this data in your package. If this data needs to be dynamic, to add new access numbers for example, you will probably want to create a soup for this data.

Global Functions

The following functions are provided.

RegDialinNetwork

`RegDialinNetwork(`*networkSym*`,` *networkFrame*`) //platform file fn.`

Registers a new dial-in network with the 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 kRegDialinNetworkFunc with (`*networkSym*`,` *networkFrame*`);`

◆

networkSym	A symbol uniquely identifying the network
networkFrame	A network frame, as described in "Network Frame" (page 8-2).

This function should usually be called from your part's `InstallScript`, as in the following code sample:

```
DefConst ( 'dudeNetFrame,
             {
                title: "DudeNet",
                id: 'dudeNet,
                GetAccessNumbers: func(worksite,city)
                       begin
                          local result := [];
                          if worksite then
                             AddArraySlot (
                                    result,
                                    {
                                    mailPhone:"111-1111",
                                    mailNetwork: 'dudeNet,
                                    baud: 9600
                                    }
                             )
                          if city then
                             AddArraySlot (
                                result,
                                {
                                   mailPhone:"222-2222",
                                   mailNetwork: 'dudeNet,
                                   baud: 2400
                                }
                             )
                          result;
                       end
             }
          );
partData := {};
InstallScript := func(partFrame,removeFrame) //auto part
   begin
      call kRegDialinNetworkFunc with ('dudeNet,dudeNetFrame);
   end;
```

UnRegDialinNetwork

```
UnRegDialinNetwork(networkSym) //platform file function
```

Unregisters a dial-in network from the system which had been registered with a call to `RegDialinNetwork`.

IMPORTANT

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

```
call kUnRegDialinNetworkFunc with (networkSym);
```

◆

networkSym The symbol used in the call to `RegDialinNetwork`.

This function should usually be called from your part's `RemoveScript`.

GetLocalAccessNums

```
GetLocalAccessNums(entry, which) //platform file function
```

Retrieves an array of access frames given a location frame and an array of dial-in network symbols to look for; see "Access Frame" (page 8-2).

IMPORTANT

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

```
call kGetLocalAccessNumsFunc with (entry, which);
```

◆

entry A location frame. Can be a worksite or a city location. If nil, `GetLocalAccessNums` uses the current emporium and city location.

For information on these various entities see the following sections of Chapter 16, "Built-in Applications

and System Data Reference," in *Newton Programmer's Reference* :

worksites	"Worksite Entries" (page 16-24), worksite entries are a type of Names soup entry.
cities	"GetCityEntry" (page 16-82), the `GetCityEntry` function returns a city location frame.
the current emporium	"User Configuration Variables" (page 16-105), the `currentEmporium` variable contains an alias to a Names worksite soup entry.

which An array of network symbols. Usually the transport's `networkSymbols` array if the Mail Enabler is used. Matches to all these symbols are returned.

This function returns an array of access frames; see "Access Frame" (page 8-2).

Note

If the mail transport does not contain the *networkSym* for the dial-in network within its `networkSymbols` slot, the network phone numbers will not appear in the connection slip. ◆

GetAllDialinNetworks

```
GetAllDialinNetworks() //platform file function
```

Returns an array of all the dial-in network frames that are registered in the system; see "Network Frame" (page 8-2).

IMPORTANT

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

```
call kGetAllDialinNetworksFunc with ();
```

◆

GetDialinNetwork

```
GetDialinNetwork(networkSym) //platform file function
```

Returns the dial-in network frame that corresponds to *networkSym;* see "Network Frame" (page 8-2).

IMPORTANT

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

```
call kGetDialinNetworkFunc with (networkSym);
```

◆

networkSym The symbol of the network whose frame to return.

CHAPTER 9

IrDA Communication Tool ERS

This ERS describes the new IrDA communications tool built into the Newton 2.1 OS.

Overview

The IrDA Tool is a comm tool implementation of the Infrared Data Association's standard for infrared communication. The IrDA standard consists of a hardware serial infrared interface specification (SIR), a link access protocol specification (IrLAP), a link management multiplexer protocol specification (IrLMP) and a transport protocol specification (IrTinyTP). The IrDA tool implements the IrLMP and IrLAP protocols and communicates with a serial driver that implements the SIR protocol. The IrDA transport protocol is not a component of the IrDA tool as it was not necessary to satisfy marketing requirements (HP IrDA printers do not require it). Future versions of the IrDA tool may include IrTinyTP, IrComm and other components.

The IrDA tool is a minimal implementation of IrLAP and IrLMP supporting both primary (for printing) and secondary (for connection) roles. Initial uses anticipated for the IrDA tool include printing, connection and beaming.

Printer drivers interface to the IrDA tool in the same way as the serial tool with the additional requirement that the destination printer must be specified with the `TCMOIrDAConnectionInfo` option, specifying "IrLPT" as the peer class name.

Connection (NBU/NCU and possibly NTK) also interfaces to the IrDA tool in the same way as the serial tool with the additional requirement of either identifying the Newton device (for the listen case) or identifying the host Mac/PC (for the connect case) with the `TCMOIrDAConnectionInfo` option.

Beaming is more complicated than printing or connection because the architecture needs to support communication with another IrDA equipped Newton device (MessagePad 2000 or eMate 300), with a SharpIR equipped Newton (MessagePad 100/110/120/130) or with a Sharp Wizard. There needs to be a way to distinguish the capabilities (beaming-wise) of the Newton to communicate with and start the appropriate SharpIR or IrDA tool.

Beaming auto-detection is done using the ProbeIR tool. For connecting, both protocols (SharpIR/IrDA) are tried, alternating between each. For listening, the infrared serial tool is opened in a mode that can detect either the Newton / Sharp Wizard protocol or the IrDA protocol. If either protocol is received then the appropriate tool is started.

Terminology

A lot of definitions and IrDA level details can be found in the IrLMP and IrLAP specification documents.

Here is a quick list/description of the IrDA protocol levels defined to date.

If you need more details, get the specs from the IrDA FTP/Web sites.

SIR	(Serial IR), hardware protocol for 9600 - 115.2k data transmission.
FIR	(Fast serial IR), hw protocol for 115.2k - 4m data transmission.
IrLAP	(Link Access Protocol). Built on top of SIR and/or FIR.
IrLMP	(Link Mgmt Protocol) - Multiplexor, Name Server, Endpoints. Built on IrLMP.
IrTP	(Transport) - Obsoleted by IrDA committee 10/95.
TinyTP	(Transport) - Simplified version of IrTP. Built on IrLMP.
IrComm	3-wire/9-wire serial/parallel-like interface. Built on TinyTP.
IrTA	Terminal Adapter. Built on IrComm.
IrOBEX	Object exchange protocol/API. Built on either IrComm or IrLMP.

The only pieces that will be supported in the first implementation of the IrDA tool are: SIR, IrLAP and IrLMP. This is the minimum required set to be IrDA compliant.

User Interface

Here are the parts of the Newton system that will make use of IrDA:

- Beaming. A user may beam to a Sharp-IR or IrDA-IR Newton. No user interface changes are required.
- Connection. Although the receiving end (a Macintosh or Windows-based PC) also needs to support communicating over IrDA.
- Printing. An IrDA capable print driver can be installed and chosen from "Choose other printer" item of the "printer" pop-up. The print driver interfaces to IrDA in a similar way that it interfaces to serial, with the exception that it must specify some options (see Developer API). A built-in IrDA printer driver for HP printers (using HPL protocol) has been added to Newton 2.1 OS.

Using the IrDA Tool

Clients of the IrDA tool access it using the Endpoint Interface. The IrDA tool service identifier is "irda". Most clients use CMGetEndpoint to open an Endpoint with the IrDA tool (printer drivers, beamer). NewtonScript applications open an Endpoint using the following option.

```
fEndpointConfigOptions :=
[
    {
        label:"irda",
        type:  'service,
        opCode:opSetRequired,
        result:nil,
    },
];
```

Establishing a connection with an IrDA device is a multi-staged process. The following describes each stage and the option(s) associated with that stage.

It should be noted that all IrDA options are evaluated at connect/listen time *only*. For convenience to developers, the options may be specified earlier at open or bind time, at connect/listen time, or even after a connect/listen has been established, but they will only be processed (evaluated) at connect/listen time. Once the connection has been established, you must disconnect and reconnect to change the connect/listen options.

Here is an example NewtonScript option array that specifies the options used with either a connect or listen request. Note that some fields of some options apply only to connect while other fields apply only to listen. They will be noted below.

Typically the only option that needs to be supplied is the ConnectionInfo option ("irci") to either identify your Newton device or to identify the device that you are connecting to. And even this option may be omitted if two Newton devices are communicating peer to peer, since they both use the default connection names.

```
fEndpointConnectOptions :=
```

```
[
            {
        label:"irdi",// IrDA discovery information
        type:  'option,
        opCode:opSetRequired,
        result:nil,
        form:  'template,
        data:
        {
            arglist:
            [
                8,
                kSvcHintPDAPalmtop,
                kSvcInfoHintPrinter,
                0,
                1,
            ],
            typelist:
            [
                kStruct,
                kULong,// num probe slots, default: 8
                kULong,// my service hint, default: PDA
                kULong,// service hint mask
                kULong,// returned devAddr of peer device
                kULong,// use standard media busy check?
            ],
        },
    },
    {
        label:"irci",// IrDA connect info
        type:  'option,
        opCode:opSetRequired,
        result:nil,
        form:  'template,
        data:
        {
            arglist:
            [
                0,
                0,
                4,     // i.e. strlen("Test")
                5,     // i.e. strlen("IrLPT")
```

```
            "Test",
            "IrLPT",
        ],
        typelist:
        [
            kStruct,
            kULong,// my lsap id, default: 0
            kULong,// peer lsap id, default: 0
            kULong,// my name length, def: 1
            kULong,// peer name length, def: 1
            [kArray, kChar, 0],// my name, def: "X"
            [kArray, kChar, 0],// peer name, def: "X"
        ],
    },
},
{
    label:"sbps",// serial bit rate
    type:  'option,
    opCode:opSetRequired,
    result:nil,
    form:  'template,
    data:
    {
        arglist:
        [
            115200,
        ],
        typelist:
        [
            kStruct,
            kULong,// max negotiate speed, def: 115.2k
        ],
    },
},
{
    label:"irrb",// IrDA recv buffers info
    type:  'option,
    opCode:opSetRequired,
    result:nil,
    form:  'template,
    data:
    {
```

```
            arglist:
            [
               2048,
               1,
            ],
            typelist:
            [
               kStruct,
               kULong,// size of ea recv buf, def: 512
               kULong,// num recv bufs used, def: 1
            ],
         },
      },
      {
         label:"irld",// IrDA link disconnect threshold
         type:  'option,
         opCode:opSetRequired,
         result:nil,
         form:  'template,
         data:
         {
            arglist:
            [
               8,
            ],
            typelist:
            [
               kStruct,
               kULong,// Time before disc, def: 40 secs
            ],
         },
      },
];
```

If the above example option is used to connect, the Newton limits discovered devices to printers, and connects to the IrDA device with the class name "IrLPT". The Newton will communicate (up to) 115.2k bps, and will receive

data using one 2k buffer. If there is no activity from the peer device, then the Newton will disconnect after 8 seconds.
Likewise, if the above example option is used to listen, the Newton advertises itself as a "PDA" with the class name "Test". The Newton will communicate (up to) 115.2k bps, and will receive data using one 2k buffer. If there is no activity from the peer device, then the Newton will disconnect after 8 seconds.

IrDA Tool Options

This section describes the IrDA communication tool options in detail.

Discovery Option

The first stage of the connection process is the Discovery Phase. This stage probes and accumulates a list of other IrDA devices within beaming range of the Newton. Each device is loosely identified by a service hints field (PDA/Computer/Printer/etc) and a device nickname. The `TCMOIrDADiscovery` option is used to specify which device to use and also how the Newton should appear to other devices that probe the Newton.

```
class TCMOIrDADiscovery : public TOption
{
    public:
              TCMOIrDADiscovery();

    ULong  fProbeSlots;
    ULong  fMyServiceHints;
    ULong  fPeerServiceHints;
    ULong  fPeerDevAddr;
    ULong  fMediaBusyCheck;
};
```

The field `fProbeSlots` is used to specify the number of "slots" used during probing. The valid choices are 1, 6, 8 and 12. The default is 8.

The fields `fMyServiceHints` and `fPeerServiceHints` are used to specify the category of device that you wish to discover (or specify the category of device you wish to be discovered as). Here are the possible service hints known to date (see latest IrDA specs for newer definitions).

```
#define kSvcHintPnPCompatible               0x00000001
#define kSvcHintPDAPalmtop                  0x00000002
#define kSvcHintComputer                    0x00000004
#define kSvcHintPrinter                     0x00000008
#define kSvcHintModem                       0x00000010
#define kSvcHintFAX                         0x00000020
#define kSvcHintLanAccess                   0x00000040
#define kSvcHintTelephony                   0x00000100
#define kSvcHintFileServer                  0x00000200
```

Note that the service hints can be bit-OR'd together to create the field. The default for `fMyServiceHints` is `kSvcHintPDAPalmtop` and the default for `fPeerServiceHints` is `0xFFFFFFFF` (accept any device during discovery phase).

The client can specify whatever it wishes for `fMyServiceHints`, but the value `kSvcHintPDAPalmtop` will be OR'd in by the IrDA comm tool.

The field `fMediaBusyCheck` is used to disable the 600ms delay before discovery begins. The default is `true` and is highly recommended to be left that way. If you are using a sender/receiver model (like Newton beaming) then you may want to set this field to `nil`. But beware, for this invalidates IrDA complicancy with other IrDA devices.

The field `fPeerDevAddr` returns the address of the discovered device. And the field `fPeerServiceHints` returns the service hints reported by the discovered device. These can be retrieved using a get option.

Note that the device nickname always sent out is "Newton" and there is no method to retrieve the device nickname of the discovered device. If more than one qualified device is discovered, then the first device will be used.

Note

Fields used by connect are `fProbeSlots`, `fPeerServiceHints` and `fMediaBusyCheck`. Fields used by listen are `fMyServiceHints`. ◆

Connection Info Option

The next stage of the connection process involves a name lookup of the service to use, or registry of the service provided. The `TCMOIrDAConnectionInfo` option is used to specify this information.

```
class TCMOIrDAConnectionInfo : public TOption
{
   public:
             TCMOIrDAConnectionInfo();

   ULong  fMyLSAPId;
   ULong  fPeerLSAPId;
   ULong  fMyNameLength;
   ULong  fPeerNameLength;
   UChar  fClassNames[61];
};
```

The field `fMyLSAPId` may be set to 0 and the IrDA comm tool chooses a random LSAP id (between 1 and 31) or `fMyLSAPId` may be a specific value from 1 to 31. The only reason to use a specific value would be to advertise your service by number vs by name—and that is not recommended per IrDA standards. (But the option is left available in case your application needs to communicate with a dumb desktop IrDA application.)

The field `fPeerLSAPId` may be set to 0 to do the service lookup by name or may be any other value from 1 to 111 (1 to `0x6f`). If the value is non-zero, then the name lookup phase is skipped and connection is made using an LSAP id with that value. But, as noted above, per IrDA standards, services should be identified and looked up by name, not number. (But if you need to communicate with a dumb desktop IrDA application that only supports access by LSAP id (no name look up) then you will be able to.)

The fields `fMyNameLength` and `fClassNames` are used to identify the service provided by the Newton. The name is stored at `fClassNames[0]`.

The fields `fPeerNameLength` and `fClassNames` are used to identify the service provided by the peer device. The name is stored at `fClassNames[(fMyNameLength + 3) & ~4]`. In other words, the name is stored on a 4 byte boundary (a NewtonScript compatibility requirement). Don't worry NewtonScript programmers—this is done automatically for you.

Note
Fields used by connect are `fPeerLSAPId`, `fPeerNameLength` and `fClassNames`. Fields used by listen are `fMyLSAPId`, `fMyNameLength` and `fClassNames`. ◆

Serial Bit Rate Option

The final stage of the connection process is to make a connection. At this point a negotiation phase takes place between the Newton and the other IrDA device. The negotiation parameters are baud rate, data size (receive buffer size), window size (number of receive buffers), and link disconnect time.

The `TCMOSerialBitRate` and the following two options (`TCMOIrDAReceiveBuffers` and `TCMOIrDALinkDisconnect`) are used to define these negotiation parameters.

```
class TCMOSerialBitRate : public TOption
{
   public:
             TCMOSerialBitRate();

   BitRatefBitsPerSecond;
};
```

The field `fBitsPerSecond` specifies the maximum speed that the Newton is willing to communicate at. The default is 115200, which is the maximum possible on the Newton. Here are the other choices:

```
#define k9600bps    9600
#define k19200bps   19200
#define k38400bps   38400
#define k57600bps   57600
#define k115200bps  115200
```

For convenience and future compatibility, the known higher FIR speeds defined for IrDA of 576000, 1152000 and 4000000 will be accepted but treated as `k115200bps` (the highest possible speed currently supported by the hardware).

Receive Buffers Option

```
class TCMOIrDAReceiveBuffers : public TOption
{
   public:
             TCMOIrDAReceiveBuffers();

   ULong  fSize;
   ULong  fCount;
};
```

The field `fSize` specifies the size of the receive buffer(s) used by the IrDA driver. The default is 512 bytes. Valid choices are 64, 128, 256, 512, 1024 and 2048.

The field `fCount` specifies the number of receive buffers used by the IrDA comm tool (window size in IrDA-speak). The default is 1. Valid choices range from 1 to 7.

Note that you may request a large buffer (and/or a large number of buffers), but the actual size of buffer and number of buffers may be less if the negotiated speed is less than the maximum that you request.

Link Disconnect Option

```
class TCMOIrDALinkDisconnect : public TOption
{
   public:
             TCMOIrDALinkDisconnect();

   ULong  fTimeout;
};
```

The field `fTimeout` specifies the time (in seconds) before communication is terminated, if no activity is received from the peer device. Proper communication protocol between IrDA devices is to either send data or a "ready to receive" packet every 500ms (minimum). If no such activity is detected for `fTimeout` seconds, then the IrDA comm tool is disconnected. The default value is 40. Valid choices are 3, 8, 12, 16, 20, 25, 30 and 40.

Note that after 3 seconds of non-activity a disconnect warning event is sent from the IrDA comm tool.

```
// Event codes used by system
#define kEventToolSpecific1
#define kEventDisconnect2
#define kEventRelease3

// IrDATool specific event data
#define kDisconnectWarningEvent1// no resp for 3+ secs
```

Note
All of the negotiation option fields are used by both connect and listen. ◆

Connect User Data Option

There is a rarely used option that may be needed for comm tools that wish to build upon the base IrDA comm tool. This is the `TCMOIrDAConnectUserData` option, which is used during connect time to send/receive "out of band" data. See IrDA documentation to get an idea of how this would be used (specifically IrTinyTP and IrComm).

```
class TCMOIrDAConnectUserData : public TOption
{
   public:
             TCMOIrDAConnectUserData();

   ULong  fDataLength;
   UChar  fData[60];
};
```

The fields `fDataLength` and `fData` used together describe data to be sent/received during connect/listen.

The option `TCMOIrDAConnectAttrName` is intended for comm tools (built upon the IrDA comm tool) that are implementing higher layers of the IrDA stack (IrComm for example) and want to register their LSAP ids with a different attribute name. This option defines the IAS attribute string for the LSAP selector that is registered or looked up by the comm tool. The default

value for the IrDA comm tool (which is implemented at the IrLMP level) is "IrDA:IrLMP:LsapSel". For example, a comm tool that implements TinyTP would supply a value of "IrDA:TinyTP:LsapSel", per the TinyTP spec.

```
class TCMOIrDAConnectAttrName : public TOption
{
    public:
              TCMOIrDAConnectAttrName();

    ULong  fNameLength;
    UChar  fName[60];
};
```

The fields `fNameLength` and `fName` used together describe the LSAP id attribute name registered for listen/connect or looked up for connect.

Getting IrDa Tool Information

After a connection has been made, you may want to know various results of the connection. Such as, what speed am I communicating, etc. The following NewtonScript option can be used with the endpoint `Option` method to get this information. The fields are described in the previous sections.

```
local connectedOptions :=
[
    {
        label:"irdi",// IrDA discovery information
        type:  'option,
        opCode:opGetCurrent,
        result:nil,
        form:  'template,
        data:
        {
            arglist:
            [
                0,
                0,
                0,     // service hints of discovered dev
                0,     // dev addr of discovered device
```

```
            0,
        ],
        typelist:
        [
            kStruct,
            kULong,// num probe slots, def:8
            kULong,// my service hint, default: PDA
            kULong,// service hint mask
            kULong,// returned devAddr of peer device
            kULong,// use standard media busy check?
        ],
    },
},
{
    label:"sbps",// serial bit rate
    type:  'option,
    opCode:opGetCurrent,
    result:nil,
    form:  'template,
    data:
    {
        arglist:
        [
            0,     // negotiated speed
        ],
        typelist:
        [
            kStruct,
            kULong,// max negotiate speed, def: 115.2k
        ],
    },
},
{
    label:"irrb",// IrDA recv buffers info
    type:  'option,
    opCode:opGetCurrent,
    result:nil,
    form:  'template,
    data:
    {
        arglist:
        [
```

```
            0,     // negotiated buffer size
            0,     // negotiated max buffers in use
        ],
        typelist:
        [
            kStruct,
            kULong,// size of ea recv buf, def: 512
            kULong,// num recv bufs used, def: 1
        ],
    },
},
{
    label:"irld",// IrDA link disconnect threshold
    type:  'option,
    opCode:opGetCurrent,
    result:nil,
    form:  'template,
    data:
    {
        arglist:
        [
            0,     // negotiated disconnect timeout
        ],
        typelist:
        [
            kStruct,
            kULong,// Time before disc, def: 40 secs
        ],
    },
},
{
    label:"irco",// "SlowIR" connect info
    type:  'option,
    opCode:opGetCurrent,
    result:nil,
    form:  'template,
    data:
    {
        arglist:
        [
            0,     // connectOptions results
        ],
```

```
            typelist:
            [
                kStruct,
                kULong,// How did we connect?
            ],
        },
    },
];
```

Slow IR Connect Option

A note about the last option in the example above. Typically, one comm tool listens while the other connects. But, IrDA has the capability to connect if both devices are connecting (called symmetric connections here). Both connect and listen options must be specified for this to work and they both need to have and look for the same class names. But, after the connection is established, one of the two devices has (invisibly to the comm tool client) taken the role of the listener. The return value from the last option above lets you know if you are the connecter or the "listener".

```
class TCMOSlowIRConnect : public TOption
{
    public:
            TCMOSlowIRConnect();

    ULong  connectOptions;// connect-time options.
};

#defineirActiveConnection(0x02)
```

If the flag `irActiveConnection` is set in `connectOptions`, then comm tool takes role of connecter. If flag is not set then comm tool takes role of listener.

Note

SlowIR required passing the `TCMOSlowIRConnect` option with `irSymmetricConnect` set in `connectOptions` to request "symmetric connecting." But, IrDA does not—the concept is part of IrDA. And a more important note: Don't confuse the old SlowIR `TCMOSlowIRConnect` option and the new IrDA `TCMOIrDAConnectInfo` option. They are two completely different things. ◆

IrDA Statistics Option

At any time, you may want to know the status of the connection, in terms of error/retry rates. The following Newton Script option can be used with the endpoint `Option` method to get this information.

```
local statsOption :=
[
    {
        label:"irst",// IrDA/(slowIR) stats info
        type: 'option,
        opCode:opGetCurrent,
        result:nil,
        form: 'template,
        data:
        {
            arglist:
            [
                0,     // Packets received
                0,     // CRC errors detected
                0,     // Packets sent out
                0,     // Retries
                0,
                0,     // Serial errors detected
                0,     // Protocol errors
            ],
            typelist:
            [
                kStruct,
                kULong,
                kULong,
```

```
                kULong,
                kULong,
                kULong,
                kULong,
                kULong,
            ],
        },
    },
];
```

eMate 300 Multi-user ERS

In some classrooms eMate 300 units will be assigned to particular students who will keep their work on the eMate 300 for several days or longer. The eMate 300 unit will be used by several different people over the course of the day. This requires the ability to protect one students data from other students. When a user wakes up the eMate 300 they will be asked for a user name and if applicable, a password. Applications designed to be used with eMate 300, including our own NewtonWorks, will create a separate soup for each student. The connection application will only backup the work (soups) of the currently signed in student.

3rd party applications not modified to work with eMate 300 will work just as they do on other Newtons--all users will see all data.

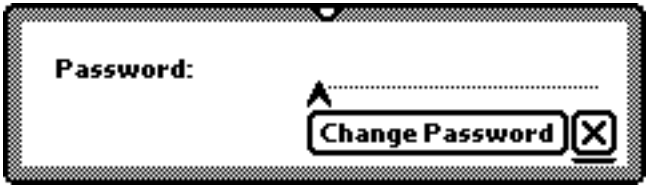
Configuration UI

The Teacher Setup application appears in both the Extras drawer (in the Unfiled Icons folder), and the Easy Extras drawer. The first time this application is opened, it prompts the user to specify a password. Figure 8-1 depicts the slip in which the user sets the password to be used.

Figure 8-1 The set password slip

When the application is opened subsequently, it displays the get password slip shown in Figure 8-2. The user can also change the current password from within this slip.

Figure 8-2 The get password slip

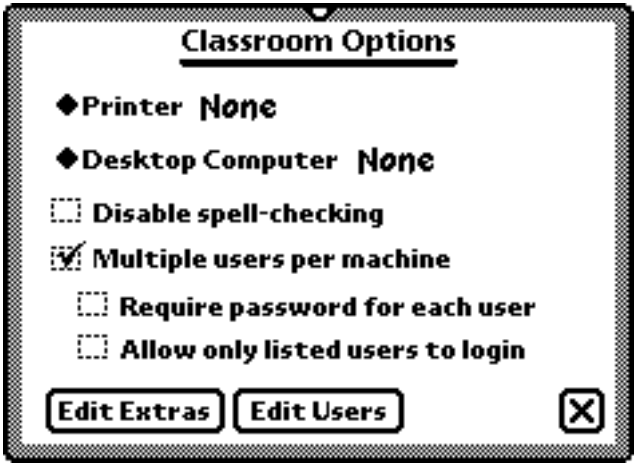


Once the password slip has been navigated successfully, the application displays the Teacher Setup slip, shown in Figure 8-3. This slip specifies whether the eMate 300 is to operate in a limited-access Classroom Mode or in Full Newton System mode.

Figure 8-3 The Teacher Setup slip

The Classroom Options button in the Teacher Setup slip displays the Classroom Options slip shown in Figure 8-4.

Figure 8-4 The Classroom Options slip



The Classroom Options slip specifies the following configuration options, which apply to Classroom mode as well as to Full Newton System mode:

- select the printer with which the eMate 300 is to connect automatically on demand.
- select the desktop computer with which the eMate 300 is to connect automatically on demand.
- disable automatic spell-checking
- specify password and user login behaviors avaliable in multi-user mode. When the “Require password for each user” checkbox is selected,the eMate 300 holds a separate password for each of multiple users. When the “Allow only listed users” checkbox is selected, the eMate 300 does not permit the creation of new users at login time.

Additionally, the Classroom Options slip provides the Edit Extras and Edit Users buttons, described immediately following.

The Edit Extras button, shown inFigure 8-5, displays a slip that specifies the applications visible to Classroom mode users of the Extras Drawer. Selecting the checkbox next to an item in this slip makes the corresponding application visible in the Extras Drawer while in Classroom mode.

Figure 8-5 The ExtrasApplications slip

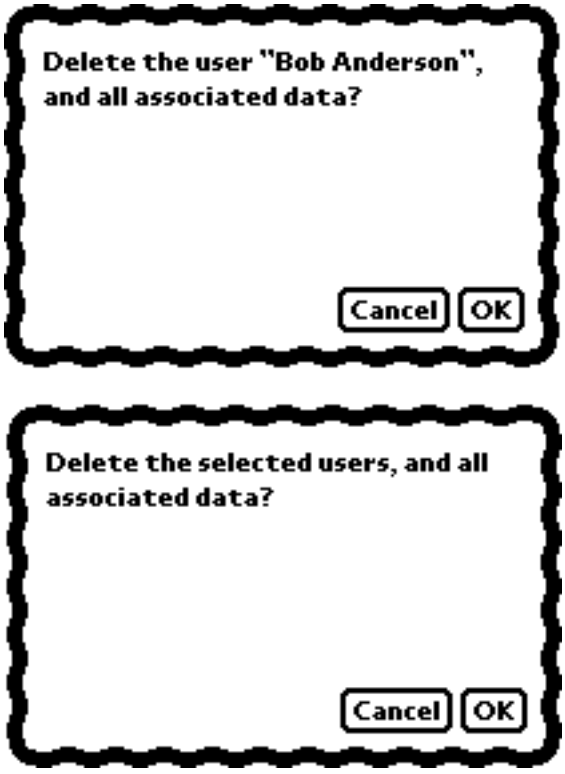
When the user taps the "Select Recommended Apps" button, the list is refreshed to show only the default applications that should appear in Classroom Extras.

The Edit Users button displays a Users slip similar to the one pictured in Figure 8-6. From this slip, the teacher can add, delete, or edit users and their associated passwords.

Figure 8-6 The Users slip

The “Delete” button removes the user account information (name and password) associated with one or more users selected in the Users slip. Before deleting this information, the Delete button warns of the destructive nature of this action by displaying an alert similar to those shown in Figure 8-7.

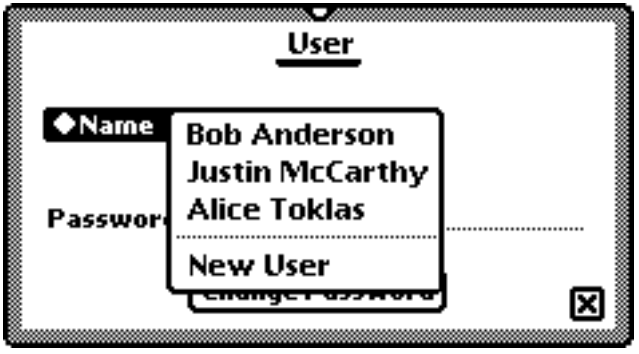
Figure 8-7 User-deletion alerts



User Login Interface

Whenever the eMate 300 wakes, it displays a login slip similar to the one shown in Figure 8-8. The Name picker in this slip is used to specify a user name. If the “Allow only listed users to login” option in the Teacher Setup application’s Classroom Options slip is selected(page 10-3), the Name picker does not include the “New User” item.

Figure 8-8 User login slip

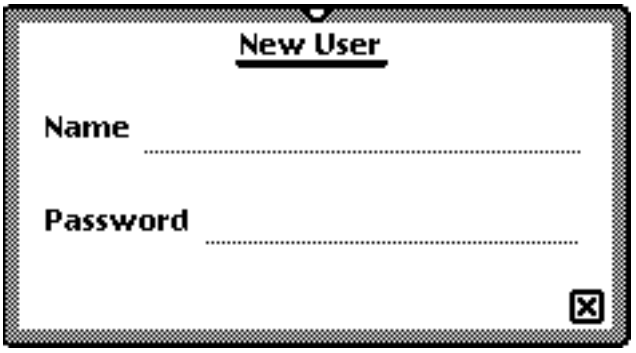


Password-protected units also require the user to enter a password in the login slip. The Password line also accepts the Teacher Setup password in lieu of an individual user's password. If the "Require Password for each user" checkbox in the Classroom Options slip is not selected, the Password line does not appear in the login slip.

The Change Password button displays the set password slip (page 10-2), which requires entry of the valid password before a change is allowed. This slip also accepts the Teacher Setup password in lieu of an individual user's password.

The New User item in the Names picker displays the New User slip shown in Figure 8-9. The New User slip is also displayed by the New User button on the Users slip (page 10-6) made available by the Edit Users button in the Classroom Options slip (page 10-3).

Figure 8-9 New user slip



Note that here, too, if "Require Password" is not on, the password line does not appear in the slip.

After logging in, the user is in the eMate 300 and can begin work. For all applications that have been written to work with eMate 300, the user will only see the data that has been created by the current user. For other applications (including all of the built-in applications) the user will see all of the data.

The data will be hidden by creating a separate soup for each user. The name of the user will be saved in the soup info in a userName slot. Each "aware" application will use the correct soup based on the current user. Each application will also register for changes to the current user slot in userconfiguration so that it can change users after the eMate 300 wakes up.

The eMate 300 connection application will know to look into the soup's info and only backup and restore the data for the current user.

If the user switches to single-user mode from multi-user mode, the generic (e.g. "NewtonWorks") soup will be shown.

Developer API

Applications will have to do a few special things to work in "multi-user" mode:

- When the application opens, check the current user and open the appropriate soup. Your application can name its soups anything you want (soup1, soup2, etc.). The only requirement is that your application provide a `GetBackupSoupNames` method that returns an array of the names of the soups for the current user, so Connection for the classroom can work properly.

GetBackupSoupNames

`GetBackupSoupNames()`

Returns an array of strings corresponding to the names of this application's soups for the current user.

- The application must also be aware that the current user can change while the application is open. To handle this, the application can register for userconfiguration changes while it's open. Here's an example of how the method that responds to userconfig changes should look:

```
func(changeSym) begin
    if changeSym = 'KCurrentUser then
        GetRoot().HomePage:MultiUserSwitch('newtWorks);
end
```

`MultiUserSwitch(`*appSymbol*`)` handles the regular case, by closing and reopening the application when the user changes.

MultiUserSwitch

`MultiUserSwitch(`*appSymbol*`)`

Performs setup and housekeeping tasks required to change users.

GetUserConfig

`GetUserConfig(`*userSym*`)`

Returns a string corresponding to the name of the specified user, or `nil` if the unit is not in multi-user mode.

userSym	Symbol specifying the current user. The only valid value for this argument is the `'KCurrentUser` symbol

GenSoupName

`GetRoot().HomePage:GenSoupName(`*soupName*`,` *currentUser*`)`

Returns a string that can be used as the soup name for a given user.

GenSoupName concatenates the soupName and the userName, putting a colon in between. If the userName is nil (indicating that the unit is not in multi-user mode), the soupName is returned without being modified.

The userName is limited to 19 characters. You must ensure that the user's soup name doesn't exceed 19 characters if it will be used in multi-user mode.

KClassroomAware slot

An application that works in Classroom mode can set the `'KClassroomAware` slot in its base view, causing the application to be checked by default in the Edit Extras slip.

Compatibility Information

For applications to be compatible with multi-user mode, they must meet several requirements:

1. There should be no folder tab in the application. Because folders are stored on a per-application basis, one user could delete a folder that another user needs, which would be very confusing.

2. The application must follow the rules in the Developer API section above.

Applications that don't meet these requirements should not set the `'KClassroomAware` slot. These applications still appear in the Edit Extras slip, but are not selected by default.